

e-maxx :: algo

Welcomes you book collected on this Site e-maxx.ru/algo (As of 26 Apr 2012 1:48).

In this book you will find the description, implementation, and evidence for a variety of algorithms, from the famous to all those that are the lot of the best specialists in olimpiadnikov and Computer Science. At the end are links to thematic literature, which can be downloaded from my site, as well as some information about me.

Happy reading!

Table of contents

Algebra

elementary algorithms

- Euler's function and its calculation
- Binary exponentiation in $O(\log N)$
- Euclid's algorithm of finding the GCD (greatest common divisor)
- Sieve of Eratosthenes
- Advanced Euclidean algorithm
- Fibonacci numbers and their rapid calculation
- Reverse ring element modulo
- Gray code
- Long arithmetic
- Discrete logarithm modulo M algorithm baby-step-giant-step Shanks for $O(\sqrt{M} \log M)$
- Diophantine equations with two unknowns: $AX + BY = C$
- Modular linear first-order equation: $AX = B$
- Chinese remainder theorem. Garner's algorithm
- Finding factorial power divider
- Ternary balanced system of value
- Factorial $N!$ modulo P for $O(P \log N)$
- Through all this mask subpatterns. Score $3N$ for the total number of all subpatterns masks
- Primitive root. Algorithm for finding
- Discrete root extract
- Sieve of Eratosthenes with linear time work

sophisticated algorithms

- The simplicity of the test BPSW numbers in $O(\log N)$
- Efficient algorithms for factorization Pollard p-1, Pollard p, Bent, Pollard Monte Carlo Farm
- Fast Fourier Transform for the $O(N \log N)$. Application to the multiplication of two polynomials or long numbers

Counts

elementary algorithms

- Search width
- Dfs
- Topological Sort
- Search the connected components

strongly connected components, bridges, etc.

- Search strongly connected component, condensation build graph for $O(N + M)$
- Search for bridges $O(N + M)$
- Search articulation points for the $O(N + M)$

- Search online bridges in O (1) average

the shortest path from one vertex

- Dijkstra finding the shortest paths from a given vertex to all other vertices in O ($N^2 + M$)
- Dijkstra sparse graph for finding the shortest paths from a given vertex to all other vertices in O ($M \log N$)
- Bellman-Ford algorithm for finding the shortest paths from a given vertex to all other vertices in O (NM)
- Leviticus algorithm finding the shortest paths from a given vertex to all other vertices in O (NM)

shortest paths between all pairs of vertices

- Finding the shortest paths between all pairs of vertices by Floyd Warshall for O (n^3)
- Counting the number of fixed-length paths between all pairs of vertices, finding the shortest paths for fixed-length O ($n^3 \log k$)

the minimum spanning tree

- Minimum spanning tree. Prim's algorithm in O (n^2) and O ($m \log n$)
- Minimum spanning tree. Kruskal's algorithm in O ($M \log N + N^2$)
- Minimum spanning tree. Kruskal's algorithm with the data structure 'system of disjoint sets' in O ($M \log N$)
- Kirchhoff matrix theorem. Finding the number of spanning trees in O (N^3)
- Prüfer code. Cayley formula. The number of ways to make a graph connected

Cycles

- Finding a negative cycle in the graph in O (NM)
- Finding Euler path or cycle in O (M)
- Checking on the acyclic graph and finding cycle for O (M)

the lowest common ancestor (LCA)

- Lowest common ancestor. For finding O (\sqrt{N}) and O ($\log N$) preprocessing with O (N)
- Lowest common ancestor. Being in O ($\log N$) preprocessing with O ($N \log N$) (binary lifting method)
- Lowest common ancestor. Finding the O (1) with preprocessing O (N) (algorithm Farah-Colton and Bender)
- Problem RMQ (Range Minimum Query - at least in the interval). Solution in O (1) preprocessing with O (N)
- Lowest common ancestor. Finding the O (1) in the offline (Tarjan algorithm)

flows and related problems

- Edmonds-Karp algorithm for finding the maximum flow in O ($N M^2$)
- Method push predpotoka finding the maximum flow in O (N^4)
- Modification of the method for pushing predpotoka O (N^3)
- Flow restrictions
- Minimum cost flow (min-cost-flow). Ways for enhancing algorithm O ($N^3 M$)
- Assignment problem. Solution with a min-cost-flow for O (N^5)
- Assignment problem. Hungarian algorithm (Kuhn algorithm) for O (N^4)
- Finding minimal cut algorithm Stoer-Wagner for O (N^3)
- Minimum cost flow, circulation of minimum cost. Algorithm for removing cycles of negative weight
- Dinitz algorithm for finding the maximum flow

matchings and related problems

- Kuhn's algorithm for finding the greatest matchings in O (NM)
- Checking graph on bipartition and split into two parts for the O (M)
- Finding the greatest weight vertex-weighted matchings in O (N^3)

- Edmonds algorithm for finding the greatest matchings in arbitrary graphs in $O(N^3)$
- Covering ways directed acyclic graph
- Tutte matrix. Randomized algorithm for finding the maximum matching in an arbitrary graph

coherence

- Edge connectivity. Properties and being
- Vertex connectivity. Properties and being
- Graphing with the stated values of the vertex and edge connectivity and the lowest degree of the vertices

K-s path inverse

problems

- Inverse problem SSSP (inverse-SSSP - the inverse problem of the shortest paths from one vertex) in $O(M)$
- The inverse problem of MST (inverse-MST - inverse problem the minimum spanning tree) in $O(NM^2)$

miscellanea

- Paint edges of the tree (data structure) - decision in $O(\log N)$
- Task 2-SAT (2-CNF). The decision is $O(N + M)$
- Heavy-light decomposition

Geometry

elementary algorithms

- Length of association on the line segments in $O(N \log N)$
- The sign area of a triangle and the predicate 'Clockwise'
- Checking on the intersection of two segments
- Finding the equation for a straight line segment
- Finding the point of intersection of two lines
- Finding the point of intersection of two segments
- Finding the area of a polygon in a simple $O(N)$
- Pick's theorem. Finding the square lattice polygon in $O(1)$
- Segments covering problem points
- Centers of gravity of polygons and polyhedra

more sophisticated algorithms

- The intersection of a circle and a straight line
- The intersection of two circles
- Construction of the convex hull algorithm for Andrew Graham-O ($N \log N$)
- Finding area combining triangles. Vertical decomposition method
- Check points on the convex polygon accessory for $O(\log N)$
- Finding the inscribed circle of a convex polygon using the ternary search for $O(N \log_2 C)$
- Finding the inscribed circle of a convex polygon sides compression method for $O(N \log N)$
- Voronoi diagram in the two-dimensional case, its properties, and application. A simple algorithm for constructing $O(N^4)$
- Finding all the faces, the outer edge of a planar graph in $O(N \log N)$
- Finding the closest-pair algorithm, divide-and-conquer in $O(N \log N)$
- Geometric transformation inversion
- The search for common tangent to two circles
- Find pairs of intersecting segments algorithm swept by lines for $O(N \log N)$

Line

- Z-line function of demand and its calculation for the O (N)
 - Prefix function, its calculation and application. Algorithm Knuth-Morris-Pratt
 - Hashing algorithms in problems on the line
 - Rabin-Karp algorithm for the substring search for O (N)
 - Expression parsing in O (N). Reverse Polish Notation
 - Suffix array. Building for the O (N log N) and application
 - Suffix automaton. Building for the O (N) and application
 - Finding all subpalindromes for O (N)
 - Lyndon decomposition. Duval algorithm. Finding the smallest cyclic shift for the O (N) time and O (1) memory
 - Algorithm Aho-Korasik
 - Suffix tree. Ukkonen's algorithm
 - Search all tandem repeats in a string algorithm Maine-Lorentz (divide-and-rule) for the O (N log N)
 - Search substring in a string using a Z-or-prefix function. Algorithm Knuth-Morris-Pratt
 - Solution of the problem "row compression"
 - Determination of the number of distinct substrings in O (N² log N)
-

Data Structures

- Sqrt-decomposition
 - Fenwick tree
 - System of disjoint sets
 - Segment tree
 - Cartesian tree (treap, deramida)
 - Modification of the stack and queue to retrieve the minimum in O (1)
 - Randomized heap
-

Algorithms on strings

- Problem RMQ (Range Minimum Query - at least on the interval)
 - Finding of the longest increasing subsequence in O (N²) and O (N log N)
 - K-th order statistic in O (N)
 - Finding of the longest increasing subsequence in O (N²) and O (N log N)
-

Dynamics

- Dynamics of the profile. Task "parquet"
 - Finding the greatest zero submatrix in O (NM)
-

Linear algebra

- Gauss solution of linear equations for the O (N³)

- Finding the rank of a matrix in O (N3)
- Calculating the determinant of the Gauss method for the O (N3)
- Calculating the determinant method for Kraut O (N3)

Numerical Methods

- Integration by Simpson's formula
- Search for roots by Newton's method (tangent)
- Ternary search

Combinatorics

- Binomial coefficients
- Catalan numbers
- Necklaces
- The alignment of elephants on a chessboard
- Right parenthesis sequence. Finding the lexicographically next, K-th, number identification
- Number of labeled graphs, connected labeled graphs, labeled graphs with K connected components
- Generating combinations of N elements
- Burnside lemma. Polya theorem
- The principle of inclusion-exclusion

Game theory

- Games on arbitrary graphs. Method for retrospective analysis of O (M)
- Theory Shpraga Grande. It

Timetables

- Johnson's problem with one machine
- Johnson's problem with two machines
- Optimal selection of jobs in certain completion times and durations of execution

Miscellanea

- Josephus problem
- Game Fifteen: the existence of solutions
- Stern-Brocot tree. Farey series
- Search subsegments array with a maximum / minimum amount for O (N)

Application

- Literature
- About the Author

Euler function

Definition

Euler function $\phi(n)$ (Sometimes denoted $\varphi(n)$ or $\text{phi}(n)$) - Is the number of properties from, relatively prime to n . In other words, the quantity of such properties in the interval $[1; n]$, greatest common divisor which is equal to unit.

The first few values of this function ([A000010 encyclopedia OEIS](#))

$$\begin{aligned}\phi(1) &= 1, \\ \phi(2) &= 1, \\ \phi(3) &= 2, \\ \phi(4) &= 2, \\ \phi(5) &= 4.\end{aligned}$$

Properties

The following three simple properties of the Euler - enough to learn how to calculate it for any number:

- If p - Prime, then $\phi(p) = p - 1$.
(This is evident, since any number except the p Relatively easy with him.)
- If p - Simple, a - A natural number, then $\phi(p^a) = p^a - p^{a-1}$.
(As to the number of p^a not only are relatively prime numbers of the form $pk(k \in \mathbb{N})$ Which $p^a/p = p^{a-1}$ pieces.)
- If a and b relatively prime, then $\phi(ab) = \phi(a)\phi(b)$ ("Multiplicative" Euler function)
This follows from Chinese Remainder Theorem. Consider an arbitrary number ab . Let u remainders a and b and on, respectively. Then coprime x if and only if $y(x \leq a \text{ and } y \leq b)$ coprime and separately, or what is the same, just with and mutually coprime.
Applying the Chinese remainder theorem, we see that any pair of numbers and uniquely corresponds number $z(z \leq ab)$, Which completes the proof.)

We can obtain the Euler function for any n through its **factorization** (Decomp. n into prime factors):

$$\text{if } n = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_k^{a_k}$$

$$\begin{aligned}\phi(n) &\stackrel{\text{all } p_i \text{- Common}}{=} \phi(p_1) \cdot \phi(p_2) \cdot \dots \cdot \phi(p_k) = \\ &= (p_1^{a_1} - p_1^{a_1-1}) \cdot (p_2^{a_2} - p_2^{a_2-1}) \cdot \dots \cdot (p_k^{a_k} - p_k^{a_k-1}) = \\ &= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \dots \cdot \left(1 - \frac{1}{p_k}\right).\end{aligned}$$

Implementation

The simplest code calculating the Euler function, factoring in the number of elementary method $O(\sqrt{n})$.

```
int phi (int n) {
    int result = n;
    for (int i = 2; i * i <= n;
         + + I) if (n % i ==
                    0) {
```

```
while (n% i == 0)
    n / = i;
result -= result / i;
```

```

        }
        if (n> 1)
            result -= result /
        n; return result;
    }
}

```

The key place for the calculation of the Euler function - is finding **factorization** number n . This is achieved in a time substantially less $O(\sqrt{n})$. Cm [Efficient algorithms for factorization](#).

Applications of the Euler function

The most famous and important property of Euler's function is expressed in **Euler's theorem**:

$$a^{\phi(m)} \equiv 1 \pmod{m},$$

where a and m coprime.

In the particular case when m simple Euler theorem is transformed into a so-called **Fermat's little theorem**:

$$a^{m-1} \equiv 1 \pmod{m}$$

Euler's theorem often occurs in practical applications, for example, see [Inverse element in the modulo](#).

Problem in online judges

Task List, which requires a function to calculate the Euler or use Euler's theorem, either by value of the Euler function to restore the original number:

- [UVA # 10179 "Irreducible Basic Fractions"](#) [Difficulty: Easy]
- [UVA # 10299 "Relatives"](#) [Difficulty: Easy]
- [UVA # 11327 "Enumerating Rational Numbers"](#) [Difficulty: Medium]
- [TIMUS # 1673 "Admission to the exam"](#) [Difficulty: high]

Binary exponentiation

Binary (binary) exponentiation - is a technique that allows to build any number n -Th power for $O(\log n)$ multiplications (instead n multiplications in the usual approach).

Furthermore, described herein is applicable to any reception **associative** operation, and not only to the multiplication number. Recall operation is called associative if for any a, b, c performed:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

The most obvious generalization - the remains of some modulo (obviously, the associativity is preserved). Next in "popularity" is a generalization to the matrix product (it is well-known associativity).

Algorithm

Note that for any number a and **even** number n doable obvious identity (which follows from the associativity of multiplication)

$$a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}$$

It is the main method in binary exponentiation. Indeed, even for n we showed how spending just one multiplication operation can reduce the problem to a lesser extent twice.

It remains to understand what to do if the power n **odd**. Here we do is very simple: move on to power $n - 1$ which will have an even:

So, we actually found the recurrence formula: the degree we move, if it is even to $n/2$ And otherwise - $n - 1$. It is understood that there will be $\log n$ transitions before we come to $n = 0$ (Based recursive formula). Thus, we have an algorithm that works for $O(\log n)$ multiplications.

Implementation

A simple recursive implementation:

```
int binpow (int a, int n) {
    if (n == 0)
        return 1;
    if (n% 2 == 1)
        return binpow (a, n-1) * A;
    else {
        int b = binpow (a, n
                        /2); return b * b;
    }
}
```

Non-recursive implementation, optimized (division by 2 replaced bit operations):

```
int binpow (int a, int n) {
    int res = 1;
    while (n)
        if (n & 1) {
            res *=
                a; - N;
        }
        else {
            a *= a;
            n >> =
                1;
        }
}
```


This implementation can be further simplified somewhat by noting that the construction of a the square is always performed, regardless of the condition worked odd n or not:

```
int binpow (int a, int n) {  
    int res = 1;  
    while (n) {  
        if (n & 1)  
            res *= a;  
        a *= a;  
        n >>= 1;  
    }  
    return res;  
}
```

Finally, it is worth noting that the binary exponentiation is implemented by Java, but only for a class of long arithmetic BigInteger (the pow this class works under binary exponentiation algorithm).

Euclid's algorithm of finding the GCD (greatest common divisor)

Given two non-negative integers a and b . Required to find their greatest common divisor, ie the largest number that divides both a , And English "greatest common divisor" is spelled "greatest common divisor", and its symbol is a common \gcd :

$$\gcd(a, b) = \max_{k=1 \dots \infty : k|a \ \& \ k|b} k$$

(Where the symbol " $|$ " Denotes divisibility, ie " $k|a$ " Denotes " k divides a ")

When it out number is zero, and the other is non-zero, their greatest common divisor, by definition, it is the second number. When the two numbers are zero, the result is undefined (infinite any suitable number), we assume in this case, the greatest common divisor of zero. Therefore we can talk about this rule: if one of the numbers is zero, then their greatest common divisor is the second number.

Euclid's algorithm, discussed below, solves the problem of finding the greatest common divisor of two numbers a and b for $O(\log \min(a, b))$.

This algorithm was first described in the book of Euclid's "Elements" (about 300 BC), although it may, this algorithm has an earlier origin.

Algorithm

The algorithm is extremely simple and is described by the following formula:

$$\gcd(a, b) = \begin{cases} a, & \text{if } b=0 \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Implementation

```
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a% b);
}
```

Using the ternary conditional operator C + +, the algorithm can be written even shorter:

```
int gcd (int a, int b) {
    return b? gcd (b, a% b) : A;
}
```

Finally, we give shape and a non-recursive algorithm:

```
int gcd (int a, int b) {
    while (b) {
        a% = b; swap
        (a, b);
    }
    return a;
}
```

Proof of correctness

Please note that for each iteration of the Euclidean algorithm its second argument is strictly decreasing. Consequently, since it is non-negative, then the Euclidean algorithm **always ends**.

For **correctness proofs** we need to show that for any

$$a \geq 0, b > 0$$

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

We show that the quantity in the left-hand side is divided by the present on the right and the right-hand - is divisible by standing on the left. Obviously, it would mean that the left and right sides of the same, and that proves the correctness of Euclid's algorithm.

Denote $d = \gcd(a, b)$. Then, by definition, $d|a$ and $d|b$. Next, we

expand the remainder of the division a on b through their private:

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

But then it follows:

$$d | (a \bmod b)$$

So, recalling the statement $d|b$ We obtain the system:

$$p, q, r \text{ complete } p|q \text{ and } p|r,$$

We now use the following simple fact: if for any three numbers and then performed $\gcd(q, r)$. In our situation, we obtain

Or by substituting d its definition as $\gcd(a, b)$, We obtain:

$$\gcd(a, b) | \gcd(b, a \bmod b)$$

So, we spent half the evidence to show that the left side of the right divides. The second half of the proof is similar.

Operation time

Time of the algorithm is evaluated **Theorem Lame** Which establishes a surprising connection Euclid's algorithm and the Fibonacci sequence:

If $a > b \geq 1$ and $b < F_n$ for some n , The Euclidean algorithm performs no more $n - 2$ recursive calls. Moreover, we can show that the upper bound of this theorem - optimal. At it will be performed $n - 2$ recursive call. In other words, **serial Fibonacci numbers - the worst input** for the Euclidean algorithm.

$$a = F_n, b = F_{n-1}$$

Given that the Fibonacci numbers grow exponentially (as a constant raising n), We find that the Euclidean algorithm runs in $O(\log \min(a, b))$ multiplication operations.

LCM (least common multiple)

Calculation of the least common multiple (least common multiplier, lcm) reduces to the calculation the following simple statement:

$$\text{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

Thus, the calculation of the NOC can also be done using the Euclidean algorithm, with the same asymptotic behavior:

```
int lcm (int a, int b) {
    return a / gcd (a, b) * B;
}
```

(Here divided into profitable first, and only then are multiplied by b Because it helps to avoid overflow in some cases)

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Construction and analysis** [2005]

Sieve of Eratosthenes

Sieve of Eratosthenes - an algorithm for finding all primes in the interval for $[1; n]$

$O(n \log \log n)$ operations.

The idea is simple - to write a series of numbers $1, \dots, n$. And will strike out the first all numbers divisible by 2. Except the number 2. And then dividing by 3. Except the number 3. Then 5. Then 7, 11. And all other simple to n .

Implementation

Immediately give the implementation of the algorithm:

```
int n;
vector<char> Prime (n
+1, true); prime[0] = Prime[1] =
false;
for (int i = 2; i <= n; +
    + i) if
(prime[i])
    if (i * 111 * i <= n)
        for (int j = i * i; j <= n;
            j += i) prime[j] =
            false;
```

This code first checks all numbers except zero and one, as simple, and then begins the process of sifting composite numbers. To do this, we loop through all the numbers from 2 to n . And if the current number of i prime, then mark all the numbers that are multiples of him as a constituent.

At the same time we start walking from $\Theta(n)$. As fewer multiples necessarily have a prime divisor $O(n \log \log n)$. In the

is smaller and hence, they have been eliminated previously. (But because it is easily cramp type before the second nested loop is an additional check using type .)

With this implementation, the algorithm uses memory (obviously) and performs Action
(This is proved in the next section).

$$\frac{n}{p}$$

We prove that the asymptotic behavior of the algorithm is $O(n \log \log n)$.

So, for each prime $p \leq n$ will run the inner loop, which make actions. Consequently, we need to estimate the following value:

Let us recall here two known facts: that the number of primes less than or equal to approximately equal $\frac{n}{\ln n}$. And that - the prime number is approximately equal to $\frac{1}{k \ln k}$. (This follows from the first statement). Then the sum can be written thus $\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{1}{p} \approx \frac{1}{2} \sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}$

Here we have identified the first prime of the sum, since the resulting in a division by zero.

according approximation

will ,

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_2^{\frac{n}{\ln n}} \frac{1}{k \ln k} dk.$$

We now estimate that amount by the integral of the same function on from to such an approximation, because, in fact, refers to the amount of his integral approximation formula of rectangles):

(We can produce

Primitive integrand has order, we obtain:

$\ln \ln k$. Performing substitution and removing members of the smaller

Now, returning to the initial sum, we obtain an approximate estimate of its:

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{n}{p} \approx n \ln \ln n + o(n),$$

QED.

More rigorous proof (and provide a more accurate estimate of up to constant factors) can be found in the book of Hardy and Wright "An Introduction to the Theory of Numbers" (p. 349).

Various optimizations sieve of Eratosthenes

The biggest drawback of the algorithm - that he "walks" from memory, constantly going beyond the cache, causing constant hidden in $O(n \log \log n)$ is relatively large.

Furthermore, for sufficiently large n bottleneck becomes memory usage.

The following are the methods to both reduce the number of operations performed, and significantly reduce memory consumption.

Sifting simple to the root

The most obvious point - that in order to find all simple to n enough to perform simple screening only, not the root of the superior n .

Thus, change the outer loop of the algorithm:

```
for (int i = 2; i * i <= n; ++i)
```

On the asymptotic behavior of this optimization does not affect (indeed, repeating the above proof, we obtain an estimate that, by the properties of logarithms, asymptotically is the same), although the number of transactions decreased markedly.

Sieve only odd numbers

Since all of the even numbers except 2-components, we can not process any way at all even numbers and odd numbers operate only.

First, it will halve the amount of memory required. Second, it makes the algorithm will reduce the number of operations by about half.

Reducing the amount of memory consumed

Note that Eratosthenes algorithm actually operates with n memory bits. Consequently, it can save significant memory consumption, not keeping n byte - booleans and n bit, i.e. $n/8$ bytes of memory.

However, this approach - **"Bit compression"** - Substantially complicate handling these bits. Any read or write bit will be of a few arithmetic operations, which will eventually lead to a slowdown of the algorithm.

Thus, this approach is justified only if n so large that n bytes of memory to allocate anymore. Saving memory (in 8 times), we will pay for it a substantial slowing of the algorithm.

In conclusion, it is worth noting that the language of C++ containers have already been implemented, automatically asking bit compression: `vector<bool>` and `bitset<>`. However, if speed is important, it is better to implement the compression bit manually, using bit operations - today compilers still unable to generate code fast enough.

Block sieve

Optimization of "simple screening to the root" implies that there is no need to store all the time all array $\text{prime}[1 \dots n]$. To perform screening sufficient to store only simple to root out, that is $\text{prime}[1 \dots \sqrt{n}]$ And the remainder of the array prime building block at a time, keeping the current time Only one block. - Constant that determines the size of the block, then all will $\lceil \frac{n}{s} \rceil$ blocks, the first $\lceil \frac{n}{s} \rceil$ block k Contains the numbers in the interval $[ks; ks + s - 1]$. Will process the blocks at a time, ie for each of k will go through all the simple (from 1 to \sqrt{n}) And perform their screening only within a current block. Gently handle the first unit costs - first, from simple $[1; \sqrt{n}]$ should not remove themselves, and secondly, the number 0 and 1 should be marked as not particularly simple. When processing the last block should also not forget that the last desired number n not necessarily in the end block.

We present the implementation of the sieve block. The program reads the number n and finds a number of simple 1 to n :

```

const int SQRT_MAXN = 100000; // square of a maximum value N const
int S = 10000;
bool nprime[SQRT_MAXN], Bl[S];
int primes[SQRT_MAXN], Cnt;
int main() {

    int n;
    cin >> N;
    int nsqrt = (int) sqrt (n + .0);
    for (int i = 2; i <= nsqrt; ++ I)
        if (! Nprime[i]) {
            primes[cnt + +] =
            I;
            if (i * 111 * i <= nsqrt)
                for (int j = i * i; j <= nsqrt;
                     j += i) nprime[j] =
                    true;
        }

    int result = 0;
    for (int k = 0, Maxk = n / S; k <= maxk;
         + + K) { memset (bl, 0, sizeof
                           bl);
        int start = k * S;
        for (int i = 0; i < cnt; + + I) {
            int start_idx = (start + primes[i] - 1) / Primes[i];
            int j = max(start_idx, 2) * Primes[i] - Start;
            for (; j < S; j +=
                  primes[i]) bl[j] =
                    true;
        }
        if (k == 0)
            bl[0] = Bl[1] = true;
        for (int i = 0; i < S && start + i <=
             n; + + I) if (! Bl[i])
                + + Result;
    }
    cout << Result;
}

```

Asymptotics sieve block is the same as usual and the sieve of Eratosthenes (unless, of course, the size of S units is not will be quite small), but memory usage will be reduced to $O(\sqrt{n} + s)$ and decrease "wandering" from memory. On the other hand, for each block for each of simple $[1; \sqrt{n}]$ will run the division, that will greatly affect in a smaller unit. Consequently, the choice of the constants s must strike a balance.

Experiments show that the best speed of operation is achieved when the s has a value of about 10^4 to 10^5 .

Upgrade to linear time work

Eratosthenes algorithm can be converted to a different algorithm, which is already operational in linear time - see article

"[Sieve of Eratosthenes with linear time work](#)". (However, this algorithm has drawbacks.)

Advanced Euclidean algorithm

While "Normal" Euclid's algorithm simply finds the greatest common divisor of two numbers a and b , Extended Euclidean algorithm finds the GCD also factors in addition to x and y such that:

$$a \cdot x + b \cdot y = \gcd(a, b).$$

It finds the coefficients by which the GCD of two numbers expressed in terms of the numbers themselves.

Algorithm

Make the calculation of these coefficients in the Euclidean algorithm is simple enough to derive formulas for which they are changed during the transition from the pairs (a, b) to $(b \% a, a)$:
a pair (Percent sign we mean taking the remainder from the division).

Thus, suppose we have found (x_1, y_1) problems for the new couple $(b \% a, a)$:
solution $(b \% a) \cdot x_1 + a \cdot y_1 = g$,

and want to get a solution (x, y) for our couples (a, b) :
 $a \cdot x + b \cdot y = g$.

To do this, we transform the value $b \% a$:

$$b \% a = b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a.$$

Substituting this into the above expression x_1 and y_1 and obtain:

$$g = (b \% a) \cdot x_1 + a \cdot y_1 = \left(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a \right) \cdot x_1 + a \cdot y_1,$$

and performing regrouping terms, we obtain:

$$g = b \cdot x_1 + a \cdot \left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1 \right).$$

Comparing this with the original expression of the unknown x and y , We obtain the desired expression:

$$\begin{cases} x = y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1, \\ y = x_1. \end{cases}$$

Implementation

```
int gcd (int a, int b, int & x, int & y) {
    if (a == 0) {
        x = 0; y =
            1; return b;
    }
    int x1, y1;
    int d = gcd (b % a, a, x1,
    y1); x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

b

This is a recursive function, which still returns the GCD of the numbers a and, but apart from that - as desired coefficients x and y as function parameters passed by reference.

Recursion base - case a . Then the GCD is, and obviously required coefficients x and y are equal and 1 respectively. In other cases, the usual solution works, and the coefficients are translated at the above formulas.

Advanced Euclidean algorithm in this implementation works correctly even for negative numbers.

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Construction and analysis** [2005]

Fibonacci numbers

Definition

The Fibonacci sequence is defined as follows:

$$\begin{aligned}F_0 &= 0, \\F_1 &= 1, \\F_n &= F_{n-1} + F_{n-2}.\end{aligned}$$

The first few of its members:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, , 55, 89, ...

History

These numbers are entered into in 1202 by Leonardo Fibonacci (Leonardo Fibonacci) (also known as Leonardo of Pisa (Leonardo Pisano)). However, thanks to a 19th-century mathematician Luca (Lucas) the name "Fibonacci numbers" became common.

However, the number of Indian mathematicians mentioned earlier in this sequence: Gopal (Gopala) to 1,135 city, Hemachandra (Hemachandra) - in 1150

Fibonacci numbers in nature

Fibonacci himself mentioned these numbers in connection with this challenge: "A man planted a couple of rabbits in the paddock, surrounded on all sides by a wall. How many pairs of rabbits can produce per year to light the couple, if you know that every month, starting from the second, each pair rabbits produces one pair of light? ". The solution to this problem will be the number sequence, now called in his honor. However, the situation described by Fibonacci - more mind game than real nature.

Indian mathematicians Gopala and Hemachandra mentioned this number sequence in relation to the number of rhythmic patterns, resulting from the alternation of long and short syllables in verse or strong and weak beats in music. The number of pictures having a generally n shares power F_n .

Fibonacci numbers appear in the work of Kepler in 1611, which reflected on the numbers found in nature (work "On the hexagonal flakes").

An interesting example is plants - yarrow, in which the number of stems (and hence flowers) is always a Fibonacci number. The reason is simple: while initially with a single stem, this stem is then divided by two, and then branches off from the main stem of another, then the first two stem branch again, then all the stems, but the last two branch, and so on. Thus, each stalk after his

the appearance of "missing" one branch, and then begins to divide at each level of branching that results in the Fibonacci numbers.

Generally speaking, many colors (eg, lilies), the number of petals is a way or another Fibonacci number.

Also botanically known phenomenon phyllotaxis"". As an example, the location of sunflower seeds: if you look down on their location, you can see simultaneously two series spirals (like overlapping): some twisted clockwise, the other - against. It turns out that the number of these spirals is roughly equal to two consecutive Fibonacci numbers: 34 and 55 or 89 and 144. Similar facts are true for some other colors, as well as pine cones, broccoli, pineapple, etc.

For many plants (according to some sources, 90% of them) are true and an interesting fact. Consider any leaf, and will descend from him down until we reach the sheet disposed on the stem in the same way (ie, directed exactly in the same direction). Along the way, we assume that all the leaves, gets us (ie, located at an altitude between the start and end sheet), but arranged differently. Numbering them, we will gradually make the turns around the stem (as the leaves are arranged on the stem in a spiral). Depending on whether the windings perform clockwise or counterclockwise will receive a different number of turns. But it turns out that the number of turns made by us in a clockwise direction, the number of turns made by counterclockwise, and the number of leaves encountered form three consecutive Fibonacci numbers.

However, it should be noted that there are plants for which the above calculations give the number of very different sequences and therefore can not be said that the phenomenon of phyllotaxis is the law - it

rather entertaining trend.

Properties

Fibonacci numbers have many interesting mathematical properties.

Here are some of them:

- Cassini ratio:

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n.$$

- Rule of "addition":

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$$

- From the previous equality in $k = n$ follows:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}).$$

- From the previous equality can be obtained by induction that F_{nk} always divisible F_n .

- The opposite is true of the previous statement: if F_m

Multiples F_n . Then m Multiples n .

- GCD-equality:

$$\gcd(F_m, F_n) = F_{\gcd(m, n)}.$$

- With respect to the Euclidean algorithm Fibonacci numbers have the remarkable property that they are the worst input data for this algorithm (see "Lame theorem" in [Euclidean algorithm](#)).

Fibonaccimal value

Zeckendorf theorem claims that every positive integer n can be represented in a unique way as a sum of Fibonacci numbers:

$$N = F_{k_1} + F_{k_2} + \dots + F_{k_r}$$

where $k_1 \geq k_2 + 2, k_2 \geq k_3 + 2, \dots, k_r \geq 2$ (ie can not be used in the recording of two adjacent Fibonacci numbers).

This implies that any number can be uniquely written **Fibonacci system value**, eg

$$\begin{aligned} 9 &= 8 + 1 = F_6 + F_1 = (10001)_F, \\ 6 &= 5 + 1 = F_5 + F_1 = (1001)_F, \\ 19 &= 13 + 5 + 1 = F_7 + F_5 + F_1 = (101001)_F, \end{aligned}$$

and in any number can not go two units in a row.

It is easy to get and usually adding one to the number in the Fibonacci number system, if the minor number is 0, it is replaced by 1, and if equal to 1 (ie at the end worth 01), then 01 is replaced by 10. Then "fix" record sequentially correcting all 011 100. As a result, the linear time is obtained by recording a new number.

Translation numbers in the Fibonacci number system is as simple as "greedy" algorithm: just iterate through the Fibonacci numbers from larger to smaller and if some, it is included in the record number n . And we subtract F_k from n and continue the search.

The formula for the n-th Fibonacci number

Formula radicals through

There is a wonderful formula, called after the French mathematician Binet (Binet), although it was known to him Moivre (Moivre):

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

This formula is easily proved by induction, but you can bring it by the concept of forming functions or with a solution of the functional equation.

Immediately you will notice that the second term is always less than 1 in absolute value, and moreover, decreases very rapidly (exponentially). This implies that the value of the first term F_n gives the "almost" value. This can be written simply as:

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} \right],$$

where the square brackets denote rounding to the nearest integer.

However, for practical use in the calculation of these formulas hardly suitable, because they require very high precision work with fractional numbers.

Array formula for the Fibonacci numbers

It is easy to prove the following matrix equation:

$$(F_{n-2} \ F_{n-1}) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_{n-1} \ F_n).$$

But then, denoting

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

obtain

Thus, to find the Fibonacci number F_n is necessary to build a matrix

Remembering that the construction of the matrix in the n -th power can be accomplished

the power) it turns out, that n th Fibonacci number can be easily calculated for $O(\log n)$ using only integer arithmetic.

P the power

$O(\log n)$ (Cf. [Binary construction](#))

Frequency of the Fibonacci sequence modulo

Consider the Fibonacci sequence and wherein the period begins with the F_i some modulo. We prove that it is periodic, p pairs F_0 .

We prove this by contradiction. Consider $p^2 + 1$ pairs Fibonacci numbers taken modulo p :

$$(F_1, F_2), (F_2, F_3), \dots, (F_{p^2+1}, F_{p^2+2}).$$

Since modulo p can only be different pairs, among this sequence there are at least two identical pairs. This already means that the sequence is periodic.

We now choose among all such identical pairs of two identical pairs with the lowest numbers. Let this pair with some rooms and . We prove that . Indeed, otherwise

$$\frac{(F_a, F_{a+1}) \ (F_b, F_{b+1})}{(F_{a-1}, F_a) \ d(F_{b-1}, F_b)} \stackrel{a=1}{\longrightarrow}$$

for them there the previous couple which, by property of Fibonacci numbers as are equal to each other. However, this contradicts the fact that we chose the matching pairs with the lowest numbers, as required.

Literature

- Ronald Graham, Donald Knuth, and Oren Patashnik. **Concrete Mathematics** [1998]

Reverse ring element modulo

Definition

Suppose we are given a natural module m And consider the ring formed by the module (i.e., consisting of a number of 0 to $m - 1$). Then for some elements of this ring can be found **inverse**.

The inverse of the number a modulo m called a number b That:

$$a \cdot b \equiv 1 \pmod{m},$$

and it is often denoted by a^{-1} .

It is clear that for the zero inverse element does not exist ever; for the remaining elements of the inverse can exist or not. It is argued that the inverse exists only for those elements a That

coprime module m .

Consider the following two ways of finding the inverse element employed, provided that it exists.

Finally, consider an algorithm that allows you to find all the numbers back to some modulo linear time.

Finding using the Extended Euclidean algorithm

Consider the auxiliary equation (relatively unknown x and y)

It linear Diophantine equation of the second order. As shown in a related article from terms $\gcd(a, m) = 1$ that this equation has a solution which can be found using **The extended Euclidean algorithm** (Hence the same incidentally, follows when $\gcd(a, m) \neq 1$ Decisions, and therefore and an inverse element does not exist).

On the other hand, if we take from both sides of the residue modulo m , We obtain:

$$a \cdot x = 1 \pmod{m}.$$

Thus, found x and will be the inverse of a .

Implementation (including that found x we must take the modulo m And x may be negative):

```
int x, y;
int g = gcdex (a, m, x, y);
if (g != 1)
    cout << "No solution";
else {
    x = (x % m + m) % M;
    cout << x;
}
```

Asymptotics of the solutions obtained $O(\log m)$.

Finding the binary exponentiation

We use Euler's theorem:

$$a^{\phi(m)} \equiv 1 \pmod{m},$$

which is true just in the case of relatively prime a and m .

Incidentally, in the case of a simple module m we get a more simple statement - Fermat's little theorem:

$$a^{m-1} \equiv 1 \pmod{m}.$$

Multiply both sides of each of the equations on a^{-1} , We obtain:

- for any module m :

$$a^{\phi(m)-1} \equiv a^{-1} \pmod{m},$$

- for a simple module m :

$$a^{m-2} \equiv a^{-1} \pmod{m}.$$

Thus, we got the formula for direct calculation of the inverse. For practical applications typically use effective [algorithm for binary exponentiation](#), Which in this

case will bring about exponentiation for $O(\log m)$.

This method appears to be somewhat easier described in the previous paragraph, but it requires knowledge of the values of the Euler function that actually requires factorization module m That can sometimes be quite challenging.

If the quotient of the number is known, then this method also works for the asymptotic behavior $O(\log m)$.

$$m.[1; m - 1]$$

$$\begin{aligned} O(m) \cdot & \quad O(m \log m) \\ & \text{opposite to the desired} \\ r[i] & \text{number} \quad i \text{ modulo. } m \end{aligned}$$

Decision it is as follows. We denote

Then for $i > 1$ true identity:

Implementation this amazingly concise solutions:

```
r[1] = 1;
for (int i = 2; i < m; ++i)
    r[i] = (m - (m / i) * R[m % i] % M) % M;
```

Proof this decision is a chain of simple transformations: We write the value

$m \bmod i$:

$$m \bmod i = m - \left\lfloor \frac{m}{i} \right\rfloor \cdot i,$$

whence, taking both sides modulo m , We obtain:

$$m \bmod i = - \left\lfloor \frac{m}{i} \right\rfloor \cdot i. \pmod{m}$$

Multiplying both sides by the inverse of i and its inverse $(m \bmod i)$, We obtain the desired formula:

$$r[i] = - \left\lfloor \frac{m}{i} \right\rfloor \cdot r[m \bmod i], \pmod{m}$$

QED.

Gray code

Definition

Gray code numbering system is defined to be a non-negative integers, where codes of two adjacent numbers differ in exactly one bit.

For example, for numbers of length 3 bits have a sequence of Gray codes: 000,001,011,010,110, 111,101,100. Eg $G(4) = 6$.

This code was invented by Frank Gray (Frank Gray) in 1953.

Finding the Gray code

Consider the number of bits and the bits $G(n)$ of the number n . Note that the first bit $i + 1$ equal to unity only when Th bit equal to one and $G(n) = n \oplus (n >> 1)$. First is zero, or vice versa (Th bit is zero, and First is unity). Thus, we have:

```
int g (int n) {
    return n ^ (n >> 1);
}
```

Finding the inverse Gray code

Required by the Gray code g restore the original number n .

We will go from older to younger bits (let LSB is 1, and the oldest k). We obtain the following relations between the bits n_i number n and bats g_i number g :

$$\begin{aligned} n_k &= g_k, \\ n_{k-1} &= g_{k-1} \oplus n_k = g_k \oplus g_{k-1}, \\ n_{k-2} &= g_{k-2} \oplus n_{k-1} = g_k \oplus g_{k-1} \oplus g_{k-2}, \\ n_{k-3} &= g_{k-3} \oplus n_{k-2} = g_k \oplus g_{k-1} \oplus g_{k-2} \oplus g_{k-3}, \end{aligned}$$

B a program code is the easiest way to write this:

```
int rev_g (int g) {
    int n = 0;
    for ( ; g; g >>
        =1) n ^
        = g;
    return n;
}
```

Applications

Gray codes have several applications in different areas, sometimes quite unexpected:

- n Bit Gray code corresponds to a Hamiltonian cycle n -Cube.
- In the art, Gray codes are used for **minimizing the error** when converting analog signals to digital signals (e.g. in sensors). In particular, the Gray codes were discovered in connection with this application.
- Gray codes are used in solving the problem of **Tower of Hanoi**.

Let - number of disks. Let's start with the Gray code length, consisting of zeros (ie $G(i)$), And will move i

Gray Codes (from proceed to). With every bit of the current CB Gray code - first drive (and most significant bit corresponds to the smallest disk size and the oldest bat

- Maximum). Since at each step exactly one bit is changed, then we can understand the bit change i as moving i Th disc. Note that for all drives except the smallest, at each step there is exactly

one option course (except for the starting and final products). For there is always the smallest drive
two versions of course, but there is progress in the strategy of choice, always leads to the answer: if n
odd, then the sequence of movements of the smallest drive has the form (where f - the final product, t -
- Starter rod t - The final rod r - The remainder of the rod), and if n even, then
 $f \rightarrow r \rightarrow t \rightarrow f \rightarrow r \rightarrow t \rightarrow \dots$

- Gray codes are also used in the theory **Genetic Algorithms**.

Problem in online judges

List of tasks that can be taken, using Gray codes: • [SGU](#)

249 "Matrix" [Difficulty: average]

Long arithmetic

Long arithmetic - a set of tools (data structures and algorithms) that allow you to work with numbers much larger quantities than it allows standard data types.

Types of long integer arithmetic

Generally speaking, even just in the Olympiad set of problems is large enough, so we make a classification of different types of long arithmetic.

Classical long arithmetic

The basic idea is that the number is stored as an array of digits it.

Numbers can be used from one system or another value, commonly used decimal system and its power (ten thousand billion), or binary system.

Operations on numbers in the form of a long arithmetic made using "school" algorithms for addition, subtraction, multiplication, long division. However, they are also useful for fast multiplication algorithms: [Fast Fourier transform](#) and the Karatsuba algorithm.

Described here only work with non-negative long numbers. Support for negative numbers must enter and maintain additional flag "negativity" numbers, or else work in complementary codes.

Data structure

Keep long numbers will be in the form of a Vector of Numbers *int* Where each element - is one digit number.

```
typedef vector <int> Lnum;
```

To increase efficiency in the system will work in base billion, i.e. each element of the vector *lnum* contains not one, but 9 numbers:

```
const int base = 1000*1000*1000;
```

The numbers will be stored in a vector in such a manner that at first there are the least significant digit (ie, ones, tens, hundreds, etc.).

Furthermore, all the operations are implemented in such a manner that after any of them leading zeros (ie excess leading zeros) are not (of course under the assumption that prior to each leading zeros is also available). It should be noted that the implementation representation for the number zero is well supported just two representations: an empty vector of numbers and digits vector containing a single element - zero.

Output

The most simple - a conclusion of a long number.

First, we simply display the last element of the vector (or 0 if the vector is empty), and then derive all the remaining elements of the vector, supplementing them with zeros to 9 characters:

```
printf ("% D", A.empty() ? 0 :  
A.back()); for (int i =(int)a.size()-2;  
i>=0; - I)  
    printf ("% 09d", A[i]);
```

(Here a small fine point: not to forget to write down the cast
number *a.size()* are unsigned, and if *a.size() ≤ 1* Then the subtraction overflows)

(*int*) Because otherwise

Reading

Reads a string in *string* And then convert it into a vector:

```
for (int i =(int)s.length(); i>0; i-=9)
```

```

if (i < 9)
    a.push_back (atoi (s.substr (0, I). C_str ()));
else
    a.push_back (atoi (s.substr (i-9,9). C_str ()));

```

If we use instead *string* array *char*'S, the code will be even smaller:

```

for (int i = (int)strlen(s); i>0; i-=9)
    { s[i] = 0;
      a.push_back (atoi (i>=9 ? s + i-9 : S));
}

```

If the input number has leading zeros may be, they can remove, after reading the following way:

```

while (a.size() > 1 && A.back () == 0)
    a.pop_back ();

```

Addition

Adds to the number of *a* number *b* and stores the result in *a*:

```

int carry = 0;
for (size_t i =0; i <max(a.size(), B.size()) || Carry; +
     + I) { if (i == a.size())
            a.push_back (0);
            a[i] += Carry + (i <b.size() ? b[i] : 0);
            carry = a[i] >= Base;
            if (carry) a[i] -= Base;
}

```

Subtraction

Takes the number of *a* number *b* (*a* \geq *b*), And stores the result in *a*:

```

int carry = 0;
for (size_t i =0; i <b.size() || Carry; ++ I) {
    a[i] -= Carry + (i <b.size() ? b[i] : 0);
    carry = a[i] < 0;
    if (carry) a[i] += Base;
}
while (a.size() > 1 && A.back () == 0)
    a.pop_back ();

```

Here we are after subtracting remove leading zeros to maintain the predicate that they do not exist.

Multiplying the length of a short

Multiplies long *a* briefly *b* (*b* < base), And stores the result in *a*:

```

int carry = 0;
for (size_t i =0; i <a.size() || Carry; + +
     I) { if (i == a.size())
            a.push_back (0);
            long long cur = carry + a[i] * 111 * b;
            a[i] = int (cur% base);
            carry = int (cur / base);
}
while (a.size() > 1 && A.back () == 0)
    a.pop_back ();

```

Here we are after the division remove leading zeros to maintain the predicate that they do not exist.

(Note: the way **additional optimization**. If speed is critical, you can try to replace two division one: count only the integer portion of a division (in the code is a variable *carry*), And then to calculate it by the remainder of dividing (using a multiplication operation).

Typically, this technique allows faster code, but not very much.)

Multiplication of two long numbers

Multiplies a on b and stores the result c :

```
lnum c (a.size() + B.size());
for (size_t i = 0; i < a.size(); ++i)
    for (int j = 0, Carry = 0; j < (int)b.size() || Carry; ++j) {
        long long cur = c[i + j] + A[i] * 111 * (j < (int)b.size()
? b[j] : 0) + Carry;
        c[i + j] = int (cur % base); carry = int (cur / base);
    }
while (c.size() > 1 && C.back() == 0)
    c.pop_back();
```

Long division for a short

Divides long a briefly b ($b < \text{base}$), Private stores a residue in $carry$:

```
int carry = 0;
for (int i = (int)a.size() - 1; i >= 0; - I) {
    long long cur = a[i] + Carry * 111 * base;
    a[i] = int (cur / b);
    carry = int (cur % b);
}
while (a.size() > 1 && A.back() == 0)
    a.pop_back();
```

Long arithmetic in factored form

The idea here is not to store the number itself, and its factorization, ie each incoming power it simple.

This method is also very simple to implement, and it is very easy to make multiplication and division, but it is impossible to perform the addition or subtraction. On the other hand, this method saves memory in comparison with a "classic" approach, and allows the multiplication and division significantly (asymptotically) faster.

This method is often used when you need to make on the delicate division module: then enough to store a number in the form of powers to the prime divisors of the module, and another number - balance on the same module.

Long arithmetic system of simple modules (Chinese theorem or scheme Garner)

The bottom line is that some selected system modules (usually small, fit into standard data types), and the number is stored as a vector of residuals from dividing it by each of these modules.

According to the Chinese Remainder Theorem, it is enough to uniquely store any number between 0 and the product of these modules minus one. Thus there [Garner's algorithm](#) which allows to produce

is the restoration of a modular form into the usual "classical" form number.

Thus, this method saves memory compared to the "classic" long arithmetics (although in some cases not as radically as the factorization method). Furthermore, in a modular fashion, you can very quickly make addition, subtraction and multiplication, - all for adding identical time asymptotically proportional to the number of modules in the system.

However, given the price of all this is very time-consuming translation number of this modular form in the usual form, which, in addition to considerable time costs also need the implementation of "classical" long arithmetic multiplication.

In addition, to produce **division** numbers in this representation system for simple modules is not possible.

Types long fractional arithmetic

Operations on fractional numbers found in the Olympiad problems are much less common, and work with huge

fractional numbers is much more complicated, so only found in specific competitions subset of fractional long arithmetic.

Long arithmetic in an irreducible fraction

The number appears as an irreducible fraction, where $\frac{a}{b}$ and b - Integers. Then all operations on fractional numbers easily reduced to operations on the numerator and denominator of these fractions.

Usually, when this storage numerator and denominator have to use long arithmetic, but, however, it is the easiest form - "classic" long arithmetic, although sometimes is sufficiently embedded 64-bit numeric type.

Isolation of floating point position as a separate type

Sometimes the problem is required to make calculations with very large or very small numbers, but it does not prevent them from overflowing. Built-byte type ~~is known~~ $double$ allows the exponent value in the range $[-308; 308]$. What can sometimes be enough.

Reception, in fact, very simple - introduce another integer variable responsible for the exponent, and after each operation, the fractional number of "normal", ie returns to the segment, $[0.1; 1)$ by increasing or decreasing exponential.

When multiplying or dividing two such numbers should add up accordingly or subtract their exponents. When adding or subtracting before proceeding number should lead to an exponential one, which one of them is multiplied by 10 raising exponential difference.

Finally, it is clear that it is not necessary to choose 10 as the base of the exponent. Based on the embedded device floating-point types, the best seems to put an equal basis 2 .

Discrete logarithm

Discrete logarithm problem is, according to an a, b, m solve the equation:

$$a^x = b \pmod{m},$$

where a and m - coprime (Note: if they are not relatively prime, then the algorithm described below is incorrect, although presumably it can be modified so that it was still working).

Here we describe an algorithm, known as "**Baby-step-giant-step algorithm**" Proposed **Shanks (Shanks)** in 1971 g, the running time for . Often this simple algorithm $O(\sqrt{m} \log m)$ algorithm called "**Meet-in-the-middle**" (Because it is one of the classic applications of technology "meet-in-the-middle": "separation of tasks in half").

Algorithm

So, we have the equation:

$$a^x = b \pmod{m},$$

where a and m coprime. Transform

equation. Put

$$x = np - q,$$

where n - Is preselected constant (as it is chosen depending on the m We will understand later). Sometimes p called "giant step" (as an increase in its per unit increases x at once n), And in contrast, the q - "Baby step".

Obviously, any x (The interval - it is understood that this range of values will suffice) can be represented in a form wherein it will be sufficient for the values:

$$p \in \left[1; \left\lceil \frac{m}{n} \right\rceil\right], \quad q \in [0; n].$$

Then the equation becomes:

$$a^{np-q} = b \pmod{m},$$

where, using the fact that a and m coprime, we obtain:

$$a^{np} = ba^q \pmod{m}.$$

To solve the original equation, we need to find the appropriate values p and q That the values of the left and right sides are aligned. In other words, it is necessary to solve the equation:

$$f_1(p) = f_2(q).$$

This problem is solved by the meet-in-the-middle as follows. The first phase of the algorithm: we can calculate the function for all values of the argument, and we can sort these values. The second phase algorithm: we will sort out the value of the second variable q , Calculate the second function, and seek the value of the predicted values of the first function using a binary search.

Asymptotics

First, we estimate the computation time of each of the functions. And she and the other comprises the construction of the power that can be performed using [algorithm binary exponentiation](#). Then both of these functions, we can compute in time $O(\log m)$.

The algorithm in the first phase comprises calculating features for each possible value of p and further sorting of values that gives us the asymptotic behavior:

$$O\left(\left\lceil \frac{m}{n} \right\rceil \left(\log m + \log \left\lceil \frac{m}{n} \right\rceil \right)\right) = O\left(\left\lceil \frac{m}{n} \right\rceil \log m\right).$$

In the second phase of the algorithm is evaluated function
 search the array of values that gives us the asymptotic behavior: $f_2(q)$ for each possible value of q and binary

Now, when we combine these two asymptotic we get $\log m$ Multiplied by the sum and m/n And almost clear that the minimum is achieved when $\approx m/n$ le algorithm for optimal constant n should be chosen as follows:

Then the asymptotic behavior of the algorithm takes the form:

$$O(\sqrt{m} \log m).$$

Note. We could exchange roles and (ie the first phase to compute values of the function, and the second - f_1). But it is easy to understand that the result will not change, and the asymptotic behavior of this we can not improve.

Implementation

The simplest implementation

Function `powmod` performs a binary number erection a the power b modulo m See Binary construction the power.

Function `solve` They produce their own solution to the problem. This function returns a response (the number in the interval $[0; m]$). More precisely, one of the answers. Function returns -1 If there are no solutions.

```
int powmod (int a, int b, int m) {
    int res = 1;
    while (b > 0) {
        if (b & 1) {
            res = (res * a) % M;
            - B;
        }
        else {
            a = (a * a) % M;
            b >> = 1;
        }
    }
    return res% m;
}

int solve (int a, int b, int m) {
    int n = (int) sqrt (m + .0) + 1;
    map <int,int> Vals;
    for (int i = n; i >= 1; - I)
        vals[ powmod (a, i * n, m) ] = I;
    for (int i = 0; i <= n; + + I) {
        int cur = (powmod (a, i, m) * B) % M;
        if (vals.count(cur)) {
            int ans = vals[cur] * N - i;
            if (ans < m)
                return ans;
        }
    }
    return -1;
}
```

Here we are for the convenience of the implementation of the first phase of the algorithm used the data structure "Map" (red-black tree) that for each value of the function $f_1(i)$ argument holds i In which this value is achieved. Here, if the same value is achieved repeatedly recorded smallest of all the arguments. This is done in order to subsequently, on the second phase of the algorithm is found in the response interval $[0; m]$.

Given that the argument of the first phase we iterate from one and up to n And the argument of the second phase $f_2()$ moves from zero to n , Then eventually we cover the whole set of possible answers, because

segment $[0; n^2]$ contains a gap. At the same time, a negative response could happen, and responses greater than or equal m we can not ignore - should still be in the corresponding period of the answers $[0; m)$.

This function can be changed in the event if you want to find **all solutions** the discrete logarithm problem. To do this, replace "map" on any other data structure that allows for a single argument to store multiple values (for example, "multimap"), and to amend the code of the second phase.

An improved

At **speed optimization** can proceed as follows.

First, immediately struck by the uselessness of the binary exponentiation algorithm in the second phase.

Instead, you can simply make a variable and multiplies it every time a .

Secondly, the same way you can get rid of the binary exponentiation, and the first phase: in fact, once is enough to calculate the value of a^n . And then just multiplies it.

Thus, the logarithm in the asymptotic behavior will remain, but it will only logarithm associated with the data structure (ie, in terms of the algorithm, sorting and binary search values) - ie it will be the logarithm of the \sqrt{m} , which in practice provides a notable acceleration.

```
int solve (int a, int b, int m) {
    int n = (int) sqrt (m + .0) + 1;

    int an = 1;
    for (int i = 0; i <n; + +
        I) an = (an * a) % M;

    map <int,int> Vals;
    for (int i = 1, Cur = an; i <= n; +
        + I) { if (!
        Vals.count (cur))
        vals[cur] = I;
        cur = (cur * an) % M;
    }

    for (int i = 0, Cur = b; i <= n; +
        + I) { if
        (vals.count (cur)) {
            int ans = vals[cur] * N - i;
            if (ans <m)
                return ans;
        }
        cur = (cur * a) % M;
    }
    return -1;
}
```

Finally, if the module m small enough, then we can do to get rid of the logarithm in the asymptotic behavior - instead of just having got $map <>$ normal array.

You can also recall the hash table: on average, they work well for that, in general gives asymptotics $O(\sqrt{m})$.

Linear Diophantine equations in two variables

Diophantine equation with two unknowns has the form:

$$a \cdot x + b \cdot y = c,$$

where a, b, c - Given integers, x and y - Unknown integers.

Below are several classical problems of these equations: finding any solutions, getting all the decisions finding the number of solutions and the solutions themselves in a certain period, to find a solution with the least amount of unknowns.

Degenerate case

A degenerate case we immediately excluded from consideration when $a = b = 0$. In this case, of course, either the equation has infinite number of arbitrary decision, or it has no solution at all (depending on $c = 0$ or not).

Finding the solutions

Find one of the solutions of the Diophantine equation with two unknowns, you can use [Extended Euclid's algorithm](#).

Assume first that number a and b nonnegative.

Advanced Euclidean algorithm to specify a non-negative numbers a/b and finds the greatest common divisor

$$g = \frac{a}{b}$$

n

As well as such factors g .

d That:

States that when divided into , The Diophantine equation has a solution; otherwise Diophantine equation has no solutions. This follows from the obvious fact that a linear combination of two numbers still must be divisible by a common divisor.

Assume that c divided into g . Then obviously holds:

ie one of the solutions of the Diophantine equation are the numbers:

$$\begin{cases} x_0 = x_g \cdot (c/g), \\ y_0 = y_g \cdot (c/g). \end{cases}$$

b

We have described the decision in the case where the number a and non-negative. If one of them or both are negative, then we can proceed as follows: take their modulus and apply them to the Euclidean algorithm, as described above, and then change the sign found x_0 and y_0 in accordance with the number sign a and b respectively.

Implementation (recall here, we believe that the input data $a = b = 0$ allowed):

```
int gcd (int a, int b, int & X, int & Y) {
    if (a == 0) {
        x = 0; y =
        1; return b;
    }
    int x1, y1;
    int d = gcd (b% a, a, x1,
    y1); x = y1 - (b / a) * x1;
    y = x1;
    return d;
}

bool find_any_solution (int a, int b, int c, int & X0, int & Y0, int & G) {
    g = gcd (abs(a), abs(b), X0, Y0);
```

```
if (c%g! = 0)
```

```

        return false;
x0 *= c / g;
y0 *= c / g;
if (a < 0)  x0 *= -1;
if (b < 0)  y0 *= -1;
return true;
}

```

Getting all the solutions

We show how to get all the other solutions (and there are infinitely many) Diophantine equation, knowing one of the solutions (x_0, y_0) .

So, let $g = \gcd(a, b)$ And number x_0, y_0 satisfy the condition:

$$a \cdot x_0 + b \cdot y_0 = c.$$

Then we note that, by adding to x_0 number b/g while depriving a/g from y_0 We do not violate the equality:

$$a \cdot (x_0 + b/g) + b \cdot (y_0 - a/g) = a \cdot x_0 + b \cdot y_0 + a \cdot b/g - b \cdot a/g = c.$$

Obviously, this process can be repeated any number, ie all numbers of the form:

$$\begin{cases} x = x_0 + k \cdot b/g, \\ y = y_0 - k \cdot a/g, \end{cases} \quad k \in \mathbb{Z}$$

are solutions of the Diophantine equation.

Moreover, only the number and type of such a solution is that we describe the set of all solutions of the Diophantine equation (it turned out to be infinite if not imposed additional conditions).

Finding the number of solutions and the solutions themselves in a given interval

Suppose two segments and And want to find the number of solutions

Diophantine equations underlying the data segments, respectively.

Note that if one of the numbers $[min_x; max_x]$ is zero, then the problem is no more than one solution, so these cases we (in this section) exclude from consideration.

First, we find the best solution with the minimum, ie . To do this, we first find any solution of the Diophantine equation (see paragraph 1). Then get out of it with the smallest solution - For this we use the procedure described in the preceding paragraph, and will reduce / increase Until it would not . And wherein the minimum. This can be done , Arguing with what factor to apply a transform to obtain a minimum number greater than or equal to

throu

Denote found $\geq min_x$ gh .

$O(1)$

ie min_x .

Similarly, you can find the best solution with the maximum

$x = rx1$ $x \leq max_x$
And a solution of
 $[min_y; max_y]$

maximum $y \leq max_y$. Denote Coefficients of these solutions through $rx2$ and $rx1$ respectively.

Cross sections $[lx1; rx1]$ $[lx2; rx2]$; denote the resulting segment by $[lx; rx]$. Allegedly that any solution which x Coefficient lies $[lx; rx]$ - Any such decision is appropriate. (This is true in virtue of the construction of this segment: we first met separately and restrictions on x y . Receiving two segments, and then crossed them by getting an area in which both conditions are met.)

Thus, the number of solutions will be equal to the length of this interval, divided by the coefficient can be changed only by $\pm b$, Plus one.

Show implementation (it is difficult to obtain, since it requires carefully consider the cases of positive and negative coefficients a and b)

```

void shift_solution (int & X, int & Y, int a, int b, int cnt) {
    x += cnt * b;
}

```

```
y -= cnt * a;  
}
```

```

int find_all_solutions (int a, int b, int c, int minx, int maxx, int miny,
int maxy) {
    int x, y, g;
    if (!find_any_solution (a, b, c, x, y, g))
        return 0;
    a /= g; b /= g;

    int sign_a = a>0 ? +1 : -1;
    int sign_b = b>0 ? +1 : -1;
    shift_solution (x, y, a, b, (minx - x) /
B);if (x <minx)
        shift_solution (x, y, a, b,
sign_b);if (x> maxx)
        return
0; int lx1 = x;
shift_solution (x, y, a, b, (maxx - x) /
B);if (x> maxx)
        shift_solution (x, y, a, b,-
sign_b);int rx1 = x;
shift_solution (x, y, a, b, - (miny - y) /
A);if (y <miny)
        shift_solution (x, y, a, b,-
sign_a);if (y> maxy)
        return
0; int lx2 = x;
shift_solution (x, y, a, b, - (maxy - y) /
A);if (y> maxy)
        shift_solution (x, y, a, b,
sign_a);int rx2 = x;
if (lx2> rx2)
    swap      (lx2,
rx2);int lx = max (lx1,
lx2); int rx = min (rx1,
rx2);

return (rx - lx) / abs(b) + 1;
}

```

It is also easy to add to this realization the withdrawal of all the solutions found: it is enough to enumerate x in segment $[x; rx]$ increments, finding for each of them corresponding y directly from equation $ax + by = c$.

Finding solutions in a given interval with the least amount of $x + y$

Here at x and y should also be imposed any restrictions, otherwise the answer will almost always be negative infinity.

The idea solution is the same as in the previous paragraph: first find any solution of Diophantine equations, and then to apply this procedure in the preceding paragraph, we arrive at the best solution.

Indeed, we have the right to do the following transformation (see previous item):

$$\begin{cases} x' = x + k \cdot (b/g), \\ y' = y - k \cdot (a/g), \end{cases} \quad k \in \mathbb{Z}.$$

Note that the sum of $x + y$ changes as follows:

If $a < b$ it is necessary to select as a smaller value if k . If $a > b$, Then you need to select as greater importance.

If $a = b$ We have no way to improve decision - all solutions will have the same amount.

Problem in online judges

List of tasks that can be taken on the subject of Diophantine equations with two

unknowns:

- [SGU # 106 "The Equation"](#) [Difficulty: average]

Modular linear equation of the first order

Statement of the Problem

This equation of the form:

$$a \cdot x = b \pmod{n},$$

where a, b, n - Given integers, x - Unknown integer.

Required to find the desired value at the interval $[0; n - 1]$ (As on the real line, clear there may be infinitely many solutions that are different to each other on $n \cdot k$ Where - any integer). If the solution is not unique, then we'll look at how to get all solutions.

The decision by finding the inverse element

Consider first the simplest case - when a and n coprime. Then we can find back element among a , And multiplying on both sides of it, to obtain a solution (and it will only)

$$x = b \cdot a^{-1} \pmod{n}$$

Now consider the case when a and n are not coprime. Then, clear solution will exist not always (e.g., $2 \cdot x = 1 \pmod{4}$).

Let $g = \gcd(a, n)$ ie their greatest common divisor (Which in this case is more than one).

Then, if b not divisible by g then no solution exists. In fact, for any x left-hand side of the equation, ie is $(a \cdot x) \pmod{n}$ Always divisible by g , while the right side of it is not divisible, which implies that there are no solutions.

If divisible by g , then, dividing both sides of this (ie, dividing, a by g , b by g , n by g) we arrive at new equation:

$$a' \cdot x' = b' \pmod{n'}$$

wherein a' and b' already be relatively prime, and this equation we have learned to solve. We denote its solution through x' .

Clearly, this is x' will also be a solution of the original equation. However, if $g > 1$ Then it is

not only solution. It can be shown that the original equation will have exactly g decisions, and they will

look like $(x' + i \cdot n') \pmod{n}$,
 $i = 0 \dots (g - 1)$.

Summarizing, we can say that the number of decisions modular linear equation is either $g = \gcd(a, n)$ Or zero.

Solution using the Extended Euclidean algorithm

We present our modular equation to Diophantine equation as follows:

where x and k - Unknown integers.

The method of solving this equation is described in the relevant article [Linear Diophantine equations of the second order](#) And it is in the application [The extended Euclidean algorithm](#).

There also described a method for obtaining all solutions of this equation one solution found, and, by the way, this way on closer examination is equivalent to the method described in the preceding paragraph.

The Chinese remainder theorem

Formulation

In its modern formulation of the theorem is as follows:

Let $p = p_1 \cdot p_2 \cdot \dots \cdot p_k$ Wherein p_i - Pairwise coprime.

We assign the arbitrary number a ($0 \leq a < p$) cortege (a_1, \dots, a_k) Wherein $a_i \equiv a \pmod{p_i}$.

$$a \iff (a_1, \dots, a_k).$$

Then the correspondence (between numbers and tuples) will be **bijective**. And, moreover, the operations performed on the number a You can perform the equivalent of the corresponding elements of tuples - by independent operations on each component.

That is, if

$$\begin{aligned} a &\iff (a_1, \dots, a_k), \\ b &\iff (b_1, \dots, b_k), \end{aligned}$$

then we have:

$$\begin{aligned} (a + b) \pmod{p} &\iff ((a_1 + b_1) \pmod{p_1}, \dots, (a_k + b_k) \pmod{p_k}), \\ (a - b) \pmod{p} &\iff ((a_1 - b_1) \pmod{p_1}, \dots, (a_k - b_k) \pmod{p_k}), \\ (a \cdot b) \pmod{p} &\iff ((a_1 \cdot b_1) \pmod{p_1}, \dots, (a_k \cdot b_k) \pmod{p_k}). \end{aligned}$$

In its original formulation of this theorem was proved by the Chinese mathematician Sun Tzu around 100 BC
Namely, he has shown in the special case of modular equivalence of solutions of equations and solving a modular equation (see Corollary 2 below).

Corollary 1

Modular system of equations:

$$\begin{cases} x \equiv a_1 \pmod{p_1}, \\ \dots, \\ x \equiv a_k \pmod{p_k} \end{cases}$$

has a unique solution modulo p .

(As above, $p = p_1 \cdot \dots \cdot p_k$, The number p_i pairwise relatively prime, and set a_1, \dots, a_k - Arbitrary set of integers)

Corollary 2

Consequence is the connection between the system of modular equations and one corresponding modular equation: Equation:

$$x \equiv a \pmod{p}$$

equivalent to the system of equations:

$$\begin{cases} x \equiv a \pmod{p_1}, \\ \dots, \\ x \equiv a \pmod{p_k} \end{cases}$$

(As above, it is assumed that $p = p_1 \cdot \dots \cdot p_k$, The number p_i pairwise relatively prime, and a - An arbitrary integer)

Garner's algorithm

Of the Chinese remainder theorem that can replace operations on the number of operations on tuples. Recall, each number a is associated with a tuple (a_1, \dots, a_k) Wherein:

$$a_i \equiv a \pmod{p_i}.$$

It can be widely used in practice (in addition to direct application to restore the number he balances of the various modules), as we thus can replace surgery in the long

arithmetic operations with an array of "short" numbers. Say, an array of 1000 elements "enough" to number about 3000 characters (if selected as the first 1000 simple); and if as filter's simple about a billion, then by enough already with approximately 9000 signs. But, of course, then need to learn **restore** by this tuple. Corollary 1 shows that this recovery is possible and, moreover, the only (provided $0 \leq a < p_1 \cdot p_2 \cdot \dots \cdot p_k$).

Garner's algorithm and an algorithm, allowing it to perform recovery and effectively.

We seek a solution in the form:

$$a = x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 + \dots + x_k \cdot p_1 \cdot \dots \cdot p_{k-1},$$

i.e. mixed radix digits with weights p_1, p_2, \dots, p_k .

We denote r_{ij} ($i = 1 \dots k - 1$, $j = i + 1 \dots k$) Number, which is the inverse p_i modulo p_j (Finding the inverses of elements in the ring of modulo described [here](#)):

We substitute a mixed radix in the first equation, we get:

$$a_1 \equiv x_1.$$

We now substitute in the second equation:

$$a_2 \equiv x_1 + x_2 \cdot p_1 \pmod{p_2}.$$

Transform this expression, subtracting from both sides x_1 and dividing by n_1 :

$$\begin{aligned} a_2 - x_1 &\equiv x_2 \cdot p_1 \pmod{p_2}; \\ (a_2 - x_1) \cdot r_{12} &\equiv x_2 \pmod{p_2}; \\ x_2 &\equiv (a_2 - x_1) \cdot r_{12} \pmod{p_2}. \end{aligned}$$

Substituting into the third equation, the same way we obtain:

$$\begin{aligned} a_3 &\equiv x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 \pmod{p_3}; \\ (a_3 - x_1) \cdot r_{13} &\equiv x_2 + x_3 \cdot p_2 \pmod{p_3}; \\ ((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} &\equiv x_3 \pmod{p_3}; \\ x_3 &\equiv ((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \pmod{p_3}. \end{aligned}$$

Already quite clearly visible pattern that is easier to express the code:

```
for (int i = 0; i < k; ++i) {
    x[i] = A[i];
    for (int j = 0; j < i; ++j) {
        x[i] = R[j][i] * (x[i] - x[j]);
        x[i] = X[i] % P[i];
        if (x[i] < 0) x[i] += P[i];
    }
}
```

So we learned to calculate the coefficients x_i during $O(k^2)$ itself is the answer - the number - can be restored formula:

It should be noted that in practice almost always need to calculate the answer using **Long arithmetic**, But coefficients themselves x_i still calculated on the built-in types, and therefore the whole Garner algorithm is very effective.

Implementation of the algorithm Garner

Preferred to implement this algorithm in Java, because it contains a long arithmetic standard, but because there are no problems with the transfer of the number of the modular system in the usual number of (Using a standard class BigInteger).

The following implementation of the algorithm Garner supports addition, subtraction and multiplication, and supports the work with negative numbers (see notes about this after the code). Implemented a number of customary translation desyatichkogo submission to a modular system and vice versa.

In this example taken 100 after simple 10^9 . That allows you to work with numbers up to about 10^{900} .

```
final int SZ = 100;
int pr[] = new int[SZ];
int r[][] = new int[SZ][SZ];

void init() {
    for (int x=1000*1000*1000, i=0; i<SZ; + +x)
        if (BigInteger.valueOf(x).isProbablePrime(100)) pr[i+
                +] = x;

    for (int i=0; i<SZ; + +i)
        for (int j=i+1; j<SZ; + +j)
            r[i][j] = BigInteger.valueOf(pr[i]).modInverse( BigInteger.valueOf(pr
[j])).intValue();
}

class Number {

    int a[] = new int[SZ];

    public Number() {
    }

    public Number (int n) {
        for (int i=0; i<SZ; + +i)
            a[i] = n % pr[i];
    }

    public Number (BigInteger n) {
        for (int i=0; i<SZ; + +i)
            a[i] = n.mod( BigInteger.valueOf(pr
[i])).intValue();
    }

    public Number add (Number n) {
        Number result = new Number();
        for (int i=0; i<SZ; + +i)
            result.a[i] = (a[i] + n.a[i]) %
pr[i]; return result;
    }

    public Number subtract (Number n) {
        Number result = new Number();
        for (int i=0; i<SZ; + +i)
            result.a[i] = (a[i] - n.a[i] + pr[i]) %
pr[i]; return result;
    }

    public Number multiply (Number n) {
        Number result = new Number();
        for (int i=0; i<SZ; + +i)
            result.a[i] = (int) ((a[i] * 11 * n.a[i]) %
pr[i]);
        return result;
    }
}
```

```

public BigInteger bigIntegerValue (boolean can_be_negative) {
    BigInteger result = BigInteger.ZERO,
        mult = BigInteger.ONE;
    int x[] = new int[SZ]; for
    (int i=0; i<SZ; + +i) {
        x[i] = a[i];
        for (int j=0; j<i; + +j) {
            long cur = (x[i] - x[j]) * 11 * r[j] [i];
            x[i] = (int) ((cur % pr[i] + pr[i]) %
pr[i]);
        }
        result = result.add( mult.multiply
(BigInteger.valueOf(x[i])));
        mult = mult.multiply( BigInteger.valueOf
(pr[i]));
    }

    if (can_be_negative)
        if (result.compareTo( mult.shiftRight(1)) > =
0) result = result.subtract( mult );
    return result;
}
}

```

Support for **negative** Number deserves mention (flag **can_be_negative**)
function **bigIntegerValue()**). Modular scheme itself does not imply differences between positive
& negative numbers. However, it can be seen that if the answer to a specific problem modulo does not exceed half of the product of
all primes, the positive numbers will be different from the negative that positive numbers are obtained than this mid and negative -
more. So after we Garner classical algorithm compares the result to the middle, and if it is, then we deduce minus
& invert the result (ie, subtract it from the product of all prime, and print it already).

Finding factorial power divider

Given two numbers n and k . Required to calculate what degree divider includes a number, ie find the greatest x such that $n!$ divided into k^x .

Solution for the case of simple k

Consider first the case when k simple. Let us write the

expression for the factorial explicitly:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

Note that each n th term of this work is divided into

le gives one to answer; the number of such members

equally $\frac{1}{k}$.

$$k$$

Furthermore, we note that every n th term of this series is divided into $\lfloor \frac{n}{k} \rfloor$ such members

le gives another one to answer (given that in first degree has already been considered before); the number of such members $\lfloor \frac{n}{k^2} \rfloor$.

And so on, each $\lfloor \frac{n}{k^i} \rfloor$. The first term of the series gives one to answer, and the number of such members is equal $\lfloor \frac{n}{k^i} \rfloor$. Thus, the magnitude of response is:

This amount, of course, is not infinite, because Only the first approximately

$\log_k n$ members are different from zero.

Consequently, the asymptotic behavior of the algorithm $O(\log_k n)$.

Implementation:

```
int fact_pow (int n, int k) {
    int res = 0;
    while (n) {
        n /= k;
        res += 
            n;
    }
    return res;
}
```

Solution for the case of composite k

The same idea is applied directly anymore.

But we can factorize k . Solve the problem for each of its prime factor, and then to select a minimum of answers. More formally, let - this is the first factor of the number belongs to a degree. To solve the problem with p_i using the above formula for $O(\log n)$ though we got an answer Ans_i . Then answer for composite k will be a minimum of units Ans_i / p_i .

Given that the factorization is performed in a very simple manner $O(\sqrt{n})$. Obtain final asymptotics $O(\sqrt{n})$.

Ternary balanced system of value

Ternary balanced system of value - it is non-standard positional numeral system. The base of power 3 . But it differs from the usual ternary system that figures are

$-1, 0, 1$. Since the use -1 to single digits very uncomfortable, it usually takes a special designation.

We agree to denote here minus one letter z .

For example, by 5 balanced ternary system is written as 1zzz And by -5 - As $\text{z}11$. Ternary balanced system value allows you to record negative numbers without writing a single

"minus" sign. Balanced ternary system allows fractional numbers (eg, $1/3$ written as $0.\text{1}$).

Translation algorithm

Learn to translate numbers in balanced ternary system. For this we must first

convert the number in the ternary system.

It is clear that now we have to get rid of digits 2 For which we note that $2 = 3 - 1$ ie we can replace the two in the current discharge on -1 While increasing the next (ie the left of it in the natural record) discharge on 1 . If we move from right to left on the record and perform the above operation (in this case in some discharges can overflow more 3) In this case, naturally, "reset" extra triples in MSB), we arrive at a balanced ternary recording. As is easily seen, the same rule holds true for fractional numbers.

More gracefully above procedure can be described as follows. We take the number of ternary, adds to it an infinite number of $\dots 11111.11111\dots$ And then from the result subtract each digit one (already without any hyphens).

Knowing now the translation algorithm of conventional balanced ternary system, we can easily implement the operations of addition, subtraction and division - simply reducing them to the corresponding operations on ternary unbalanced numbers.

Factorial calculation modulo

In some cases it is necessary to consider on some simple module p complex formulas, which may contain, including factorials. Here we consider the case when the module p relatively small. It is clear that this problem only makes sense if the factorial and included in the numerator and the denominator of the fractions. Indeed, the factorial and all subsequent vanish modulo p . But in fractions $p!$, all factors containing p can be reduced, and the resulting expression has to be different from zero modulo p .

Thus, formally task such. Required to calculate the modulo a prime P , without taking into account all the multiple p factors included in the factorial. By learning to effectively compute a factorial, we can quickly calculate the value of various combinatorial formulas (eg [Binomial coefficients](#)).

Algorithm

Let us write this "modified" factorial explicitly:

$$\begin{aligned} n! \% p &= 1 \cdot 2 \cdot 3 \cdots \underbrace{(p-2) \cdot (p-1)}_{p} \cdot \underbrace{(p+1) \cdot (p+2) \cdots \cdot (2p-1)}_{2p} \cdot \underbrace{(2p+1) \cdots}_{p^2} \\ &\quad \cdot (p^2 - 1) \cdot \underbrace{1 \cdots (p^2 + 1)}_{p^2} \cdots n = \\ &= 1 \cdot 2 \cdot 3 \cdots \underbrace{(p-2) \cdot (p-1)}_{p} \cdots \underbrace{1 \cdot 2 \cdots}_{p} \underbrace{(p-1) \cdot 2}_{2p} \cdots \underbrace{(p-1) \cdot 1}_{p^2} \cdots \\ &\quad \cdot 1 \cdot 2 \cdots (n \% p) \pmod{p}. \end{aligned}$$

With this record shows that the "modified" factorial divided into several blocks of length p (The last block may be shorter), which are all identical, except for the last element:

Common part calculate blocks easily - it's just $(p-1)! \pmod{p}$ which can be calculated or programmatically theorem Wilson (Wilson) immediately $(p-1)! \pmod{p} = p-1$. To multiply these common parts All units, need to elevate the obtained value exponentiation p what can be done for $O(\log n)$ operations (Cf. [Binary exponentiation](#); however, you will notice that we actually erect minus one in which- the degree and therefore the result will always be either or $\frac{1}{p-1}$, Depending on the parity index. Value in the latter, incomplete block also can be calculated separately for $O(p)$. Only the last elements blocks, we consider them carefully:

And again we come to the "modified" factorial, but smaller dimension (as much as was full blocks and have their $\frac{n!}{(p-1)!}$). Thus, the calculation of the "modified" factorial $n! \% p$ we have reduced operations to the calculation for $O(p)$ already $O(\log_p n)$. Expanding this recurrence relation, we obtain that recursion depth is $O(\log_p n)$. Total **asymptotics** algorithm is obtained $O(p \log_p n)$.

Implementation

It is clear that the implementation is not necessary to use recursion explicitly: as tail recursion, it is easy to deploy in the cycle.

```
int factmod (int n, int p) {
    int res = 1;
    while (n > 1) {
        res = (res * ((n / p) % 2 ? p-1 : 1)) %
            p; for (int i = 2; i <= n % p; ++i)
```

```
        res = (res * i) % P;
        n /= p;
    }
    return res% p;
}
```

This implementation works for $O(p \log_p n)$.

Through all this mask subpatterns

Enumerating subpatterns fixed mask

Dana bitmask m . Requires effectively enumerate all its subpatterns, ie such masks s , Which may be included only those bits that are included in the mask m .

Immediately look at the implementation of this algorithm, based on tricks with Boolean operations:

```
int s = m;
while (s > 0) {
    You can use ... s ...
    s = (s-1) & M;
}
```

or by using a more compact operator *for*:

```
for (int s = m; s; s = (s-1) & M)
    You can use ... s ...
```

The only exception for both versions of the code - the subpattern is zero, will not be processed. Processing it will take out of the loop, or use a less elegant design, for example:

```
for (int s = m; ; s = (s-1) & M) {
    You can use ... s ...
    if (s == 0) break;
}
```

Let us examine why the above code really finds all of this mask subpattern, without repetitions in O (number), and in descending order.

Let us have a current capturing subpattern s And we want to move to the next subpattern. Subtract from the mask s unit, thus we remove the rightmost single bit, and all the bits to the right of him to put in 1 . Next, remove all the "extra" bits set, which are not included in the mask m and therefore can not be included in the subpattern. Removal operation is performed bit. As a result, we "cut off the" mask before the largest value that it can take, i.e. until after the next subpattern s in descending order.

Thus, the algorithm generates all subpatterns this mask in order strictly decreasing, spending on each transition on two basic operations.

Especially consider the instant

when $s = 0$. After completing $s - 1$ we obtain a mask in which all bits included (bit representation of the number -1), And after removing the extra bits operation $(-1) \& m$ will nothing but a mask m . So with a mask $s = 0$ should be careful - if time does not stop at zero mask, the algorithm may enter an infinite loop.

Through all the masks with their subpatterns. Rating 3^n

In many problems, especially in the dynamic programming by masks is required to sort out all the masks, and masks for each - all subpatterns:

```
for (int m = 0; m < (1 << N); ++ M)
    for (int s = m; s; s = (s-1) & M)
        Use ... s and m ...
```

We prove that the inner loop will execute total $O(3^n)$ iterations.

Proof: 1 way. Consider i Th bit. For him, generally speaking, there are exactly three ways: it is not included in the mask m (And therefore in the subpattern s); it enters m But is not included in s ; it enters m and s . Just bits n So unique combinations will 3^n , As required.

Proof: 2 way. Note that if the mask m has k included bits, it will have

2^k subpatterns. Since the length of the masks enabled bits have all combinations will be:

C_n^k (Cf. "Binomial coefficients") Is

Calculate this amount. For this we note that it is nothing like the binomial theorem expansion in the expression $(1 + 2)^n$ ie 3^n , As required.

Primitive roots

Definition

A primitive root modulo n (Primitive root modulo n) is a number g such that all its powers modulo n run through all the numbers relatively prime to n . Mathematically, this is formulated as follows: if g is a primitive root modulo n , then for any integer a such that $\gcd(a, n) = 1$, there is an integer k such that $g^k \equiv a \pmod{n}$.

In particular, in the case of a simple n degree primitive roots run through all the numbers from 1 to $n - 1$.

Existence

A primitive root modulo n if and only if n degree is either an odd prime or double prime power, and also in cases $n = 1, n = 2, n = 4$.

This theorem (which has been fully proved by Gauss in 1801) is given here without proof.

Communication with Euler function

Let g - A primitive root modulo n . Then we can show that the smallest number of k for which $(g^k \equiv 1 \pmod{n})$ (Multiplicative order), power. Moreover, the converse is also true, and this fact will be used later in our algorithm for finding primitive roots.

Furthermore, if the modulo n has at least one primitive root, then all their $\phi(\phi(n))$ (Since the cyclic group with k element has $\phi(k)$ generators).

Algorithm for finding a primitive root

Naive algorithm requires values for each test g time to compute all of its powers and verify that they are all different. It's too slow algorithm, below we are using several well-known theorems of number theory obtain a faster algorithm.

Above, we present a theorem that if the smallest number for which $g^k \equiv 1 \pmod{n}$ (i.e. k - Index of g) Power $\phi(n)$ Then g - primitive root. Since for any number theorem holds Euler ($a^{\phi(n)} \equiv 1 \pmod{n}$), Then to verify that the primitive root, it suffices to verify that for all numbers d smaller than $\phi(n)$ Operating $g^d \not\equiv 1 \pmod{n}$. However, until it is too slow algorithm.

From Lagrange's theorem that the index of any number modulo n is a divisor of $\phi(n)$. So it is sufficient to verify that all proper divisors $d \mid \phi(n)$ performed $g^d \not\equiv 1 \pmod{n}$. This is a much faster algorithm, but we can go further. Factorize number. We prove that in the previous algorithm, it suffices to consider in

As a number of the form $\frac{\phi(n)}{p_i}$. Indeed, let d - Any proper divisor of $\phi(n)$. Then, obviously, there is a j such that $d \mid \frac{\phi(n)}{p_j}$ ie $d \cdot k = \frac{\phi(n)}{p_j}$. However, if $g^d \equiv 1 \pmod{n}$ Then we would get:

ie still among the numbers of the form $\frac{\phi(n)}{p_i}$ there would be something for which the condition is not met, as required.

Thus, an algorithm for finding a primitive root. Find $\phi(n)$. Factorize it. Now iterate through all the numbers $g = 1 \dots n$ And for each consider all quantities $g^{\frac{\phi(n)}{p_i}} \pmod{n}$. If the current g All these numbers were different from 1, It g is the desired primitive root.

The running time (assuming that the number $\phi(n)$ there is $O(\log \phi(n))$ divisors, and erection

degree algorithm is executed [Binary exponentiation](#) for equal $\mathcal{O}(\log \phi(n) \cdot \log n)$ plus the time factorization number $\phi(n)$ in Ans - The result, ie value of the unknown primitive root.

About the rate of growth with the growth of primitive roots n known only estimates. Known that primitive roots - relatively small quantities. One of the famous guest - evaluation Shupe (Shoup), that, assuming the truth of the Riemann hypothesis, there is a primitive root $\mathcal{O}(\log^6 n)$.

Implementation

Function powmod () performs a binary exponentiation modulo a function generator (int p) - is a primitive root modulo a prime (factoring of p) implemented here simple algorithm for $\mathcal{O}(\sqrt{\phi(n)})$. To adapt this function to arbitrary enough p add calculation Euler function variable phi.

```

int powmod (int a, int b, int p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = int (res * 111 * a% p), - B;
        else
            a = int (a * 111 * a% p), B >> = 1;
    return res;
}

int generator (int p) {
    vector <int>
    Fact;
    int phi = p-1, N = phi;
    for (int i =2; i * i <= n;
    + + I)
        if (n% i == 0) {
            fact.push_back
            (i);while (n% i ==
            0)
                n / = i;
        }
    if (n> 1)
        fact.push_back (n);

    for (int res =2; res <= p; + +
    Res) { bool ok = true;
        for (size_t i =0; i <fact.size() && Ok; + + I)
            ok & = powmod (res, phi / fact[i], P) !=
            1;if (ok) return res;
    }
    return -1;
}

```

Discrete root extract

Problem of discrete root extraction (by analogy with [the discrete logarithm problem](#)) Reads as follows. According n (n - Simple) a, k required to find all x satisfying the condition:

$$x^k \equiv a \pmod{n}$$

An algorithm for solving

Will solve the problem by reducing it to the discrete logarithm problem.

To do this, apply the concept of [Primitive root modulo \$n\$](#) . Let g - A primitive root modulo n (Because n - Simple, it exists). We can find it, as described in the related article, for

$$O(\text{Ans} \cdot \log \phi(n) \cdot \log n) = O(\text{Ans} \cdot \log^2 n)$$
 plus the time factorization number $\phi(n)$.

Immediately discard the case when $a = 0$ - In this case we immediately find the answer $x = 0$.

Since in this case (n - Prime) any number of 1 to $n - 1$ represented in the form of a power of a primitive root, the root of the discrete problem, we can be written as

$$(g^y)^k \equiv a \pmod{n}$$

where

$$x \equiv g^y \pmod{n}$$

Trivial transformation we obtain:

$$(g^k)^y \equiv a \pmod{n}$$

Here is the unknown quantity y So we came to the discrete logarithm problem in pure form. This problem can be solved [algorithm baby-step-giant-step Shanks](#) for, ie, find one of $O(\sqrt{n} \log n)$ solutions y_0 this equation (or find that this equation has no solution).

Suppose we have found a solution y_0 this equation, then one of the solutions of the discrete root will $x_0 = g^{y_0} \pmod{n}$.

Finding all solutions, knowing one of them

To completely solve the problem, we must learn one found

$$x_0 = g^{y_0} \pmod{n}$$
 find all other solutions.

For this, we recall a fact that a primitive root always has order

[Primitive Root](#) ie the least degree of , giving unit, is

exponent term with $\phi(n)$ does not change anything:

$\phi(n)$ (Cf. article about $\phi(n)$). Therefore, in addition

Hence, all the solutions have the form:

$$x = g^{y_0 + \frac{l \cdot \phi(n)}{k}} \pmod{n} \quad \forall l \in \mathbb{Z}$$

where l chosen so that the fraction was intact. To this fraction was intact, the numerator must be a multiple of the least common multiple $\phi(n)^k$ and k Where (recalling that the least common multiple of two numbers $\text{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$), We obtain:

$$x = g^{y_0 + i \frac{\phi(n)}{\gcd(k, \phi(n))}} \pmod{n} \quad \forall i \in \mathbb{Z}$$

This is the final convenient formula, which gives a general view of all the solutions of the discrete root.

Implementation

We present a complete implementation, including finding a primitive root, discrete logarithm and finding and conclusion of all solutions.

```
int gcd (int a, int b) {
    return a? gcd (b% a, a) : B;
}

int powmod (int a, int b, int p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = int (res * 111 * a% p), - B;
        else
            a = int (a * 111 * a% p), B >> = 1;
    return res;
}

int generator (int p) {
    vector <int> Fact;
    int phi = p-1, N = phi;
    for (int i =2; i * i <= n;
    + + I)
        if (n% i == 0) {
            fact.push_back
            (i);while (n% i ==
            0)
                n / = i;
        }
    if (n> 1)
        fact.push_back (n);

    for (int res =2; res <= p; + +
        Res) { bool ok = true;
        for (size_t i =0; i <fact.size() && Ok; + + I)
            ok & = powmod (res, phi / fact[i], P) != 1;if (ok) return res;
    }
    return -1;
}

int main() {

    int n, k, a;
    cin >> N >> k >> a;
    if (a == 0) {
        puts ("1 \
        n0"); return
        0;
    }

    int g = generator (n);

    int sq = (int) sqrt (n + .0) + 1;
    vector <pair <int,int>> Dec
    (sq);for (int i =1; i <= sq; + +
    I)
        dec[i-1] = Make_pair (powmod (g, int (i * sq * 111 * k% (n
        -1)), N), I);
    sort (dec.begin(),
    Dec.end());int any_ans = -1;
    for (int i =0; i <sq; + + I) {
        int my = int (powmod (g, int (i * 111 * k% (n - 1)), N) *
        111 * a% n);
        vector <pair <int,int>> :: Iterator it =
            lower_bound (dec.begin(), Dec.end(), Make_pair (my,
            0));if (it! = dec.end() && It-> first == my) {
            any_ans = it-> second * sq -
            i; break;
    }
}
```

```
}
```

```
if (any_ans == -1) {
```

```
    puts ("0");
```

```
    return 0;
}

int delta = (n-1) / Gcd (k, n-1);
vector <int> Ans;
for (int cur = any_ans% delta; cur <n-1; cur += delta)
    ans.push_back (powmod (g, cur, n));
sort (ans.begin(),
Ans.end());printf ("% D \ n",
Ans.size());
for (size_t i =0; i <ans.size(); +
    + I) printf ("% D",
Ans[i]);
}
```

Sieve of Eratosthenes with linear time work

Given the number n . Required to find **all simple** in the interval $[2; n]$.

The classic way to solve this problem - **Sieve of Eratosthenes**. This algorithm is very simple, but it works for the time $O(n \log \log n)$.

Although there is currently a lot of known algorithms running in sublinear time (ie), the algorithm described below is $o(n)$ interesting for its **simplicity** - It is practically difficult to classical Sieve of Eratosthenes.

Furthermore, the algorithm presented here as a "side effect" is actually computes **factorization of all the numbers** in the interval $[2; n]$. That can be useful in many practical applications.

Disadvantage driven algorithm is that it uses **more memory** than the classical sieve of Eratosthenes: start requires an array of n numbers, while the classical sieve of Eratosthenes only enough n bits of memory (which is obtained 32 times less).

Thus, the described algorithm is only used to sense properties of the order 10^7 . Not more.

Authorship algorithm apparently belongs Grice and Misra (Gries, Misra, 1978 - see references at the end). (And, in fact, call the algorithm "Sieve of Eratosthenes" correctly: too different these two algorithms.)

Description of the algorithm

Our goal - to count for each number i in the interval from $[2; n]$ its **minimal prime divisor** $lp[i]$. In addition, we need to keep a list of all primes found - let's call it an array $pr[]$.

Initially all values are filled with zeros, which means that we are assuming all the numbers simple. In the course of the algorithm, this array will be gradually filled.

We now iterate the current number i from 2 to n . We can have two cases:

- $lp[i] = 0$ - This means that the number of i - Simple, because for it has not found any other divisors. Consequently, it is necessary to assign $lp[i] = i$ and adding i end of the list $pr[]$.
- $lp[i] \neq 0$ - This means that the current number of i - Composite, and it is a minimal prime divisor $lp[i]$.

In both cases, the process then begins **placement values** in the array: we will take the number, **fold** i and update their value. However, our goal - to learn to do it so that in the end each number value $lp[]$ would be set more than once.

It is argued that this can do so. Consider the number of the form:

where the sequence p_j - It's simple, do not exceed $lp[i]$ (Just for this purpose we need to keep a list of all primes).

All properties of this kind places a new value $lp[x_j]$ - Apparent power, it will be p_j .

Why such an algorithm is correct, and why it works in linear time - see below, in the meantime let its implementation.

Implementation

Sieve is performed before said in a constant number N .

```
const int N = 10000000;
int lp[N + 1]; vector<int> Pr;

for (int i = 2; i <= N; ++i)
    if (lp[i] == 0) {
        lp[i] = i;
        pr.push_back (i);
```

```

}
for (int j = 0; j <(int)pr.size() && Pr[j] <= lp[i] && I * pr[j] <=
N; ++j) lp[i * pr[j]] = Pr[j];
}

```

This implementation can accelerate a bit, getting rid of the vector pr (Replacing it with a regular array with counter) as well as getting rid of duplicate multiplying a nested loop for (For which the result of work just have to remember a particular variable).

Proof of correctness

We prove **correctness** algorithm, i.e. it places all the correct values, each of which $lp[]$ will be installed only once. This will imply that the algorithm works in linear time - since all other steps of the algorithm obviously work for $O(n)$.

For this, note that any number of i the only representation like this:

$$i = lp[i] \cdot x,$$

where $lp[i]$ - (As before) the minimum prime divisor i And by x has no divisors less $lp[i]$, ie:

$$lp[i] \leq lp[x].$$

Now compare this with the fact that our algorithm does - it actually for each x enumerates all simple to that it can be multiplied, ie simple to inclusive $lp[x]$ to get the number in the specified above view.

Therefore, the algorithm really pass for each composite number exactly once, putting him right value $lp[]$.

This means the correctness of the algorithm and the fact that it runs in linear time.

Time and memory required

Although the asymptotics better asymptotics $O(n \log \log n)$ classical sieve of Eratosthenes, the difference between them small. In practice this means a two-fold difference in speed, and optimized versions sieve of Eratosthenes and not lose here given algorithm.

Given the cost of memory required for this algorithm - array length n and an array of all prime $lp[]$ length about $n / \ln n$ - This seems to be inferior to the classical algorithm sieve on all counts.

However, it makes that the array, calculated by the algorithm, allows you to search any number factorization in the segment $[2, n]$ the order of the size of this factorization. Moreover, the price of one additional array can be made to factorization in this division operation is not needed.

Knowing the factorization of all the numbers - very useful information for some tasks, and this algorithm is one of the few that allow you to look for it in linear time.

Literature

- David Gries, Jayadev Misra. **A Linear Sieve Algorithm for Finding Prime Numbers** [1978]

BPSW test for primality

Introduction

Algorithm BPSW - this is a test number on simplicity. This algorithm is named for its inventors: Robert Bailey (Ballie), Carl Pomerance (Pomerance), John Selfridge (Selfridge), Samuel Wagstaff (Wagstaff). Algorithm was proposed in 1980. To date, the algorithm has not been found any counterexample, as well as the proof has not been found.

BPSW algorithm has been tested on all numbers up to 1015. Besides, trying to find a counterexample using the PRIMO (see [6]) Based on the simplicity of the test using elliptic curves. Program worked for three years, did not find any counterexample, whereby Martin suggested that there is no no BPSW-pseudosimple smaller 1010000 (pseudosimple number - a composite number for which the algorithm produces a result "simple"). At the same time, Carl Pomerance in 1984 presented a heuristic proof that there are infinitely many BPSW-pseudosimple numbers.

Complexity of the algorithm has BPSW O ($\log_3(N)$) bit operations. If we compare the algorithm BPSW with other tests, such as the Miller-Rabin test, the algorithm BPSW is usually 3-7 times slower.

The algorithm is often used in practice. Apparently, many commercial mathematical packages, wholly or partly rely on an algorithm to check BPSW Primality.

Brief description

The algorithm has several different implementations, differing only in details. In this case, the algorithm has the following form:

1. Run the Miller-Rabin test with base 2.
2. Run a strong Lucas-Selfridge test using Lucas sequence with parameters Selfridge.
3. Redo "prime" only when both tests have returned "prime".

+0. In addition, at the beginning of the algorithm can add a check for trivial divisors, say, 1000. This will increase the speed of operation on a composite number, however, has slowed somewhat simple algorithm.

Thus BPSW algorithm based on the following:

1. (Fact) test and the Miller-Rabin test Lucas-Selfridge and if wrong, it is only one way: some composite numbers are recognized as these algorithms are simple. Conversely, these algorithms do not make mistakes ever.
2. (Assumption) test and the Miller-Rabin test Lucas-Selfridge and if wrong, that are never wrong on one number at a time.

In fact, the second assumption seems to be as wrong - heuristic proof-refutation Pomerance below. Nevertheless, in practice, no one pseudosimple still have not found, so we can assume conditional second assumption is true.

Implementation of algorithms in this article

All the algorithms in this paper will be implemented in C++. All programs were tested only on the compiler Microsoft C++ 8.0 SP1 (2005), should also compile on g++.

Algorithms are implemented using templates (templates), which allows their use as a built-in numeric types, and own class that implements the long arithmetic. [Until long arithmetic is not included in the article - TODO]

In the article itself will be given only the most essential functions, texts as auxiliary functions can be downloaded in the appendix to this article. Here, we present only the headers of these functions along with the comments:

```
/ /! Module 64-bit long long  
number abs (Long long n);
```

```

unsigned long long abs (unsigned long long n);

// ! Returns true, if n is even
template <class T>
bool even (Const T & n);

// ! Divides the number by 2
template <class T>
void bisect (T & n);

// ! Multiplies the number by 2
template <class T>
void redouble (T & n);

// ! Returns true, if n - the exact square of a prime number
template <class T>
bool perfect_square (Const T & n);

// ! Calculates the root of a number, rounding it down
template <class T>
Tsq_root (Const T & n);

// ! Returns the number of bits including
template <class T>
unsigned bits_in_number (T n);

// ! Returns the value of k-bit number (bits are numbered from zero)
template <class T>
bool test_bit (Const T & n, unsigned k);

// ! Multiplies a * = b (mod n)
template <class T>
void mulmod (T & a, T b, const T & n);

// ! Computes a ^ k (mod n)
template <class T, class T2>
Tpowmod (T a, T2 k, const T & n);

// ! Converts the number n in the form q * 2 ^ p
template <class T>
void transform_num (T n, T & p, T & q);

// ! Euclid's algorithm
template <class T, class T2>
Tgcd (Const T & a, const T2 & b);

// ! Calculates jacobi (a, b) - Jacobi symbol
template <class T>
T jacobi (T a, T b);

// ! Calculates pi (b) of the first prime numbers. Returns a vector with
// simple and pi - pi (b)
template <class T, class T2>
const std :: vector & get_primes (Const T & b, T2 & pi);

// ! N trivial check on simplicity, all the divisors are moving to m. /
// ! Results: 1 - if exactly n prime, p - found its divisor 0 - if unknown
template <class T, class T2>
T2 prime_div_trivial (Const T & n, T2 m);

```

Miller-Rabin test

I will not focus on the Miller-Rabin test, as it is described in many sources, including in Russian (eg see [5]).

My only comment is that the speed of his work has $O(\log_3(N))$ bit operations and bring the finished implementation of this algorithm:

```

template <class T, class T2>
bool miller_rabin (T n, T2 b)
{
    // first check for trivial cases if (n
    == 2)
        return true;
    if (n < 2 || even (n))
        return false;

    // check that n and b are relatively prime (otherwise it will fail)
    // if they are not relatively prime, then either n is not easy,
    // either
need to increase
the b if
(b <2)
    b = 2;
for (T g; (g = gcd (n, b)) != 1; ++ b)
    if (n > g)
        return false;

    // expand the n-1 = q *
2 ^ p T n_1 = n;
- N_1;
T p, q;
transform_num (n_1, p, q);

    // compute b ^ q mod n, if it is equal to 1 or n-1, n is prime
(or pseudosimple)
    T rem = powmod (T (b), q,
n); if (rem == 1 || rem ==
n_1) return true;

    // Now compute b ^ 2q, b ^ 4q, ..., b ^ ((n-1) / 2)
    // If any of them is equal to n-1, n is prime (or pseudosimple) for
(T i = 1; i < p; i++)
{
    mulmod (rem, rem, n);
    if (rem == n_1)
        return true;
}

return false;
}

```

Strong test Lucas-Selfridge

Strong Lucas-Selfridge test consists of two parts: Selfridge algorithm for calculating a parameter, and a strong algorithm Lucas performed with this parameter.

Algorithm Selfridge

Among the sequences 5, -7, 9, -11, 13, ... by a first D, for which $J(D, N) = -1$ and $\gcd(D, N) = 1$, where $J(x, y)$ - Jacobi symbol.

Parameters Selfridge will $P = 1$ and $Q = (1 - D) / 4$.

It should be noted that the parameter does not exist for Selfridge numbers that are accurate squares. Indeed, if the number is a perfect square, then bust D comes to \sqrt{N} , where it appears that $\gcd(D, N) > 1$, ie found that the number N is composite.

In addition, the parameters will be calculated incorrectly Selfridge for even numbers and units; however, verification of these cases is not difficult.

Thus, **before the beginning of the algorithm** should verify that the number N is odd, greater than 2 and not a perfect square, otherwise (under penalty of at least one condition) should immediately withdraw from the algorithm with the result of "composite".

Finally, we note that if D for some number N is too large, then the algorithm from a computational point of view would be inapplicable. Although in practice this was not observed (exerted enough 4-byte number), however the likelihood of this event should not be excluded. However, for example, in the interval [1;

106] max (D) = 47, and on the interval [1,019; 1019 + 106] max (D) = 67. Furthermore, Bailey and Vagstaf 1980 analytically proved this observation (see Ribenboim, 1995/96, page 142).

Strong algorithm Lucas

Algorithm parameters Lucas numbers are **D, P and Q** such that $D = P^2 - 4 * Q \neq 0$ and $P > 0$. (Easy to see that the parameters calculated by the algorithm Selfridge satisfy these conditions) **Lucas sequence** - this sequence U_k and V_k , defined as follows:

$$\begin{aligned} U_0 &= 0 \\ U_1 &= 1 \\ U_k &= P \cdot U_{k-1} - Q \cdot U_{k-2} \\ V_0 &= 2 \\ V_1 &= P \\ V_k &= P \cdot V_{k-1} - Q \cdot V_{k-2} \end{aligned}$$

Next, let $M = N - J(D, N)$.

If N is prime, and $\gcd(N, Q) = 1$, then we have:

$$UM \equiv 0 \pmod{N}$$

B Specifically, when the parameters D, P, Q calculated Selfridge algorithm, we have:

$$UN + 1 \equiv 0 \pmod{N}$$

The converse is not generally true. Nevertheless, pseudosimple numbers in this algorithm is not very much on what, in fact, is based algorithm Lucas.

Thus, **Lucas algorithm is to calculate the UM and comparing it with zero**.

Next, you need to find some way to speed up computations UK, otherwise, it is clear no practical sense in this algorithm would not.

We have:

$$U_k = (a_k - b_k) / (a - b),$$

$$V_k = a_k + b_k,$$

where a and b - the distinct roots of the quadratic equation $x^2 - P x +$

$Q = 0$. Now we can prove the following equation is simple:

$$U_{2k} \equiv U_k \cdot V_k \pmod{N}$$

$$V_{2k} \equiv V_k^2 - 2 \cdot Q_k \pmod{N}$$

Now, imagine if $M = E \cdot 2T$, where E - an odd number, it is easy to get:

$$UM = UE \cdot VE \cdot V2E \cdot V4E \dots V2T-2E \cdot V2T-1E \equiv 0 \pmod{N},$$

and at least one of the factors is zero modulo N.

It is understood that **it suffices to compute UE and VE** And all subsequent factors $V2E \cdot V4E \dots V2T-2E \cdot V2T-1E$ one can **get out of them already**.

Thus, it remains to learn how to quickly calculate the UE and VE odd E.

First, consider the following formulas for the addition of members of Lucas sequences:

$$Ui + j = (Ui \cdot Vj + Uj \cdot Vi) / 2 \pmod{N}$$

$$Vi + j = (Vi \cdot Vj + D \cdot Ui \cdot Uj) / 2 \pmod{N}$$

Note that the division is performed in the field ($\text{mod } N$). These formulas are proved very

simple and here are their proofs are omitted.

Now, having the formulas for addition and doubling sequences members Lucas and understandable way to accelerate computing UE and VE.

Indeed, consider the binary representation of the number of E. We first result - UE and VE - equal, respectively, U1 and V1. Go through the bits of E from younger to older, skipping only the first bit (the initial term of the sequence). For each i-th bit of U2 i will calculate and V2 i of

previous members with the aid of doubling. Furthermore, if the current i-th bit is one, that will add to the response current U2 i and V2 i using addition formulas. At the end of the algorithm that runs in O

($\log(E)$), we have obtain the desired UE and VE.

If the UE or VE were zero ($\text{mod } N$), N is a prime number (or pseudosimple). If both are different from

zero, then calculate $V2E, V4E, \dots, V2T-2E, V2T-1E$. If at least one of them is comparable to zero modulo N, the number N is prime (or pseudosimple). Otherwise, the number N is composite.

Discussion of the algorithm Selfridge

Now that we have looked at Lucas algorithm, we can elaborate on its parameters D, P, Q, one of the ways upon which the algorithm is Selfridge.

Recall the basic requirements for the parameters:

```
P > 0,  
D = P^2 - 4 * Q? 0.
```

Now we continue the study of these parameters.

D should not be a perfect square ($\text{mod } N$).

Indeed, otherwise we get:

$D = b^2$, hence $J(D, N) = 1$, $P = b + 2$, $Q = b + 1$, it $U_{n-1} = (Q^{n-1} - 1) / (Q - 1)$.

Ie if D - perfect square, then the algorithm becomes almost Lucas conventional probabilistic test. One of

the best ways to avoid this - **require that $J(D, N) = -1$** .

For example, it is possible to select the first sequence number D of 5, -7, 9, -11, 13, ... for which $J(D, N) = -1$.

Also suppose $P = 1$. Then $Q = (1 - D) / 4$. This method was proposed Selfridge.

However, there are other ways to select D. You can select it from the sequence 5, 9, 13, 17, 21, ... Also, let P - the smallest odd, privoskhodyaschee sqrt(D). Then $Q = (P^2 - D) / 4$.

It is clear that the choice of a particular method of calculating the parameters depends Lucas and its result - pseudosimple may vary for different methods of parameter selection. As shown, the algorithm proposed by Selfridge, was very successful all pseudosimple Lucas-Selfridge are not pseudosimple Miller-Rabin, at least, no counterexample was found.

Implementing a strong algorithm Lucas-Selfridge

Now you only have to implement the algorithm:

```
template <class T, class T2>  
bool lucas_selfridge (const T & n, T2 unused)  
{  
  
    // first check for trivial cases if (n  
    == 2)  
        return true;  
    if (n < 2 || even(n))  
        return false;  
  
    // check that n is not a perfect square, otherwise the algorithm  
    give an error  
    if (perfect_square (n))  
        return false;  
  
    // Selfridge algorithm: find the first number d such that:
```

```

// Jacobi (d, n) = -1, and it belongs to the range of {5, -7.9, -
11.13 ...}
T2 dd;
for (T2 d_abs = 5, d_sign = 1;; d_sign =-d_sign, + + + d_abs)
{
    dd = d_abs * d_sign;
    T g = gcd (n, d_abs);
    if (1 <g && g <n)
        / / Find divisor - d_abs
        return false;
    if (jacobi (T (dd), n) == -
        1) break;
}

// parameters Selfridge
T2
p = 1,
q = (p * p - dd) / 4;

// decompose n +1 = d * 2 ^ s
T n_1 = n;
+ + N_1;
T s, d;
transform_num (n_1, s, d);

/ / Algorithm Lucas
T
u = 1, v
= p, u2m
= 1, v2m
= p, qm
= q,
qm2 = q *
2, qkd =
q;
for (unsigned bit = 1, bits = bits_in_number (d); bit <bits; bit +
+)
{
    mulmod (u2m, v2m, n);
    mulmod (v2m, v2m, n);
    while (v2m <qm2)
        v2m += n;
    v2m -= qm2;
    mulmod (qm, qm, n);
    qm2 = qm;
    redouble (qm2);
    if (test_bit (d, bit))
    {
        T t1, t2;
        t1 = u2m;
        mulmod (t1, v, n);
        t2 = v2m;
        mulmod (t2, u, n);

        T t3, t4;
        t3 = v2m;
        mulmod (t3, v, n);
        t4 = u2m;
        mulmod (t4, u, n);
        mulmod (t4, (T) dd,
n);

        u = t1 + t2;
        if (! even
(u))
            u += n;
        bisect (u);
        u% = n;

        v = t3 + t4;
        if (! even
(v))

```

```
v +=  
n; bisect(v);  
v% = n;  
mulmod(qkd, qm, n);  
}
```

```

    } // just simple (or pseudo-prime) if
    (u == 0 || v == 0)
        return true;

    // dovychislyuem remaining members
    T qkd2 = qkd;
    redouble (qkd2);
    for (T2 r = 1; r <s; ++r)
    {
        mulmod (v, v, n);
        v -= qkd2;
        if (v <0) v += n;
        if (v <0) v += n;
        if (v> = n) v -=
n; if (v> = n) v -
= n; if (v == 0)
            return true;
        if (r <s-1)
        {
            mulmod (qkd, qkd, n);
            qkd2 = qkd;
            redouble (qkd2);
        }
    }

    return false;
}

```

Code BPSW

Now it remains to simply combine the results of all three tests: checking for small trivial divisors, the Miller-Rabin test, test strong Lucas-Selfridge.

```

template <class T>
bool baillie_pomerance_selfridge_wagstaff (T n)
{
    // first check for trivial divisors - for example, to 29 int
    div = prime_div_trivial (n, 29);
    if (div == 1)
        return true;
    if (div> 1)
        return false;

    // Miller-Rabin test with base 2
    if (!miller_rabin (n,
        2))
        return false;
    // Strong test Lucas-Selfridge
    return lucas_selfridge (n, 0);
}

```

Hence one can download the program (Source + Exe), containing the full implementation of the test BPSW. [77 KB]

Quick implementation

Code length can be greatly reduced at the expense of flexibility, abandoning templates and various support functions.

```
const int trivial_limit = 50;
```

```

int p [1000];

int gcd (int a, int b) {
    return a? gcd (b% a, a): b;
}

int powmod (int a, int b, int m)
{int res = 1;
while (b)
    if (b & 1)
        res = (res * 111 * a)% m, - b;
    else
        a = (a * 111 * a)% m, b >> = 1;
return res;
}

bool miller_rabin (int n) {
    int b = 2;
    for (int g; (g = gcd (n, b)) != 1; ++
b)
        if (n> g)
            return false;
    int p = 0, q = n-1;
    while ((q & 1) == 0)
        ++
p,     q >> = 1;
    int rem = powmod (b, q, n);
    if (rem == 1 || rem == n-1)
        return true;
    for (int i = 1; i <p; ++ i) {
        rem = (rem * 111 * rem)% n;
        if (rem == n-1) return true;
    }
    return false;
}

int jacobi (int a, int b)
{
    if (a == 0)    return 0;
    if (a == 1)    return 1;
    if (a <0)
        if ((b & 2) == 0)
            return jacobi (-a, b);
        else
            return - jacobi (-a, b);
    int a1 = a, e = 0;
    while ((a1 & 1) == 0)
        a1 >> = 1, ++
e;
    int s;
    if ((e & 1) == 0 || (b & 7) == 1 || (b & 7) ==
7) s = 1;
    else
        s = -1;
    if ((b & 3) == 3 && (a1 & 3) == 3)
        s = -s;
    if (a1 == 1)
        return s;
    return s * jacobi (b% a1, a1);
}

bool bpsw (int n) {
    if ((int) sqrt (n +0.0) * (int) sqrt (n +0.0) == n) return
false; int dd = 5;
    for (; ; ) {
        int g = gcd (n, abs (dd));
        if (1 <g && g <n) return false;
        if (jacobi (dd, n) == -1) break;
        dd = dd <0? -Dd +2:-dd-2;
    }
    int p = 1, q = (p * p-
dd) / 4; int d = n +1,
s = 0;

```

```
while ((d & 1) == 0)
    ++ s, d >>
= 1;
```

```

long long u = 1, v = p, u2m = 1, v2m = p, qm = q, qm2 = q * 2, qkd =
q;
for (int mask = 2; mask <= d; mask <<= 1) {
    u2m = (u2m * v2m) % n;
    v2m = (v2m * v2m) % n;
    while (v2m < qm2)      v2m += n;
    v2m -=
    qm2;
    qm = (qm * qm) % n;
    qm2 = qm * 2;
    if (d & mask) {
        long long t1 = (u2m * v) % n,      t2 = (v2m * u) % n,
* Dd) % n;
        u = t1 + t2;
        if (u & 1)  u += n;
        u = (u >> 1) % n;
        v = t3 + t4;
        if (v & 1)  v += n;
        v = (v >> 1) % n;
        qkd = (qkd * qm) % n;
    }
}
if (u == 0 || v ==
0)          return true;
long long qkd2 = qkd * 2;
for (int r = 1; r <s; ++ r)
{
    v = (v * v) % N - qkd2;
    if (v <0)  v += n;
    if (v <0)  v += n;
    v -=
    if (v> = n)  n;
    v -=
    if (v> = n)  n;
    if (v == 0)  return true;
    if (r <s-1) {
        qkd = (qkd * 111 * qkd) % n;
        qkd2 = qkd * 2;
    }
}
return false;
}

bool prime (int n) { / / this function must be called to check on simplicity
for (int i = 0; i <trivial_limit && p [i] <n; + + i)
    if (n% p [i] == 0)
        return false;
    if (p [trivial_limit-1] * p [trivial_limit-1]>
        = n) return true;
    if (! miller_rabin
        (n)) return
        false;
    return bpsw (n);
}

void prime_init () { / / called before the first call to
prime ()! for (int i = 2, j = 0; j <trivial_limit;
+ + i) {
    bool pr = true;
    for (int k = 2; k * k <=
        i; + + k) if (i% k
        == 0)
            pr = false;
    if (pr)
        p [j + +] = i;
}

```

Heuristic proof-refutation Pomerance

Pomerance in 1984 proposed the following heuristic proof.

Adoption: **Number BPSW-pseudosimple from 1 to X more X1-a any a>0.**

Proof.

Let k> 4 - arbitrary but fixed number. Let T - some big number.

Let $P_k(T)$ - the set of primes p in the interval $[T; Tk]$, for which:

(1) $p \equiv 3 \pmod{8}$, $J(5, p) = -1$

(2) The number $(p-1)/2$ is not a perfect square

(3) The number $(p-1)/2$ is composed solely of ordinary $q < T$

(4) The number $(p-1)/2$ is composed solely of prime q , $q \equiv 1 \pmod{4}$

(5) number $(p+1)/4$ is not a perfect square

(6) The number $(p+1)/4$ composed exclusively of common $d < T$

(7) number $(p+1)/4$ is composed solely of simple d , that $q \equiv 3 \pmod{4}$

It is understood that about $1/8$ of all prime in the interval $[T; Tk]$ satisfies the condition (1). Also it can be shown that the conditions (2) and (5) retain a certain part number. Heuristically, the conditions (3) and (6) also allows us to leave some of the numbers from the interval $(T; Tk)$. Finally, the event (4) has a probability of $(c(\log T) - 1/2)$, as well as an event (7). Thus, the cardinality of the set $P_k(T)$ is approximately at $T \rightarrow \infty$

$$\frac{cT^k}{\log^2 T}$$

where c - is a positive constant depending on the choice of k .

Now we can construct a number n that is not a perfect square, composed of simple l of $P_k(T)$, where l is odd and

less than $T^{2/3} \log(Tk)$. Number of ways to choose a number n is approximately

for large T and fixed k . Further, each n is a number less eT^2 .

Denoted by Q_1 product of prime $q < T$, for which $q \equiv 1 \pmod{4}$, and in Q_3 - a product of primes $q < T$, for which $q \equiv 3 \pmod{4}$.

Then $\gcd(Q_1, Q_3) = 1$, and $Q_1 Q_3 \leq eT$. Thus, the number of ways to select n

with the additional conditions

$$n \equiv 1 \pmod{Q_1}, \quad n \equiv -1 \pmod{Q_3}$$

be heuristically at least

$$e^{T^2(1 - 3/k)} / e^{-2T} \geq e^{T^2(1 - 4/k)}$$

for large T .

But each such n - is a counterexample to the test BPSW. Indeed, n is the number of Carmichael (ie, the number on which the Miller-Rabin test is wrong for any reason), so it will automatically pseudosimple base 2. Since $n \equiv 3 \pmod{8}$ and each $p | n \equiv 3 \pmod{8}$, it is obvious that n will pseudosimple strong base 2. Since $J(5, n) = -1$, then every prime $p | n$ satisfies $J(5, p) = -1$, and since the $p + 1 | n + 1$ for every prime $p | n$, it follows that n - pseudosimple Lucas Lucas for any test with discriminant 5.

Thus, we have shown that for any fixed k and all large T , will be at least $e^{T^2(1 - 4/k)}$ counterexamples to test BPSW of numbers less than eT^2 . Now, if we put $x = eT^2$, will be at least $x^{1 - 4/k}$ counterexamples smaller x . Since k - a random number, then our evidence indicates that

number of counterexamples, smaller x , is a number greater than x^{1-a} for any $a > 0$.

Appendix. All programs

[Download](#) all programs of this article. [242 KB]

Literature

Usable me literature is available online: 1. Robert Baillie;

Samuel S. Wagstaff

Lucas pseudoprimes

Math. Comp. 35 (1980) 1391-1417

[mpqs.free.fr / LucasPseudoprimes.pdf](http://mpqs.free.fr/LucasPseudoprimes.pdf)

2. Daniel J. Bernstein

Distinguishing prime numbers from composite numbers: the state of the art in 2004

Math. Comp. (2004)

cr.yp.to/primetests/prime2004-20041223.pdf

3. Richard P. Brent

Primality Testing and Integer Factorisation

The Role of Mathematics in Science (1990)

[wwwmaths.anu.edu.au / ~ brent/pd/rpb120.pdf](http://wwwmaths.anu.edu.au/~brent/pd/rpb120.pdf)

4. H. Cohen; H. W. Lenstra

Primality Testing and Jacobi Sums

Amsterdam (1984)

www.openaccess.leidenuniv.nl/bitstream/1887/2136/1/346_065.pdf

5. Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest

Introduction to Algorithms

[Without reference]

The MIT Press (2001)

6. M. Martin

PRIMO - Primality Proving

www.ellipsa.net

7. F. Morain

Elliptic curves and primality proving

Math. Comp. 61 (203) (1993) citeseer.ist.psu.edu/rd/43190198%2C72628%2C1%2C0.25%2CDownload/ftp%2AqSqqSqftp.inria.frqSqlNRlAqSqpPublicationqSqp publi-ps-gzqSqrRqSqrR-1256.ps.gz

2CDDownload/ftp%2AqSqqSqftp.inria.frqSqlNRlAqSqpPublicationqSqp publi-ps-gzqSqrRqSqrR-1256.ps.gz

8. Carl Pomerance

Are there counter-examples to the Baillie-PSW primality test?

Math. Comp. (1984)

[www.pseudoprime.com / dopo.pdf](http://www.pseudoprime.com/dopo.pdf)

9. Eric W. Weisstein

Baillie-PSW primality test

MathWorld (2005) [mathworld.wolfram.com /](http://mathworld.wolfram.com/)

[Baillie-PSWPrimalityTest.html](http://mathworld.wolfram.com/BailliePSWPrimalityTest.html)

10. Eric W. Weisstein

Strong Lucas pseudoprime

MathWorld (2005) [mathworld.wolfram.com /](http://mathworld.wolfram.com/)

[StrongLucasPseudoprime.html](http://mathworld.wolfram.com/StrongLucasPseudoprime.html)

11. Paulo Ribenboim

The Book of Prime Number Records

Springer-Verlag

(1989) [no link]

List of recommended books, which I could not find on the Internet:

12. Zhaiyu Mo; James P. Jones

A new primality test using Lucas sequences

Preprint (1997)

13. Hans Riesel

Prime numbers and computer methods for factorization

Boston: Birkhauser (1994)

Efficient algorithms for factorization

Here are the implementation of several factorization algorithms, each of which individually can work as quickly or very slowly, but together they provide a very fast method.

Descriptions of these methods are, the more that they are well described on the Internet.

Method Pollard p-1

Probabilistic test gives fast response is not for all properties. Returns either found

divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_p_1 (T n)
{
    // algorithm parameters significantly affect performance
    // search quality
    const T b = 13;
    const T q [] = {2, 3, 5, 7, 11, 13};

    // Several attempts algorithm
    T a = 5% n;
    for (int j = 0; j <10; j + +)
    {

        // is looking for a, which is relatively
        prime to n while (gcd (a, n) != 1)
        {
            mulmod (a, a, n);
            a += 3;
            a% = n;
        }

        // calculate a ^ M
        for (size_t i = 0; i <sizeof q / sizeof q [0]; i + +)
        {
            T qq = q [i];
            T e = (T) floor (log ((double) b) / log ((double)
                qq)); T aa = powmod (a, powmod (qq, e, n), n);
            if (aa == 0)
                continue;

            // Check whether the answer is not found
            T g = gcd (aa-1, n);
            if (1 <g && g <n)
                return g;
        }
    }

    // If nothing found
    return 1;
}
```

Pollard's method of "Po"

Probabilistic test gives fast response is not for all properties. Returns either found

divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_rho (T n, unsigned iterations_count = 100000)
```

```

    T
        b0 = rand ()% n,
        b1 =
        b0, g;
    mulmod (b1, b1, n);
    if (+ + b1 == n)
        b1 = 0;
    g = gcd (abs (b1 - b0), n);
    for (unsigned count = 0; count <iterations_count && (g == 1 || g == n); count++)
    {
        mulmod (b0, b0, n);
        if (+ + b0 == n)
            b0 = 0;
        mulmod (b1, b1, n);
        + + B1;
        mulmod (b1, b1, n);
        if (+ + b1 == n)
            b1 = 0;
        g = gcd (abs (b1 - b0), n);
    }
    return g;
}

```

Bent method (modification of the method of Pollard "Po")

Probabilistic test gives fast response is not for all properties. Returns either found

divider, or 1 if the divisor was not found.

```

template <class T>
T pollard_bent (T n, unsigned iterations_count = 19)
{
    T
        b0 = rand ()% n,
        b1 = (b0 * b0 + 2)% n, a = b1;
    for (unsigned iteration = 0, series_len = 1;
iteration <iterations_count; iteration + +, series_len
* = 2)
    {
        T g = gcd (b1-b0, n);
        for (unsigned len = 0; len <series_len && (g == 1 && g == n); len + +)
        {
            b1 = (b1 * b1 + 2)% n;
            g = gcd (abs (b1-b0), n);
        }
        b0 = a;
        a = b1;
        if (g != 1 && g != n)
            return g;
    }
    return 1;
}

```

Pollard's method of Monte Carlo

Probabilistic test gives fast response is not for all properties. Returns either found

divider, or 1 if the divisor was not found.

```

template <class T>
T pollard_monte_carlo (T n, unsigned m = 100)
{
    T b = rand ()% (m-2) 2 +;

    static std :: vector <T>
primes; static T m_max;
    if (primes.empty ())

```

```

        primes.push_back (3);
if (m_max <m)
{
    m_max = m;
    for (T prime = 5; prime <= m; + + + + prime)
    {
        bool is_prime = true;
        for (std :: vector <T> :: const_iterator iter =
primes. begin (), end = primes.end ());
            iter! = end; + + Iter)
        {
            T div = * iter;
            if (div * div>
                prime)
                break;
            if (prime% div == 0)
            {
                is_prime = false;
                break;
            }
        }
        if (is_prime)
            primes.push_back (prime);
    }
}

T g = 1;
for (size_t i = 0; i <primes.size () && g == 1; i + +)
{
    T cur = primes
    [i]; while (cur <=
n)
        cur * = primes
        [i]; cur / = primes [i];
    b = powmod (b, cur, n);
    g = gcd (abs (b-1), n);
    if (g == n)
        g = 1;
}

return g;
}

```

Method Farm

This wide method, but it can be very slow if you have small number of divisors. So run it costs only after all other methods.

```

template <class T, class T2>
T ferma (const T & n, T2 unused)
{
    T2
    x = sq_root (n),
    y = 0,
    r = x * x - y * y - n;
    for (; ;)
        if (r == 0)
            return x! = y? x-y: x + y;
        else
            if (r> 0)
            {
                r - = y + y
                +1; + + Y;
            }
            else
            {
                r + = x + x
                +1; + + X;
            }
}

```


Trivial division

This basic method is useful to immediately process numbers with very small divisors.

```
template <class T, class T2>
T2 prime_div_trivial (const T & n, T2 m)
{
    // first check for trivial cases if (n
    == 2 || n == 3)
        return 1;
    if (n < 2)
        return 0;
    if (even (n))
        return 2;

    // generate a simple 3 to m
    T2 pi;
    const vector <T2> & primes = get_primes (m, pi);

    // Divide by all simple
    for (std :: vector <T2> :: const_iterator iter = primes.begin (), end =
        primes.
end ());
        iter! = end; + + Iter)
    {
        const T2 & div = *
        iter; if (div * div > n)
            break;
        else
            if (n% div == 0)
                return div;
    }

    if (n < m * m)
        return 1;
    return 0;
}
```

Putting it all together

Combine all methods in a single function.

Also, the function uses the simplicity of the test, otherwise the factorization algorithms can work for very long. For example, you can select a test BPSW ([read article BPSW](#)).

```
template <class T, class T2>
void factorize (const T & n, std :: map <T,unsigned> & result, T2 unused)
{
    if (n == 1)
        ;
    else
        // Check if not a prime number
        if (isprime (n))
            + + Result [n];
        else
            // If the number is small enough, it
expand the simple search
            if (n < 1000 * 1000)
            {
                T div = prime_div_trivial (n, 1000);
                + + Result [div];
                factorize (n / div, result, unused);
            }
            else
            {
                // Number of large, run it
                T div;
```

```
// first go fast algorithms Pollard div =
pollard_monte_carlo (n);
if (div == 1)
    div = pollard_rho (n);
if (div == 1)
    div = pollard_p_1 (n);
if (div == 1)
    div = pollard_bent (n);
// have to run 100% Farm algorithm if (div
== 1)
    div = ferma (n, unused);
// recursively found factors factorize (div,
result, unused);
factorize (n / div, result, unused);
}
}
```

Application

[Download \[5 KB\]](#) source program that uses all of these methods and test factorization BPSW for simplicity.

Fast Fourier Transform for the O (N log N).

Application to the multiplication of two polynomials or long numbers

Here we consider an algorithm which allows to multiply two polynomials of length n during $O(n \log n)$ is much better than the time $O(n^2)$ achieved trivial multiplication algorithm. Obviously, multiplication of two long numbers can be reduced to the multiplication of polynomials, so the two long numbers can also multiply during $O(n \log n)$.

Fast Fourier Transform invention is attributed to Cooley (Coolet) and Taki (Tukey) - 1965 actually invented a FFT repeatedly before, but its importance is not fully recognized until the appearance of modern computers. Some researchers attribute the discovery FFT Runge (Runge) and Koenig (Konig) Finally in 1924, the discovery of this method is attributed to more Gaussian (Gauss) in 1805

Discrete Fourier transform (DFT)

Suppose there is a polynomial n First degree:

$$A(x) = a_0x^0 + a_1x^1 + \dots + a_{n-1}x^{n-1}.$$

Without loss of generality, we can assume that n is a power of 2. When actually n not a power of 2, then we just add the missing coefficients, setting them equal to zero.

Theory of functions of a complex variable is known that the complex roots n Th roots of unity there are exactly n . We denote these roots by $w_{n,k}, k = 0 \dots n-1$, Then it is known that $w_{n,k} = e^{i\frac{2\pi k}{n}}$. In addition, one of these roots $w_n = w_{n,1} = e^{i\frac{2\pi}{n}}$ (Called the principal value of the root n -th roots of unity) is that all the other roots are its powers: $w_{n,k} = (w_n)^k$.

Then **discrete Fourier transform (DFT)** (Discrete Fourier transform, DFT) of the polynomial $A(x)$ (Or, equivalently, the DFT vector of coefficients) are the values of this polynomial at the points $x = w_{n,k}$ le is the vector:

$$\begin{aligned} \text{DFT}(a_0, a_1, \dots, a_{n-1}) &= (y_0, y_1, \dots, y_{n-1}) = (A(w_{n,0}), A(w_{n,1}), \dots, A(w_{n,n-1})) = \\ &= (A(w_n^0), A(w_n^1), \dots, A(w_n^{n-1})). \end{aligned}$$

Defined similarly **inverse discrete Fourier transform** (InverseDFT).

Inverse DFT to the vector of a polynomial - is the vector of coefficients of the polynomial $(a_0, a_1, \dots, a_{n-1})$:

$$\text{InverseDFT}(y_0, y_1, \dots, y_{n-1}) = (a_0, a_1, \dots, a_{n-1}).$$

Thus, if the direct proceeds from the DFT coefficients of the polynomial to its values in the complex roots n - Th roots of unity, the inverse DFT - on the contrary, by the values of the coefficients of the polynomial recovers.

Application of DFT for fast multiplication of polynomials

Given two polynomial A and B . Calculate the DFT for each of them: $\text{DFT}(A)$ and $\text{DFT}(B)$ - Two vector value polynomials.

Now, what happens when the multiplication of polynomials? Obviously, in each point of their values are simply multiplied, that is,

But this means that if we multiply the vector $\text{DFT}(A)$ and $\text{DFT}(B)$ Simply multiplying each element a vector to the corresponding element of another vector, then we get nothing else than from DFT polynomial $A \times B$:

Finally, applying the inverse DFT, we obtain:

$$A \times B = \text{InverseDFT}(\text{DFT}(A) \times \text{DFT}(B)),$$

where, again, right under the product of two DFT mean pairwise products of the elements of the vectors.

Such a product is obviously required to calculate only $O(n)$ operations. Thus, if we

learn how to calculate the DFT and inverse DFT during n . Then the product of two polynomials (a, consequently, the two long numbers) we can find for the same asymptotic behavior.

It should be noted that the first two polynomials to be lead to the same degree (simply adding one of these coefficients with zeros.) Second, the product of two polynomials of degree n obtain a polynomial of degree $2n - 1$, So that the result is correct, you need to double the pre-degree of each polynomial (again, adding to their zero coefficients).

Fast Fourier transform

Fast Fourier transform (Fast Fourier transform) - this method allows us to compute the DFT over time. This method relies on the properties of the complex roots of unity (namely, that extent that the same roots give other roots).

The basic idea is to divide the FFT coefficient vector into two vectors, the recursive computation of the DFT for them and merging the results into a single FFT.

Thus, suppose there is a polynomial $A(x)$ degrees n Wherein n - A power of two, and $n > 1$:

$$A(x) = a_0x^0 + a_1x^1 + \dots + a_{n-1}x^{n-1}.$$

Divide it into two polynomials, one - with even and the other - with the odd coefficients:

$$\begin{aligned} A_0(x) &= a_0x^0 + a_2x^1 + \dots + a_{n-2}x^{n/2-1}, \\ A_1(x) &= a_1x^0 + a_3x^1 + \dots + a_{n-1}x^{n/2-1}. \end{aligned}$$

It is easy to verify that:

an
Polynomials A_0 and A_1 have twice the smaller degree than the polynomial A . If we can in linear time calculated by $\text{DFT}(A_0)$ and $\text{DFT}(A_1)$ calculate $\text{DFT}(A)$, Then we obtain the desired algorithm Fast Fourier Transform (as is the standard chart of "divide and conquer", and for her known asymptotic estimate $O(n \log n)$).

So, suppose we have calculated the vector $\{y_k^0\}_{k=0}^{n/2-1} = \text{DFT}(A_0)$ and $\{y_k^1\}_{k=0}^{n/2-1} = \text{DFT}(A_1)$.

Find expressions for $\{y_k\}_{k=0}^{n-1} = \text{DFT}(A)$.

First, recalling (1), we immediately obtain the values for the first half of the coefficients:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \dots n/2 - 1.$$

For the second half of the coefficients after transformation also get a simple formula:

$$\begin{aligned} y_{k+n/2} &= A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) = A_0(w_n^{2k} w_n^n) + w_n^k w_n^{n/2} A_1(w_n^{2k} w_n^n) = \\ &= A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1. \end{aligned}$$

(Here we have used (1), as well as the identities $w_n^n = 1$, $w_n^{n/2} = -1$.) So, as a

result we got the formula to calculate the total vector $\{y_k\}$:

(These formulas, ie two formulas of the form $a + bc$ and $a - bc$, Sometimes called the "transformation Butterflies (" butterfly operation "))

Thus, we finally built the FFT algorithm.

IFFT

So, let a vector $(y_0, y_1, \dots, y_{n-1})$. Values of the polynomial A degrees n at points $x = w_n^k$.

Need to recover the coefficients $(a_0, a_1, \dots, a_{n-1})$ polynomial. This well-known problem called **interpolation**. For this task, there are some common algorithms for solving, but in this case is obtained by a very simple algorithm (a simple fact that it does not differ from the direct FFT).

DFT, we can write, according to his definition, in matrix form:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

The vector $(a_0, a_1, \dots, a_{n-1})$ can be found by multiplying the vector by the inverse matrix to the matrix, stands to the left (which, incidentally, is called Vandermonde matrix):

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Direct verification shows that this inverse matrix is as follows:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}.$$

Thus, we obtain:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}.$$

Comparing it with the formula for y_k :

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj},$$

we notice that these two problems are not real, so that the coefficients a_k . You can find such a same algorithm "divide and rule" as a direct FFT, but instead should be used everywhere, and each element of the result must be divided by n .

Thus, the calculation of the inverse DFT is not very different from the direct computation of the DFT, and can also be performed during the $O(n \log n)$.

Implementation

Consider a simple recursive **FFT implementation** and IFFT sell them as one function, because the differences between the direct and inverse FFT low. For storage complex numbers use the standard in C ++ STL type complex (defined in the header file `<complex>`).

```
typedef complex<double> Base;

void fft (vector<Base> & a, bool invert)
{
    int n = (int) a.size();
```

```

if (n == 1) return;

vector<base> a0 (n /2), A1 (n
/2); for (int i =0, J =0; i <n; i +
=2, + + J) {
    a0[j] = A[i];
    a1[j] = A[i
+1];
}
fft (a0, invert);
fft (a1, invert);
double ang = 2* PI / n * (invert? -1 :
1); base w (1), Wn
(cos(ang),sin(ang)); for (int i =0; i <n
/2; + + I) {
    a[i] = A0[i] + W * a1[i]; a[i
+ n /2] = A0[i] - W *
a1[i]; if (invert)
        a[i] /= 2, A[i + n /2] /= 2;
    w *= wn;
}
}

```

In argument **a** function takes an input vector of coefficients, and it will contain the same result. Argument shows direct or inverse DFT should be calculated. Inside the function first checks if the length of the vector **a** is unity, there is nothing else to do - he is the answer. Otherwise, the vector **a** is divided into two vectors **a0** and **a1**For which recursively calculated DFT. It then calculates the value of **w_n**,

Starts and variable **w** Containing the current degree of **w_n**. Then calculated by DFT elements resulting formulas described above.

If the flag is specified **invert = true** Then **w_n** is replaced by $\frac{w_n}{2}$, as each element of the result is divided by 2 (Given that these dividing by 2 occur in each level of recursion, then eventually just turns out that all the elements on the share **n**).

Then the function for **multiply two polynomials** will be as follows:

```

void multiply (const vector<int> & A, const vector<int> & B, vector
<int> & res) {
    vector<base> fa (a.begin(),
A.end()), fb (b.begin(), B.end());
    size_t n = 1;
    while (n <max (a.size(), b.size())) n << = 1;
    n << = 1;
    fa.resize (n), fb.resize (n);
    fft (fa, false), Fft (fb,
false); for (size_t i =0; i <n; + +
I)
        fa[i] *=
        fb[i]; fft (fa, true);
    res.resize (n);
    for (size_t i =0; i <n; + + I)
        res[i] = int (fa[i].Real () + 0.5);
}

```

This function works with polynomials with integer coefficients (although, of course, theoretically nothing prevents her work with fractional coefficients). However, it appears the problem of a large error in the DFT: error can be significant, so better to round the number the surest way - by adding 0.5 and then rounding down.

Finally, the function for **multiplying two long numbers** practically does not differ from the function for multiplying polynomials. The only feature - that after the multiplication of numbers as they should normalize polynomials, ie perform all the carries bits:

```

int carry = 0;
for (size_t i =0; i <n; + +
I) { res[i] +=
    Carry; carry =
    res[i] / 10; res[i]
    % = 10;
}

```

(Since the length of the product of two numbers never exceed the total length of numbers, the size of the vector `res` enough to perform all the carries.)

Improved execution: computing "on the spot" without additional memory

To increase the efficiency abandon recursion explicitly. In the above recursive implementation, we clearly separated vector a two vector - elements on even positions attributed to the same time creates a vector, and on the odd - to another. However, if we reorder elements in a certain way, the need for creating temporary vectors would then be eliminated (ie, all the calculations we could produce "on the spot" right in the vector a).

Note that the first level of recursion elements, junior (first) position bits are zero, refer to the vector a_0 And the least significant bits positions which are equal to one - to the vector a_1 . On the second level of recursion is done the same thing, but for the second bit, etc. So if we are in a position i each element

$a[i]$ invert the order of the bits, and reorder elements of the array a in accordance with the new indices, we and obtain the desired order (it is called **bitwise inverse permutation** (Bit-reversal permutation)).

For example, for $n = 8$ this order is as follows:

$$a = \left\{ \left[(a_0, a_4), (a_2, a_6) \right], \left[(a_1, a_5), (a_3, a_7) \right] \right\}.$$

Indeed, on the first level of recursion (surrounded by curly braces) conventional recursive algorithm is a division of the vector into two parts: and, As we see in $[a_0, a_2, a_4, a_6], [a_1, a_3, a_5, a_7]$ bitwise inverse permutation of this vector corresponds to a division into two halves: the first element and the last $n/2$ element. Then there is a recursive call of each half; let the resulting DFT from each of them had returned to the place of the elements themselves (ie in the first and second halves of the vector a , respectively):

$$a = \left\{ \left[[y_0^0, y_1^0, y_2^0, y_3^0], [y_0^1, y_1^1, y_2^1, y_3^1] \right] \right\}.$$

Now we need to merge the two into one DFT for the vector. But the elements got so well that the union can be performed directly in the array. Indeed, we take the elements and is applicable to

$$y_0^0 \quad y_0^1$$

These butterfly transformation, and the result put in their place - and this place and would thus that it should have received:

$$a = \left\{ \left[[y_0^0 + w_n^0 y_1^1, y_1^0, y_2^0, y_3^0], [y_0^0 - w_n^0 y_1^1, y_1^1, y_2^1, y_3^1] \right] \right\}.$$

Similarly, we apply the transformation to a butterfly y_1^0 and y_1^1 and the result is put in their place, etc. As a result, we obtain:

$$a = \left\{ \left[[y_0^0 + w_n^0 y_1^1, y_1^0 + w_n^1 y_1^1, y_2^0 + w_n^2 y_2^1, y_3^0 + w_n^3 y_3^1], [y_0^0 - w_n^0 y_1^1, y_1^0 - w_n^1 y_1^1, y_2^0 - w_n^2 y_2^1, y_3^0 - w_n^3 y_3^1] \right] \right\}.$$

Ie We got exactly the desired DFT of the vector a .

We describe the process of calculating the DFT on the first level of recursion, but it is clear that the same arguments are valid for all other levels of recursion. Thus, **after applying bitwise inverse permutation to calculate the DFT can be in place**, without the involvement of additional arrays.

But now you can **get rid of the recursion** explicitly. So, we applied bitwise inverse permutation elements. Now do all the work done by the lower level of recursion, ie vector a divide into pairs of elements for each butterfly transform applied, resulting in the vector a will be the results of the lower level of recursion. In the next step we split vector a on Four elements, apply the transformation to each butterfly as a result we obtain the DFT for each foursome. And so on, finally, the last step we received the results of the DFT for the two halves of the vector a . Applicable to them and get butterflies transform DFT for the vector a .

Thus, the implementation of:

```
typedef complex<double> Base;

int rev (int num, int lg_n) {
    int res = 0;
```

```

        for (int i =0; i <lg_n; +
            + I) if (num &
            (1<< I))
                res |= 1<<(lg_n-1-I);
    return res;
}

void fft (vector <base> & a, bool invert)
{ int n = (int) a.size ();
int lg_n = 0;
while ((1 << Lg_n) <N) ++ Lg_n;

for (int i =0; i <n; ++ I)
    if (i <rev(i, lg_n))
        swap (a[i], A[rev(i, lg_n)]);

for (int len =2; len <= n; len << =1) {
    double ang = 2* PI / len * (invert? -1 :
    1); base wlen (cos(ang),sin(ang));
    for (int i =0; i <n; i += len) {
        base w (1);
        for (int j =0; j <len /2; ++ J) {
            base u = a[i + j], v = a[i + j + len
            /2] * W; a[i + j] = U + v;
            a[i + j + len /2] = U
            - v; w *= wlen;
        }
    }
}
if (invert)
    for (int i =0; i <n; +
        + I) a[i] /= N;
}

```

Initially, a vector bitwise inverse permutation is applied, which calculates the amount of significant bit (a)

($\lg n$ including n). And for each item is the corresponding position, which bit write have a bit representation of the number i written in reverse order. If the resulting position was as a result of more i , The elements in these two positions should be exchanged (unless this condition, each pair will exchange twice and in the end nothing happens.)

Then, the $\lg n - 1$ algorithm steps, on the second of which $\dots \lg n$ DFT block length. For all of these units will be the same value of a primitive root w_{2^k} , And is stored in a variable $wlen$. Loop iterates on the blocks, and invested in it by cycle j applies the transformation to all elements of the butterfly unit.

You can further **optimization of reverse bits**. In the previous implementation, we explicitly took over all bits of the number, incidentally bitwise inverted order number. However, reverse bits can be performed differently.

For example, suppose that - already counted the number equal to the number of bits reverse permutation i .

Then, during the transition to the next number we must add one to the number, but add it to a "Inverted" notation. In a conventional binary system to add one - so remove all units standing at the end of the number (ie, younger group of units), and put the unit in front of them. Accordingly, the "inverted" system, we have to go through the bits of the number, starting with the oldest, and while there are ones, delete them and move on to the next bit; when will meet the first zero bit, put him in a unit and stop.

Thus, we obtain a realization:

```

typedef complex <double> Base;

void fft (vector <base> & a, bool invert)
{ int n = (int) a.size ();
for (int i =1, J =0; i <n; + +
    I) { int bit = n >> 1;
        for (; j> = bit; bit >>
            =1) j -= bit;

```

```
j += bit;
if (i <j)
    swap (a[i], A[j]);
}
```

```

        for (int len = 2; len <= n; len <<= 1) {
            double ang = 2 * PI / len * (invert? -1 :
                1); base wlen (cos(ang),sin(ang));
            for (int i = 0; i <n; i += len) { base w (1);
                for (int j = 0; j <len /2; ++ J) {
                    base u = a[i + j], v = a[i + j + len
                        /2] * w; a[i + j] = u + v;
                    a[i + j + len /2] = u
                        - v; w *= wlen;
                }
            }
        }
        if (invert)
            for (int i = 0; i <n; +
                + I) a[i] /= N;
    }
}

```

Additional optimization

Here are a list of other optimizations, which together can significantly speed up the above "improved" implementation:

- **Predposchitat reversed bits** for all numbers in a global table. It is especially easy, when the size of n for all calls the same.

This optimization becomes noticeable when a large number of calls. However, its effect can be even notice when three calls (three calls - the most common situation, ie when it is required once to multiply two polynomials).

- Abandon the use of **vector** ([go to normal arrays](#)).

The effect of this depends upon the particular compiler, but is typically present and accounts for approximately 10% -20%.

- Predposchitat **all power** number. In fact, in this cycle algorithm repeatedly made through all the powers of $wlen$ from 0 to $\lfloor \frac{len}{2} \rfloor - 1$:

```

        for (int i = 0; i <n; i += len) { base w (1);
            for (int j = 0; j <len /2; ++ J) { [...]
                w *= wlen;
            }
        }
    }
}

```

Accordingly, before this cycle we can predposchitat from an array all the required extent, and thus get rid of unnecessary multiplications in a nested loop.

Tentative acceleration - 5-10%.

- Get rid of **array accesses Index**, Instead use pointers to the current array elements, advancing them to the right one at each iteration.

At first glance, optimizing compilers should be able to cope with this, but in practice it appears that the replacement of references to arrays $a[i + j]$ and $i + j + \lfloor \frac{len}{2} \rfloor$ pointers program accelerates common compilers. Profit is 5-10%.

- **Abandon the standard type of complex numbers**

complex Rewriting it to

own implementation.

Again, this may seem surprising, but even in modern compilers benefit from such rewriting can be up to several tens of percent! This indirectly confirms a widespread assertion that compilers perform worse with sample data types, optimizing work with them much worse than non-formulaic types.

- Another useful optimization is **clipping length** When the length of the working unit becomes small (say, 4), to calculate the DFT for it "manually". If the paint in these cases explicit formulas with a length equal to, the values of \sin , cosines take integer values, whereby it is possible to get a speed boost for a few tens of percent.

Let us here describe improved realization with (except the last two items which lead to overgrowth of code):

```

int rev[MAXN]; base
wlen_pw[MAXN];

void fft (base a[], int n, bool invert) {
    for (int i =0; i <n; + + I)
        if (i <rev[i])
            swap (a[i], A[rev[i]]);

    for (int len =2; len <= n; len << =1) {
        double ang = 2* PI / len * (invert? -
1+1); int len2 = len >>1;
        base wlen (cos(ang),sin(ang));
        wlen_pw[0] = Base (1,0);
        for (int i =1; i <len2; + + I)
            wlen_pw[i] = Wlen_pw[i-1] * Wlen;

        for (int i =0; i <n; i += len) { base t,
            * Pu = a + i,
            * Pv = a + i +
            len2, * pu_end = a
            + i + len2, * pw =
            wlen_pw;
            for (; pu! = pu_end; + + Pu, + + Pv, +
            + pw) { t = * Pv ** pw;
            * Pv = * pu -
            t; * Pu += t;
            }
        }
    }

    if (invert)
        for (int i =0; i <n; +
            + I) a[i] / =
            N;
}

void calc_rev (int n, int log_n) {
    for (int i =0; i <n; + +
        I) {
        rev[i] = 0;
        for (int j =0; j <log_n; +
            + J) if (i & (1<<
            J))
            rev[i] |= 1<<(log_n-1-J);
    }
}

```

On common compilers faster than the previous implementation of this "improved" version of 2-3.

Discrete Fourier transform in modular arithmetic

At the heart of the discrete Fourier transform are complex numbers, roots n Th roots of unity. Its effective computation used features such roots as the existence of n different roots, forming a group (ie, the degree of the same root - always another square, among them there is one element - the generator of the group, called a primitive root).

But the same is true of the roots n Th roots of unity in modular arithmetic. I mean, not for any module p there n different roots of unity, but these modules do exist. Still important for us to find among them a primitive root, ie:

$$(w_n)^n = 1 \pmod{p},$$

$$(w_n)^k \neq 1 \pmod{p}, \quad 1 \leq k < n.$$

All the rest $n - 1$ roots n Th roots of unity in modulus p . You can get your degree primitive root w_n (As in the complex case).

For use in the fast Fourier transform algorithm we needed to root primitive existed for some n , A power of two, and all the lesser degrees. And if in the complex case, there was a primitive root for any n , In the case of the modular arithmetic is generally not

so. However, note that if $n = 2^k$ is k -th power of two, the modulo $m = 2^{k-1}$ we have:

Thus, if w_n - A primitive root $n = 2^k$ th roots of unity, w_n^2 - A primitive root n th roots of unity. Therefore, all powers of two, smaller than 2^{k-1} th roots of unity also exist, and may be calculated as appropriate degree w_n .
 n primitive roots desired w_n .

The final touch - for inverse DFT we used instead w_n inverse element w_n^{-1} . But simple module inverse is also always there.

Thus, all the required properties observed in the case of modular arithmetic, as long as we choose some fairly large module p and found in it a primitive root-degree of unity. For example, you can take the following values:
module $p = 7340033$ $w_{2^{20}} = 5$. If this module is not enough to find another pair, you can use the fact that for the modules of the form $c2^k + 1$ still necessarily simple) there is always primitive root 2^k of the unit.

```
const int mod = 7340033;
const int root = 5;
const int root_1 = 4404020;

const int root_pw = 1<<20;

void fft (vector <int> & A, bool invert)
{
    int n = (int) a.size();

    for (int i = 1, j = 0; i < n; i +=
        I) { int bit = n >> 1;
        for (; j >= bit; bit >>
            =1) j -= bit;
        j += bit;
        if (i < j)
            swap (a[i], a[j]);
    }

    for (int len = 2; len <= n; len << =1) {
        int wlen = invert? root_1: root;
        for (int i = len; i <root_pw; i <<
            =1)
            wlen = int (wlen * 111 * wlen%
            mod); for (int i = 0; i <n; i += len) {
                int w = 1;
                for (int j = 0; j <len / 2; + + J) {
                    int u = a[i +      v = int (a[i + j + len / 2]
                        j],                      * 111
* W% mod);
                    a[i + j] = U + v <mod? u + v: u + v-mod;
                    a[i + j + len / 2] = U-v> = 0 ? u-v: u-v +
                    mod;
                    w = int (w * 111 * wlen% mod);
                }
            }
        if (invert) {
            int nrev = reverse (n, mod);
            for (int i = 0; i <n; + + I)
                a[i] = int (a[i] * 111 * nrev% mod);
        }
    }
}
```

Here, the function `reverse` is the inverse of the element modulo `mod` (Cf. [Inverse element in a field module](#)). Constants `mod`, `root`, `root_pw` specify the module and a primitive root, and `root_1` - Inverse root element modulo `mod`.

As practice shows, the implementation of integral DFT works even slower implementation of complex numbers (due to the sheer number of transactions taking absolute value), but it has advantages such as low memory usage and the lack of rounding errors.

Some applications

In addition to direct application for multiplying polynomials or long numbers, we describe here

some other applications of the discrete Fourier transform.

All possible sums

and. You want to find all sorts of numbers of the
Problem: given two arrays a and b form
deduce the number of ways to get it.
For example,
for $a = (1, 2, 3)$ and $b = (2, 4)$ obtain 3 can be obtained by method 1, 4 - and one 5 - 2
 $6 - 1 7 - 1$.

and. As in the polynomial degrees will be performing numbers themselves,
Construct the arrays and b two polynomials A and B ie
, the values $a[i]$ and $b[i]$) As well as the coefficients of them - the number of times it occurs in the array $b[i]$.

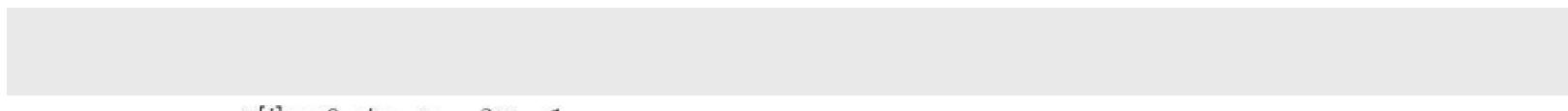
Then, multiplying these two polynomials in $O(n \log n)$, We obtain a polynomial C Wherein as degrees
will be all sorts of numbers of the form $a[i] \cdot b[i]$ And their coefficients are just the required number of

All kinds of scalar products

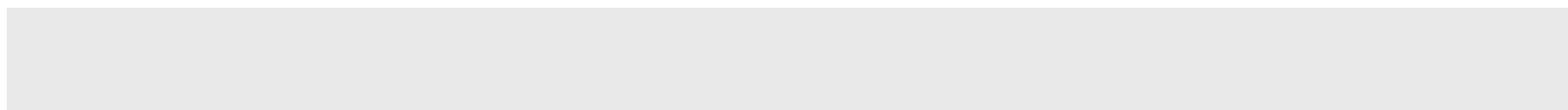
Given two arrays a and b of the same length and n . You want to display the values of each of the scalar product
of the vector a the next cyclic shift vector b .

Invert the array and assign it to the end of the zeros, and the array - simply assign himself.

Then multiply them as polynomials. Now consider the coefficients of the product $[n \dots 2n - 1]$ (As
always, all the indexes in the 0-indexed). We have:



Since all the elements $a[i] = 0$, $i = n \dots 2n - 1$, We obtain:



It is easy to see in this sum, it is the scalar product on
cyclic shift. Thus, these factors (since $a[n-i] = a[i]$)
to the problem.

$n - 1^{\text{Th}}$ and pumping $\frac{a}{2} - 2^{\text{Th}}$) - is the answer

The decision came with the asymptotic $O(n \log n)$.

Two strips

Two strips are defined as two Boolean (ie numerical values 0 or 1) and the array. Required
find all such positions in the first strip that if you apply, since this position, the second strip in any place will
not work **true** immediately on both strips. This problem can be reformulated as follows: given a map strips
as 0/1 - you can get up into the cell or not, and given some figure as a template (as an array, where 0 - no
cells, 1 - yes), requires Find all positions in the strip which can be attached figure.

This problem is actually no different from the previous problem - the problem of the scalar product. Indeed, the dot
product of two 0/1 arrays - the number of elements in which both units were. Our task is to find all cyclic shifts of the
second strip so that there was not a single item, which would be in both strips were unity. Ie we have to find all cyclic
shifts of the second array, in which the scalar product is zero.

Thus, this problem we decided for $O(n \log n)$.

Search width

Breadth-first search (bypassing width, breadth-first search) - is one of the basic graph algorithms.

As a result, breadth-first search is the shortest path length in nevzveshenennom graph, ie a path containing the smallest number of ribs.

The algorithm works for $O(n + m)$ Wherein n - The number of vertices m - The number of edges.

Description of the algorithm

Algorithm is applied to the input of the given graph (unweighted), and number of starting vertex s .

Graph can be as focused and undirected, for the algorithm is not important.

The algorithm can be understood as a process of "ignition" of the graph: at zero step ignite only the top s . At each step the fire already burning with each vertex spreads to all its neighbors; ie in one iteration of the algorithm is expanding "ring of fire" in width by one (hence the name of the algorithm).

More precisely, it is represented as follows. Create the queue that will be placed

burning tops and manages an array of boolean $used[]$ in which each vertex will celebrate lit she was or not (or in other words, was it had).

Initially placed in a queue just the tip and $s used[s] = true$ And for all other vertices $used[] = false$. Then the algorithm is a loop: while queue is not empty, get out of her head one vertex, see all the edges emanating from that vertex, and if any of the vertices visited still off, then set them on fire and put in the queue.

As a result, when the queue is empty, a detour to bypass the width of all reachable s vertices, and each comes to the shortest path. You can also calculate the length of the shortest paths (which just need to have an array of lengths of paths), and compact to save enough information to restore all of the shortest paths

(You have to make an array of "ancestors", in which each vertex store node number at which we got into this vertex).

Implementation

Implement the above algorithm in the

language C++. Input:

```
vector<vector<int>> G; //  
graph int n; // the number of  
vertices  
int s; // top of page (All vertices are numbered from zero)  
// Read graph  
...
```

Circumvention itself:

```
queue<int>  
Q; Q.push  
(s);  
vector<bool> Used (n);  
vector<int> D (n), P  
(n); used[s] = true;  
p[s] = -1;  
while (! Q.empty()) {  
    int v = q.front();  
    q.pop();  
    for (size_t i = 0; i < g[v].Size(); ++  
        i) { int to = g[v][i];  
        if (! Used[to]) {  
            used[to] = true;
```

```
q.push (to);
```

```

        d[to] = D[v] + 1;
        p[to] = v;
    }
}

```

If we now want to restore and display the shortest path to some vertex `to` This can be done as follows:

```

if (!Used[to])
    cout << "No path!";
else {
    vector<int> Path;
    for (int v = to; v != -1; v =
        p[v]) path.push_back
        (v);
    reverse(path.begin(),
    Path.end()); cout << "Path:";
    for (size_t i = 0; i < path.size(); +
        + i) cout << Path[i] + 1 <<
        "";
}

```

Application of the algorithm

- Search **shortest path** in unweighted graph.
- Search **connected components** in the column for $O(n + m)$.
To do this, we simply run wide detour from each vertex except vertices remaining visits (`used = true`) After previous launches. Thus, we perform normal start in width from each vertex, but do not reset each time an array `used[]`. Whereby every time we will get new connected component, and the total time of the algorithm will still $O(n + m)$ (Such multiple runs on a graph traversal without resetting the array `used` called a series of detours in width).
- Finding a solution to a problem (game) **with the smallest number of moves** If each state of the system can be represented by a vertex of, and transitions from one state to another - edges of the graph.
A classic example - a game where the robot moves along the field, while it can move boxes that are on the same field, and requires the least number of moves required to move boxes in position. Solved this detour in width along the graph, where the state (top) is a set of coordinates: coordinates of the robot, and the coordinates of all the boxes.
- Finding the shortest path in **0-1-column** (ie, weighted graph, but with weights equal only 0 or 1) is sufficient to slightly modify the breadth-first search, if the current edge of zero weight, and an improvement of the distance to some vertex, then this vertex does not add to the end, and of the queue.
- Location **shortest cycle** in an undirected unweighted graph: search width of each vertex; once during the traversal we try to go from the current vertex on some edge in the already visited vertices, then it means that we have found the shortest cycle and stop crawling in width; found among all such cycles (one from each run bypass) choose the shortest.
- Find all edges that lie **on any shortest path** between a given pair of vertices
And
You need to run two breadth-first search: from of . We denote array shortest distances obtained from the first bypass and through - As a result of the second passage. Now for any edge easy to check whether it is in any shortest path: the criterion will condition
- Find all vertices lying **on any shortest path** between a given pair of vertices
And
You need to run two breadth-first search: from of . We denote array shortest distances obtained from the first bypass and through - As a result of the second passage. Now for any vertex easy to check whether it is in any shortest path: the criterion will condition
- Find **even the shortest path** in the graph (ie, a path of even length). For this we need to build auxiliary graph, whose vertices are states Where - number of current peaks - The current parity. Any edge original graph in this new column will turn into two

ribs $((u, 0), (v, 1))$ and $((u, 1), (v, 0))$. After that, on this graph should preorder traversal to find the shortest path from the start vertex to the terminal, with parity equal to 0.

Problem in online judges

List of tasks that can be taken using a wide detour: • [SGU # 213](#)

"Strong Defence" [Difficulty: average]

Dfs

This is one of the basic graph algorithms.

As a result, depth-first search is lexicographically first path in the graph.

The algorithm works for $O(N + M)$.

Application of the algorithm

- Search any path in the graph.
- Search lexicographically first path in the graph.
- Checking whether a node of the tree another ancestor:
At the beginning and end of the iteration depth-first search will remember the "time" entry and exit at each vertex. Now for O(1) can find the answer: vertex i is an ancestor of node j if and only if start_i < start_j and end_i > end_j.
- [Problem LCA \(lowest common ancestor\)](#).
- [Topological Sort](#):
Run a series of depth-first search to traverse all vertices of the graph. Sort the vertices by time descending exit - this will be the answer.
- [Checking on the acyclic graph and finding cycle](#)
- [Search Strongly Connected Components](#):
Please do topological sort, then transpose graph and perform again a series of searches in depth in the order determined by the topological sorting. Each search tree - strongly connected component.
- [Search bridges](#):
First, convert the graph into a directed, making a series of searches in depth, and orienting each edge as we were trying to pass him. Then we find the strongly connected components. Bridges are those edges, the ends of which belong to different strongly connected components.

Implementation

```
vector<vector<int>> g; //  
Count int n; // Number of  
vertices  
vector<int> color; // Vertex numbers (0, 1, or 2)  
  
vector<int> time_in, time_out; // "Time" entry and exit from the top  
int dfs_timer = 0; // "Switch" to determine the time  
  
void dfs (int v) {  
    time_in[v] = dfs_timer +  
    ++; color[v] = 1;  
    for (vector<int>::iterator i = g[v].begin(); i != g[v].  
        end(); ++i) if (color[*i] == 0)  
        dfs (*i);  
    color[v] = 2;  
    time_out[v] = dfs_timer + +;  
}
```

This is the most common code. In many cases, the time of entry and exit from the top is not important, as well as the vertex colors are not important (but then you have to enter the same within the meaning of the boolean array used). Here are the most

simple implementation:

```
vector<vector<int>> g; //  
Count int n; // Number of  
vertices  
vector<char> used;  
  
void dfs (int v) {used  
    [v] = true;
```

```
for (vector<int> :: iterator i = g [v]. begin (); i! = g [v].  
    end (); + + i) if (! used [* i])  
        dfs (* i);  
}
```

Topological Sort

Given a directed graph with n and peaks m ribs. Required **renumber** its vertices so that each edge is led from the top with a lower number in the top with a lot.

In other words, find a permutation of the vertices (**topological order**) Corresponding to the order defined by all edges of the graph.

Topological sorting can be **not only** (Eg if the graph - empty, or if there are three such peaks a, b, c That of a there are ways in b and c , But neither of b in c Nor of c in b get impossible).

Topological sorting can **does not exist** at all - if the graph contains cycles (as there is a contradiction: there is a way and from one vertex to another, and vice versa).

Common task on a topological sort - Next. Yes n variables whose values are unknown to us. We only know about some of the pairs of variables that one variable is less than another. Need to check whether these are not contradictory inequality, and if not, give the variables in ascending order (if there are several solutions - to give any). Easy to see that this is exactly the problem of finding topological sorting in the graph of n vertices.

Algorithm

Solutions for use **traversal depth**.

Assume that the graph is acyclic, ie solution exists. What makes a detour into the depths? When you run out of some vertex v he tries to run along all edges emanating from v . Along the edges, the ends of which have already been visited before, it does not pass, and along all the others - and calls itself runs on their ends.

Thus, by the time the call $\text{dfs}(v)$ all vertices reachable from v both directly (by one edge) and indirectly (by the way) - all such vertices already visited bypass. Consequently, if we are at the moment out of our top add to the beginning of a list, then in the end of this list will **topological sorting**.

These explanations can be presented in a slightly different light, using the concept of "**Release date**" traversal depth.

Retention time for each vertex v - This is the point in time at which the call has finished work

$\text{dfs}(v)$ traversal depth of it (retention times can be numbered from 1 to n). Easy to understand that in a preorder traversal depth of the escape from some vertex v always greater than the output of all the vertices reachable from it

(Because they have been visited or to call $\text{dfs}(v)$ during it.) Thus, the desired topological sorting - sorting in descending order of retention times.

Implementation

Show implementation, which implies that the graph is acyclic, ie desired topological sort exists. If necessary check on the acyclic graph easily inserted into the bypass in depth, as described in [Article circumvention](#)

depth.

```
int n; // the number of
vertices vector <int>
G[MAXN]; // graph bool
used[MAXN];
vector <int> Ans;

void dfs (int v) {
    used[v] = true;
    for (size_t i = 0; i < g[v].Size(); ++
        I) { int to = g[v][i];
        if (!Used[to])
            dfs (to);
    }
    ans.push_back (v);
}

void topological_sort() {
    for (int i = 0; i < n; ++
        I) used[i] =
        false;
    ans.clear();
}
```

```
for (int i =0; i <n; +
    + I) if (!
    Used[i])
        dfs (i);
reverse (ans.begin(), Ans.end());
}
```

Here the constant **MAXN** should be set equal to the maximum possible number of vertices in the graph.

The main function of a solution - it **topological_sort**, it initializes tagging crawl deep, runs it, and the result is a response vector **ans**.

Problem in online judges

List of tasks that require topological sort search:

- [UVA # 10305 "Ordering Tasks"](#) [Difficulty: Easy]
- [UVA # 124 "Following Orders"](#) [Difficulty: Easy]
- [UVA # 200 "Rare Order"](#) [Difficulty: Easy]

Search algorithm of connected components in a graph

Given an undirected graph with n peaks and m ribs. You want to find in it all the connected components, ie, split vertices into several groups so that within a group can be reached from one vertex to any other, and between different groups - the path does not exist.

An algorithm for solving

Alternatively you can use as traversal depthAnd preorder traversal.

In fact, we will produce **series of detours**: Bypassing the first run of the first vertex and all vertices that while he walked - form the first connected component. Then find the first of the remaining vertices that have not yet been visited and run circumvention of it, thus finding a second connected component. And so on, until all the vertices will not be marked.

Summary **asymptotics** be: in fact, no such algorithm will run on the same vertex twice, and hence each edge is seen exactly twice (at one end and the other end).

Implementation

To implement a little more comfortable traversal depth:

```
int n;
vector<int>
G[MAXN]; bool
used[MAXN]; vector
<int> Comp;

void dfs (int v) {
    used[v] = true;
    comp.push_back (v);
    for (size_t i = 0; i < g[v].Size(); ++
        I) { int to = g[v][i];
        if (! used[to])
            dfs (to);
    }
}

void find_comps() {
    for (int i = 0; i < n; + +
        I) used[i] =
        false;
    for (int i = 0; i < n; + +
        I) if (!
        used[i]) {
            comp.clear ();
            dfs (i);
            cout << "Component:";
            for (size_t j = 0; j < comp.size(); +
                + J) cout << ' ' << Comp[j];
            cout << endl;
        }
}
```

The main function to call - `find_comps()`. It finds and displays the components of the graph.

We believe that given the graph adjacency lists, ie contains a list of vertices that have edges from the top $g[i]$

i. Constant **MAXN** should be set equal to the maximum possible number of vertices in the graph.

Vector **comp** contains a list of vertices in the current connected component.

Search strongly connected component, condensation build graph

Definition, problem statement

Given a directed graph whose vertex set and the set of edges - Loops and multiple edges are allowed. We denote n the number of vertices through m - The number of edges.

Strongly connected components (Strongly connected component) is called (Maximum by inclusion) subset of vertices such that any two vertices of this subset reachable from each other, ie for

where the symbol $u, v \in C$ hereinafter we denote reachability, ie existence of a path from the first vertex in the second: $u \rightarrow v, v \rightarrow u$

It is clear that for strongly connected components of the graph do not intersect, ie actually this partition all vertices of the graph. Hence the logical definition **condensation** as a graph resulting from this Count each contraction strongly connected components in a single vertex. Each vertex of condensation

corresponds to a strong component of the graph And a directed edge between two vertices Between which there was an edge graph condensation held if a pair of vertices in the original graph, ie

The most important property of the graph condensation is that it **acyclic**. Indeed, suppose that . Prove that . From the definition of condensation we get that there are two peaks and . That . Will prove the opposite, ie assume . Then there are two vertices $C \mapsto C'$ and $C' \neq C$. But an $u \in C$ and $v \in C'$. That since $u \rightarrow v$ and $v \in C$ are in the same strongly connected component, the $u' \in C$ and $v' \in C'$. As a result, combining the way, we find between them is the way; similarly for u that $u \rightarrow v$. Consequently, $u \rightarrow v$ must belong to the same strongly connected component, ie received contradiction, as required.

Algorithm described later in this column identifies all strongly connected components. Count on them to build condensation is not difficult.

Algorithm

Algorithm described here has been proposed independently Kosaraju (Kosaraju) and sarira (Sharir) in 1979 It is very easy to implement algorithm based on two series **dfs** And because working for time $O(n + m)$.

In a **first step** algorithm, a series of detours in depth attending the entire graph. For this We have passed through all the vertices of the graph and each vertex not yet visited call traversal depth. At

Moreover, for each vertex remember **Release time** $\text{tout}[v]$. These retention times play a key role algorithm, and this role is expressed in the following theorem.

First, we introduce the notation: time to $\text{tout}[C]$ of strongly connected components is defined as the maximum values of $\text{tout}[v]$ all $v \in C$. Moreover, in the proof will be referred to the time of entry and each vertex $\text{tin}[v]$ And similarly define the time of entry $\text{tin}[C]$ for each component of a strong connection of at least the quantities $\text{tin}[v]$ all $v \in C$.

Theorem. Let - Two different strongly connected components, and let the graph condensation between them an edge . Then

The proof there are two fundamentally different cases depending on which of the component first tour will go in depth, ie depending on the ratio between and :

• The first part has been achieved. This means that at some time in the crawl depth comes

in a vertex

components with all other components of the vertices

a
n
d
not yet been visited.
However, since

(C, C') $\text{tout}[C] > \text{tout}[C']$

$\text{tin}[C] < \text{tin}[C']$

by hypothesis, the graph has an edge

condensations

v C

but the whole

component

component tops

for each vertex

C

, From the top will be achieved not only the entire component

C

C'

• This means that

(C, C')

start from the top

in depth tour will pass through all

C

C'

and, and, therefore, they will be in relation to the descendants in the tree traversal in depth,

v

ie

will be performed

$\text{tout}[v] > \text{tout}[u]$, QED

- The first component was achieved C' . Again, at some time in the crawl depth comes in

some vertex $v \in C'$, And all the other vertices of the component C dC' are visited. Since by hypothesis condensations in the graph there rib (C, C') , Is due to condensation acyclic graph, not there is a way back $C' \not\rightarrow C$ ie bypassing the depth from the top v reaches heights. This means that they will be visited in the traversal depth later, whence $\text{tout}[C] > \text{tout}[C']$, QED

The above theorem is **basis for the algorithm** Search Strongly Connected Components. From this it follows condensations in the graph comes from components with greater a component that any edge (C, C') magnitude tout with smaller size.

If we sort all vertices $v \in V$ in descending order of release time $\text{tout}[v]$, It would be the first some vertex u , Owned by "root" strongly connected components, ie which excludes none of the edges in the graph condensations. Now we would like to start a tour of this vertex u . That would only visited this component strongly connected and has not gone to any other; learning how to do it, we can gradually select all strongly connected components: removing the first vertex of the graph of the selected component, we again find among the remaining vertex with the highest value tout . Again run out of it this tour, etc.

To learn how to do this tour, consider **transposed graph** ie graph obtained from G^T G changing the direction of each edge to the opposite. It is easy to understand that in this graph are those same strongly connected components as in the original graph. Moreover, the graph condensation for a will be equal transposed graph condensation of the original graph. This means that now of the considered "root" component will not be leaving the edges in other components.

So, to get around the whole "root" strongly connected components containing a vertex v enough to run from the top of bypassing v in the graph G^T . This tour will visit all the vertices of this component strong connectivity and only them. As already mentioned, then we can mentally remove these vertices of the graph, find the next vertex with the maximum value and run circumvention transposed graph therefrom, etc.

Thus we have constructed the following **algorithm** allocation Strongly Connected Components:

Step 1. Launch a series of detours in depth graph G which returns the top in order of increasing time

Release tout some list order .

Step 2. Build transposed graph G^T . Launch a series of detours in depth / width of the graph in order determined by list order (Namely, in the reverse order, ie in descending order of time exit). Each set of vertices reached as a result of the next crawling run and there will be another strongly connected components.

Asymptotics algorithm clear equal $O(n + m)$ Because it represents only two round trips depth / width.

Finally, it is appropriate to note the connection with the concept of **topological sorting**. First, step 1 algorithm is neither more nor less than a topological sort of the graph (in fact, this means sorting peaks retention time). Secondly, the chart is that the strongly connected components, and it generates in order to reduce their retention times, so it generates components - the vertices of condensation in topological sort order.

Implementation

```

vector<vector<int>> G, gr;
vector<char> Used; vector<int> Order, component;

void dfs1 (int v) {
    used[v] = true;
    for (size_t i = 0; i < g[v].Size(); +
        + I) if (!Used[g[v][i]])
        dfs1 (g[v][i]); order.push_back (v);
}

void dfs2 (int v) {
    used[v] = true;
    component.push_back (v);
    for (size_t i = 0; i < gr[v].Size(); +
        + I) if (!Used[gr[v][i]])
        dfs2 (gr[v][i]);
}

int main() {

```

```

int n;
Reading ... n ...
for (,;) {
    int a, b;
    Reading ... the next edge (a, b) ...
    g[a]. Push_back (b);
    gr[b]. Push_back
        (a);
}

used.assign (n,
false);for (int i =0; i
<n; + + I)
    if (! Used[i])
        dfs1 (i);
used.assign (n, false);
for (int i =0; i <n; + + I) {
    int v = order[n-1-I];
    if (! Used[v]) {
        dfs2 (v);
        Conclusion ... the next component ...
        component.clear ();
    }
}
}

```

Here is stored in the graph itself, and G^T . Function dfs1 - Transpose graph. Function dfs2 - By transposed G^T . Function dfs1 populates the list order increasing order of release time (in fact, makes a topological sort). Function dfs2 crawls in depth on a graph vertices preserves all reached the top of the list component , Which after each run will contain another strongly connected component.

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Construction and analysis [2005]**
- M. Sharir. **A strong-connectivity algorithm and its applications in data-flow analysis [1979]**

Search bridges

Suppose we are given an undirected graph. A bridge is an edge, whose removal makes the graph disconnected (or, rather, increases the number of connected components). You want to find all the bridges in a given graph.

Informally, this problem is formulated as follows: it is required to find the road map given all the "important" roads, ie such way that removing either of them will lead to the disappearance of the path between any pair of cities.

Below we describe an algorithm based on **DFS** And the running time $O(n + m)$ Wherein n - number of vertices m - Edges in the graph.

Note that the site is also described [online search algorithm bridges](#) - In contrast to the herein described algorithm, an online algorithm is able to maintain all the bridges in a changing graph graph (refers to adding new edges).

Algorithm

Run **traversal depth** from an arbitrary vertex of the graph; denoted by **root**. Note the following **fact** (Which is not hard to prove):

- Suppose we are bypassed in depth, looking now all edges from the top v . Then, if the current edge (v, to) such that the tops of to and any of its descendants in the tree traversal depth of no return edges to the top v or any of its ancestor, then this edge is a bridge. Otherwise, it is not a bridge.
(In fact, this condition we check if there are other ways of v in to But to descend along the edge (v, to)
Tree traversal in depth.)

It now remains to learn how to check this fact for each vertex effectively. For this we use
"Time of entry into the top", computed [depth-first search algorithm](#).

So let - this time call DFS at the top v . Now we introduce an array $fup[v]$ Which and allow us to answer the above questions. Time $fup[v]$ equal to the minimum time of entry into the very top $tin[v]$ Time of call in each vertex p That is the end of a reverse edge (v, p) And $fup[to]$ for each vertex to , Which is a direct son v in the search tree:

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all } (v, p) \text{ — back edge} \\ fup[to], & \text{for all } (v, to) \text{ — tree edge} \end{cases}$$

(Here "back edge" - the opposite edge, "tree edge" - edge of the tree)

Then, from the top v or her offspring have the opposite edge in its parent if and only if there is such a son to That $fup[to] \leq tin[v]$. (When $fup[to] = tin[v]$, It means that there is the opposite edge, coming exactly v ; if $fup[to] < tin[v]$, It means that there is an edge in the inverse ancestor vertices!)

Thus, if the current edge (v, to) (Belonging to the tree search) is performed $fup[to] > tin[v]$, then this edge is a bridge; otherwise it is not a bridge.

Implementation

If we talk about the actual implementation, here we must be able to distinguish three cases: when we go on the edge of the tree depth-first search, when we go to the opposite edge, and when trying to go along the edge of the tree in the opposite direction. It is, accordingly, the cases:

- Edges of the tree search criterion;
- $used[to] = false$ - Criterion reverse rib;
- $used[to] = true$ criterion of passage along the edge of the search tree in the opposite direction. $to = parent$

Thus, for the implementation of these criteria, we need to pass in the search function in the top of the depth of the current node ancestor.

```
const int MAXN = ...;
vector<int>
```

```
G [MAXN] ;bool  
used [MAXN] ;
```

```

int timer, tin[MAXN], Fup[MAXN];

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = Fup[v] = Timer++;
    for (size_t i = 0; i < g[v].Size(); ++i)
        if (to == p) continue;
        if (used[to])
            fup[v] = Min (fup[v], Tin[to]);
        else {
            dfs (to, v);
            fup[v] = Min (fup[v],
                           Fup[to]); if (fup[to] > Tin[v])
                IS_BRIDGE (v, to);
        }
    }
}

void find_bridges () {
    timer = 0;
    for (int i = 0; i < n; ++i)
        used[i] = false;
    for (int i = 0; i < n; ++i)
        if (!Used[i])
            dfs (i);
}

```

Here, the main function to call - it - it produces the necessary initialization and start crawling in depth for each connected component of the graph.

In this case its a function that will respond to the fact that the edge is a bridge, for example, show this to the screen edge.

Constanta MAXN in the beginning of the code should be set equal to the maximum possible number of vertices in the input graph.

It should be noted that this implementation does not work correctly in the presence of the column multiple edges. She actually does not pay attention, fold edge or whether it is unique. Of course, multiple edges should not be included in response, so calling IS_BRIDGE You can check further, not a multiple of whether we want to add an edge not top of the ancestor, and response. Another way - more accurate work with the ancestors, ie transfer to dfs the number ribs, which we entered the top (you will need to additionally store rooms all edges).

Problem in online judges

List of tasks that require search bridges:

- UVA # 796 "Critical Links" [Difficulty: Easy]
- UVA # 610 "Street Directions" [Difficulty: Medium]

Search cutpoints

Let a connected undirected graph. **Articulation point** (Or point of articulation, English. "Cut vertex" or "articulation point") is called a vertex, which makes removing the graph disconnected.

We describe an algorithm based on DFS, working for

$O(n + m)$ Wherein n - The number of vertices m

- Ribs.

Algorithm

Start the tour in depth from an arbitrary vertex of the graph; denoted by `root`. Note the following **fact** (Which is not hard to prove):

- Suppose we are bypassed in depth, looking now all edges from the top $v \neq \text{root}$. Then, if current edge (v, to) is that from the top and from any of its descendant tree traversal depth no inverse ribs or any ancestor vertices v , Then the vertex v is an articulation point. Otherwise, i.e. if bypass in depth through all the tops of the ribs v , And found satisfying the above conditions the edge, then the vertex v is not an articulation point. (In fact, this condition we check if there are other ways of v in to)
- We now consider the remaining case: $v = \text{root}$. Then this vertex is an articulation point if and only if this node has more than one son in the tree traversal in depth. (In fact, this means that the passing of the `root` for an arbitrary edge, we could not get around the whole graph, which implies that `root` - Point of articulation).

(Compare the wording with the wording of this criterion to criterion [search algorithm bridges](#).)

It now remains to learn how to check this fact for each vertex effectively. For this we use the "time of entry into the top", computed [depth-first search algorithm](#).

So, let - This time call DFS at the top

. Now we introduce an array

and allow us to answer the above questions. Time

Which equal to the minimum time of entry into the

top $tin[v]$, Time of call every vertex is the end of a reverse edge

$fup[v]$

Also all values of $fup[to]$ for each vertex to , Which is a direct son v in the search tree:

An

$(v, p)^d$

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all } (v, p) \text{ — back edge} \\ fup[to], & \text{for all } (v, to) \text{ — tree edge} \end{cases}$$

(Here "back edge" - the opposite edge, "tree edge" - edge of the tree)

Then, from the top v or her offspring have the opposite edge in its parent if and only if there is such a son to that $fup[to] < tin[v]$.

Thus, if the current edge (v, to) (Belonging to the tree search) is performed $fup[to] \geq tin[v]$, then the vertex is an articulation point. For the initial vertex $v = \text{root}$ another criterion: for this vertex need to count the number of immediate sons in the tree traversal in depth.

Implementation

If we talk about the actual implementation, here we must be able to distinguish three cases: when we go on the edge of the tree depth-first search, when we go to the opposite edge, and when trying to go along the edge of the tree in the opposite direction. It is accordingly the cases `used[to] > used[v]` and `to = parent`. Thus, we need to pass in the search function in the top of the depth of the current node ancestor.

```
vector<int>
G[MAXN]; bool
used[MAXN];
int timer, tin[MAXN], Fup[MAXN];

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = Fup[v] = Timer +
   ++; int children = 0;
```

```
for (size_t i = 0; i < g[v].Size(); ++  
    i) { int to = g[v][i];
```

```

        if (to == p) continue;
        if (used[to])
            fup[v] = Min (fup[v], Tin[to]);
        else {
            dfs (to, v);
            fup[v] = Min (fup[v], Fup[to]); if
                (fup[to] >= Tin[v] && P! = -1)
                    IS_CUTPOINT (v);
            + + Children;
        }
    }
    if (p == -1 && Children> 1)
        IS_CUTPOINT (v);
}

int main () {
    int n;
    Reading ... n and g ...

    timer = 0;
    for (int i =0; i <n; + +
        I) used[i] =
            false;
    dfs (0);
}

```

Here the constant **MAXN** must be set equal to the maximum possible number of vertices in the input graph.

Function **IS_CUTPOINT(v)** code - is a function that is responsive to the fact that the top v is an articulation point, for example, to display this on the top of the screen (it should be borne in mind that for the same vertex, this function can be called multiple times.)

Problem in online judges

Task List, which requires seek points of articulation:

- UVA # 10199 "Tourist Guide" [Difficulty: Easy]
- UVA # 315 "Network" [Difficulty: Easy]

Search bridges online

Suppose we are given an undirected graph. A bridge is an edge, whose removal makes the graph disconnected (or, rather, increases the number of connected components). You want to find all the bridges in a given graph.

Informally, this problem is formulated as follows: it is required to find the road map given all the "important" roads, ie such way that removing either of them will lead to the disappearance of the path between any pair of cities.

Algorithm described here is **online**. That means that the count was not input in advance is known and whose edges are added to it one by one, and after each addition of the algorithm recalculates all bridges in the current column. In other words, the algorithm is designed to work effectively in a dynamic, changing graph.

More precisely, **formulation of the problem**: Next. Initially empty graph and consists of n vertices. Then receives requests, each of which - a pair of vertices that represent an edge is added to the graph. Required after each request, i.e. after the addition of each edge, display the current number of bridges in the graph. (You may want to keep a list of all edges, bridges, as well as explicit support rib doubly connected components.)

Algorithm described below works in time where $O(m \log n + mn)$. The number of requests. The algorithm is based on **data structure "A system of disjoint sets"**.

Present implementation of the algorithm, however, works in time because it uses a simplified version of one place **system of disjoint sets** rank without heuristics.

Algorithm

It is known that edge-bridges divide vertices into components called costal doubly connected components. If each component costal doubly connected to squeeze into a single vertex, and leave only the edges of bridges between these components, you get an acyclic graph, ie forest.

Algorithm described below support this explicitly **forest component costal doubly connected**.

It is clear that initially, when the graph is empty, it contains n costal doubly connected component unrelated to each other in any way.

When you add the next edge (a, b) three situations may occur:

- Both ends a and b are in the same component costal doubly connected - then this edge is not a bridge, and does not alter the structure of the forest, so just skip this edge.

Thus, in this case, is not changed by the bridges.

- Vertices a and b are in different connected components, ie connect two trees. In this case, the rib (a, b) becomes the new bridge, and the two trees are merged into one (and all the old bridges remain). Thus, in this case, the number of bridges is incremented.

- Vertices a and b are in the same component, but in different components of the costal doubly connected. In this case, the rib forms a ring together with some of the old bridges. All of these bridges are no longer bridges, and the resulting cycle must be combined into a new component costal doubly connected.

Thus, in this case, the number of bridges is reduced by two or more.

Consequently, the whole problem is reduced to the effective implementation of all these operations over the forest component.

Data structure for timber

All we need from the data structures - is **system of disjoint sets**. In fact, we need to make two copies of this structure: one is to maintain **component connectivity**, other - to maintain **doubly connected component costal**.

In addition, the storage structure of trees in the forest doubly connected component for each vertex we store the pointer `par[]` its parent in the tree.

We now analyze each operation sequentially, which we must learn to realize:

- Checking whether the two are listed in the top of the same component / doubly connected.**

Typically, a query is made to the structure "A system of disjoint sets."

- Joining two trees in one** for some edge (a, b) . As it could happen that no vertex a or vertex b are not roots of their trees, the only way to connect these two trees

-**perepodvesit** one of them. For example, you can perepodvesit single tree on the vertex a And then attach it to another tree, making the top a subsidiary to b .

However, there is a question about the effectiveness of the operation perepodvesit: to perepodvesit tree rooted at r in the opposite direction, as well as changing at the top, you have to pass on the way in from redirecting pointers $\text{par}[r]$ the reference ancestor on the system of disjoint sets responsible for the connected components.

Thus, the cost of the operation has perepodveshivaniya $O(h)$ - Height of the tree. You can evaluate it even higher, saying that this is the value $O(\text{size})$ Wherein size - The number of vertices in the tree.

We now apply a standard method: we say that two trees perepodveshivat will then wherein the lower vertices. Then intuitively, that the worst case - when you combine two tree of approximately equal size, but then the result is twice the size tree that does not allow such a situation to occur many times. Formally, this can be written as the recurrence relation:

where by $T(n)$ we denote the number of operations required to obtain wood from perepodveshivaniya operations and merging trees. This is a known recurrence, and it has decision $T(n) = O(n \log n)$.

Thus, the total time spent on all perepodveshivaniya, make $O(n \log n)$ If we will always be the lesser of two perepodveshivat tree.

We have to maintain the size of each connected component, but the data structure "system disjoint sets" lets you do this easily.

.**Find a cycle** Formed by adding a new edge (a, b) in a tree. Practically, this means that we need to find the lowest common ancestor (LCA) of vertices a and b .

Note that then we compress all of the vertices in one cycle of the detected peak, so we want any of the search algorithm LCA, the running time of the order of its length.

Since all the information about the structure of the tree that we have - this links $\text{par}[]$ on the ancestors, the seems the only possible next search algorithm LCA: mark the top and as a b visited, then go to their ancestors $\text{par}[a]$ and $\text{par}[b]$ and mark them, then to their ancestors, and so on, until does not happen, that at least one of the two current peaks is already marked. This will mean that the current peak - is the desired LCA, and it will be necessary to repeat the path again to her from the top a and from the top - thus we find the desired cycle.

Obviously, this algorithm works in time order of the length of the desired cycle, since each of the two pointers could not pass a distance greater than this length.

. **Compression cycle**, formed by adding a new edge (a, b) in a tree.

We want to create a new component costal doubly connected, which will consist of all the vertices of the detected cycle (of course, found himself cycle could consist of some doubly connected component, but it does not change anything).

Furthermore, it is necessary to perform compression in such a manner not to disturb the structure of wood, and all pointers $\text{par}[]$ system and two disjoint sets are correct.

The easiest way to achieve this - **compress all the peaks found in their cycle LCA**. In fact, the top-LCA - is the highest peaks of the compressible, ie its par remains unchanged. For all other vertices compressible update also do not need anything, because these peaks simply cease to exist - in the system of disjoint sets for doubly connected component of all these vertices will just point to the top-LCA.

But then it turns out that the system of disjoint sets for doubly connected component operates without heuristics union rank: if we always attach them to the top of the cycle LCA, this is no place heuristics.

In this case arises in the asymptotic $O(\log n)$ Because without heuristics rank with any operation disjoint sets system works precisely for such a time.

To achieve the asymptotics $O(1)$ one request is necessary to combine cycle top according to rank heuristics, and then assign par new leader $\text{par}[\text{LCA}]$.

Implementation

We present here the final implementation of the whole algorithm.

For simplicity, a system of disjoint sets for doubly connected component is written **without rank heuristics**, therefore the total amount on the asymptotic behavior $O(\log n)$ of the average request. (The fact how to achieve the asymptotics $O(1)$, Described above in paragraph "compression cycle".)

Also in this embodiment, the ribs themselves are not stored bridges, and kept only their number - see variable

bridges. However, if desired, will have no difficulty to start set All of the bridges.

Initially, you must call the function that initializes the two systems of disjoint sets (assigning each vertex in a separate set, and affixing a size equal to one), marks the ancestors par.

The main function - it add_edge(a, b) That handles the request to add a new edge.

Constanta MAXN should be set equal to the maximum possible number of vertices in the input graph.

More detailed explanations to this implementation, see below.

```
const int MAXN = ...;

int n, bridges, par[MAXN], Bl[MAXN], Comp[MAXN], Size[MAXN];

void init() {
    for (int i = 0; i < n; ++i)
        { bl[i] = Comp[i] = i; size[i] = 1;
          par[i] = -1;
        }
    bridges = 0;
}

int get (int v) {
    if (v == -1) return -1;
    return bl[v]== v? v: bl[v]= Get(bl[v]);
}

int get_comp (int v) {
    v = get(v);
    return comp[v]== v? v: comp[v]= Get_comp(comp[v]);
}

void make_root (int v) {
    v = get(v);
    int root = v,
        child = -1;
    while (v!= -1) {
        int p = get(par[v]);
        par[v] = Child;
        comp[v] = Root;
        child = v; v = p;
    }
    size[root] = Size[child];
}

int cu, u[MAXN];

void merge_path (int a, int b) {
    ++Cu;
    vector <int> Va,
    vb; int lca = -1;
    for(;;) {
        if (a!= -1) {
            a = get(a);
            va.pb (a);
            if (u[a] == Cu) {
                lca = a;
                break;
            }
            u[a] = Cu;
            a = par[a];
        }
    }
}
```

```

        if (b != -1) {
            b = get(b);
            vb.pb(b);
            if (u[b] == Cu) {
                lca = b;
                break;
            }
            u[b] = Cu;
        }
        b = par[b];
    }
}

for (size_t i = 0; i < va.size(); ++I) {
    bl[va[i]] = Lca;
    if (va[i] == Lca) break;
    Bridges++;
}
for (size_t i = 0; i < vb.size(); ++I) {
    bl[vb[i]] = Lca;
    if (vb[i] == Lca) break;
    Bridges++;
}
}

void add_edge (int a, int b) {
    a = get(a); b = get(b);
    if (a == b) return;
    int ca = get_comp(a),
        cb = get_comp(b); if (ca != cb)
    {
        ++Bridges;
        if (size[ca] > Size[cb]) {
            swap(a, b);
            swap(ca, cb);
        }
        make_root(a);
        par[a] = Comp[a] = B;
        size[cb] += Size[a];
    }
    else
        merge_path(a, b);
}

```

Comment on the code in more detail.

System of disjoint sets for doubly connected component stored array, and a function that returns the leader doubly connected components - that. This function $\text{get}(v)$ used many times in the rest of the code, as you have to remember that after compress multiple vertices in one all these vertices cease to exist and instead there is only their leader, which are stored and the correct data (ancestor par , Ancestor of the system of disjoint sets for the connection components, etc.).

System for disjoint sets of connected components stored array, there is also an additional array storage component sizes. $\text{size}[]$

Function $\text{get_comp}(v)$ returns leader connected components (which is actually the root of the tree).

Function perevodshivaniya tree $\text{make_root}(v)$ operates as described above: it is from the top v ancestor to the root, each time redirecting ancestor par in the opposite direction (downward direction to the top v). Also updated pointer comp disjoint sets in the system for the components connected to point to the new root. After the new root perevodshivaniya DIMENSIONS

size connected components. Note that the implementation every time we call the function $\text{get}()$, To gain access to the leader is strongly connected components, but not to any other vertex, which may have been compressed.

The detection and compression path $\text{merge_path}(a, b)$ As described above, seeks LCA vertices a and b what rises up from them in parallel until some vertex is encountered for the second time. To be effective, passed vertices labeled by the technique of "numerical used", what works for

$O(1)$ Instead of applying `set`. Completed path is stored in the vectors `va` and `vb` Then to work through them

second time to LCA, thereby obtaining all the vertices of the cycle. All vertices are compressed by joining them to LCA (asymptotic behavior arises here) because when we do not use compression, the rank heuristic). Along the way, considered the number of edges traversed, which is the number of bridges in the detected cycle (this the amount deducted from the **bridges**).

Finally, **function query** $\text{add_edge}(a, b)$ determines the connected components in whose vertices lie a and, and if they lie in different connected components, the minimal tree $\text{perepodveshivaetsya}$ for new root and then attached to a large tree. Otherwise, if the vertices a and belong to the same tree, but in different components of doubly connected, the function is called $\text{merge_path}(a, b)$ which detects the cycle, and compresses it into a single component doubly connected.

Finding the shortest paths from a given vertex to all other vertices Dijkstra algorithm

Statement of the Problem

Given a directed or undirected weighted graph with n nodes and m edges. Weights of all edges are non-negative.

Contains some starting vertex s . Required to find the length of the shortest paths from vertex s to all other vertices, as well as provide a way to output the shortest paths themselves.

This problem is called "the problem of the shortest paths from a single source" (single-source shortest paths problem).

Algorithm

Here we describe an algorithm, which was offered by a Danish researcher **Dijkstra** (Dijkstra) in 1959

Head Array $d[]$ In which each vertex will store the current length $d[v]$ the shortest path from s to v . Initially $d[s] = 0$ And for all other vertices, this length is equal to infinity (when implemented on computer usually as infinity just choose sufficiently large number, certainly more possible path length):

$$d[v] = \infty, v \neq s$$

Furthermore, for each vertex v will be stored, labeled it yet or not, ie 's have a boolean array $u[]$. Initially, all vertices are labeled, ie

$$u[v] = \text{false}$$

Dijkstra algorithm itself consists of n iterations. On the next iteration selected vertex v with the lowest value $d[v]$ of not yet labeled, ie:

(It is understood that on the first iteration will be selected starting vertex.)

Selected so the top notes marked. Further, in the current iteration of the vertices v produced relaxation: Browse all edges (v, to) emanating from the vertex, and for each such a vertex to the algorithm tries to improve the value $d[to]$. Let the length of this edge is equal len . Then relaxation as a code looks like:

$$d[to] = \min(d[to], d[v] + \text{len})$$

At the end of the current iteration, the algorithm proceeds to the next iteration (again selected vertex with the smallest value, the relaxation produced therefrom, etc.). In the end, after n iterations, all vertices will be labeled, and the algorithm completes its work. It is argued that the values found $d[v]$ is the required length of the shortest paths from s to v .

It is worth noting that if not all the vertices reachable from the top s , The value $d[v]$ for them and remain endless. It is clear that the last few iterations of the algorithm will just choose these peaks, but no useful work to produce these iterations will not (because an infinite distance can not prorelaksirovat others, even too infinite distance). Therefore, the algorithm can be immediately stopped as soon as the selected vertices taken top with infinite distance.

Restoration paths. Of course, usually need to know not only the length of the shortest paths but also to obtain own way. We show how to preserve enough information to restore the shortest path from

s to any vertex. It's enough to so-called **array ancestors**: Array $p[]$ Wherein for each vertex $v \neq s$ node number is stored $p[v]$, Which is the penultimate in the fast track to the top v . Here we use the fact that if we take the shortest path to some vertex v , And then remove from last vertex of this path, we get the path ending with a vertex $p[v]$ And this way is the shortest for the top $p[v]$. So if we have this array of ancestors, the shortest path can be restored to him, just every time taking the ancestor of the current node, until we come to the starting vertex s - So we obtain the desired shortcut, but written in reverse order. Thus,

shortcut P to the top v is equal to:

$$P = (s, \dots, p[p[p[v]]], p[p[v]], p[v], v)$$

Necessary to understand how to build the array ancestors. However, this is very simple: at every successful relaxation, ie when the selected vertices v an improvement of the distance to a vertex to We write that ancestor vertices to is the peak v :

$$p[to] = v$$

Proof

The main assertion, based on which the correctness of Dijkstra's algorithm, following. Allegedly that after some vertex v becomes marked, the current distance to it already $d[v]$ is the shortest, and therefore more will not change.

Proof will produce by induction. For the first iteration of the justice of his obvious - for the top s have, which is the length of the shortest path to it. Suppose now that statement holds for all previous iterations, ie all already labeled vertices; prove that it is not disturbed after the current iteration.

Let v - Vertex selected in the current iteration, ie

vertex, which is going to mark the algorithm. We prove that $d[v]$ indeed equal to the length of the shortest path to the She (we denote this length by $l[v]$).

Consider the shortest path P to the top. Clearly, this path can be divided into two paths: consisting only of labeled vertices (at least the starting vertex is in the way) and the rest of the way P_1 (She also may include marked the top, but certainly begins with untagged). We denote P_2 first top path and q through - the last point of the path. P_1 . However,

We first prove the assertion for the top, ie prove the equality $d[p] = l[p]$ this almost clear: in fact on one of the previous iterations, we chose the top p and perform the relaxation of it. Since (by virtue of the choice of the vertex p) Shortest path to p is the shortest path to q plus edge (p, q) . Then if the relaxation of q value $d[p]$ really set to the desired value.

Due to the non-negativity values of edges of the shortest path length $l[p]$ (And she just proved) Does not exceed the equal $d[p]$ length of $l[v]$ the shortest path to the top. Given that $l[v] \leq d[v]$ (After Dijkstra's algorithm could not find a shorter route than it is at all possible), we finally obtain the relation:

$$d[p] = l[p] \leq l[v] \leq d[v]$$

On the other hand, since both p And v - Unmarked vertex, then, since in the current iteration been chosen vertex v And not the top p , Then we get another inequality:

$$d[p] \geq d[v]$$

From these two inequalities we conclude equality $d[p] = d[v]$ And then found out before relations and obtain:

$$d[v] = l[v]$$

QED.

Implementation

Thus, Dijkstra's algorithm is n iterations, each of which is selected unlabeled vertex with the lowest value, the vertex is marked, and then iterates through all the edges emanating from $d[v]$

Current top and along each edge is an attempt to improve the value $d[v]$ the other end of the rib. Time of the algorithm consists of:

- n Just search the top with the lowest value among all unmarked vertices, ie of vertices $O(n)$
- m times tries relaxations

At the simplest implementation of these operations on the top search operations $O(n)$ will be spent, and

One relaxation - $O(1)$ operations, and the final **asymptotics** algorithm is:

$$O(n^2 + m)$$

Implementatio n:



```

const int INF = 1000000000;

int main() {
    int n;
    Reading ... n ...
    vector <vector <pair <int,int>>> G (n);
    Graph reading ... ...
    int s = ...; / / top of page

    vector <int> D (n, INF), P (n);
    d[s] = 0;
    vector <char> U (n);
    for (int i =0; i <n; + +
        I) { int v = -1;
        for (int j =0; j <n; + + J)
            if (! U[j] && (v == -1 || D[j]
                <D[v])) v = j;
        if (d[v] == INF)
            break;
        u[v] = true;
        for (size_t j =0; j <g[v]. Size(); + +
            J) { int to = g[v] [j]. First,
                len = g[v] [j]. Second; if (d[v] + Len
                <d[to]) {
                    d[to] = D[v] + Len;
                    p[to] = v;
                }
            }
        }
    }
}

```

Here the graph is stored in the form of adjacency lists: for each vertex v list $g[v]$ contains a list of edges, originating from this node, i.e. a list of pairs $\langle \text{weight}, \text{int} \rangle$, which an edge, and the second element - the weight of the edge.

After reading infest arrays distances $d[]$, Labels $p[]$ and ancestors $p[]$. Then iterates. On n each iteration of the first peak is v . Having the smallest distance $d[v]$ of unlabelled vertices. If the distance to the selected vertex v is equal to infinity, then the algorithm stops. Otherwise, the vertex is marked as tagged, and scans all edges emanating from a given vertex, and run along each edge relaxation. If relaxation is successful (ie, the distance $d[to]$ changes) then recalculated distance $d[to]$ and stored ancestor $p[to]$.

After all the iterations in the array $d[]$ are the length of the shortest paths to all vertices in the array and $p[]$ - The ancestors of all vertices (except the starting s). Restore the path to any vertex t as follows:

```

vector <int> Path;
for (int v = t; v != s; v =
    p[v])
    path.push_back (v);
path.push_back (s);
reverse (path.begin(), Path.end());

```

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Construction and analysis** [2005]
- Edsger Dijkstra. **A note on two problems in connexion with graphs** [1959]

Finding the shortest paths from a given vertex to all other vertices of Dijkstra algorithm for sparse graphs

Formulation of the problem, the algorithm and its proof, see [article about the general Dijkstra](#).

Algorithm

Recall that the complexity of Dijkstra's algorithm consists of two basic operations: Time Spent tops with the lowest value of the distance $d[v]$ And the time of relaxation, ie time change of the $d[to]$.

At the simplest implementation, these operations require correspondingly and time. Given that first operation just performed times, and the second - We obtain the asymptotic behavior of the implementation of Dijkstra's algorithm: $O(n)$ $O(m)$ $O(1)$ $O(n^2 + m)$.

It is clear that this asymptotic behavior is optimal for dense graphs, ie when graph (i.e., less than m compared to the maximum number of edges n^2). The less becomes the optimal this estimate, and the fault of the first term. Thus, it is necessary to improve the operating times of the first type is not greatly deteriorating the run-time operations of the second type.

To do this, use a variety of auxiliary data structures. The most attractive are **Fibonacci heap** Which allow for the operation of the first kind second - for $O(1)$. Therefore, when using Fibonacci heaps time Dijkstra algorithm be $O(n \log n + m)$ That is almost the theoretical minimum for the search algorithm the shortest path. Incidentally, this estimate is optimal for algorithms based on Dijkstra's algorithm, ie Fibonacci heaps are optimal from this point of view (this is an optimality actually based on the impossibility of the existence of such an "ideal" data structure - if it existed, it would be possible to sort in linear time, which is known in the general case is impossible; however, it is interesting that there is an algorithm Thorup (Thorup), who is looking for the shortest path to the optimal linear, asymptotic behavior, but it is based on a very different idea than Dijkstra's algorithm, so no contradiction here.) However, Fibonacci heaps are quite complicated to implement (and, it should be noted, have considerable constant hidden in the asymptotic behavior).

As a compromise, you can use the data structure to allow you to **Both types operations** (In fact, this extract and update the minimum element) for Dijkstra's algorithm will be:

As such data structures programmers in C ++ is convenient to take a standard container or **set**. The first is based on the red-black tree, the second - on the binary heap. So **priority_queue** has a lower constant hidden in the asymptotic behavior, but it has the disadvantage that it not support the deletion element, because of what has to do "workaround" which actually leads to the substitution in the asymptotic $\log n$ on $\log m$ (In terms of the asymptotic behavior is actually nothing does not change, but the hidden constant increases).

Implementation

set

Let's start with the container **set**. Because we need to keep the container top, ordered according to their values, it is convenient to place the container in pairs: the first element of the pair - the distance, and the second - the number of vertices. As a result,

A **set** pair will be stored automatically sorted by the distance that we need.

```
const int INF = 1000000000;

int main() {
    int n;
    Reading ... n ...
}
```

```

vector <vector <pair <int,int>>> G (n);
Graph reading ... ...
int s = ...; / / top of page

vector <int> D (n, INF), P (n);
d[s] = 0;
set <pair <int,int>> Q;
q.insert (make_pair (d[s],
S)); while (! Q.empty()) {
    int v = q.begin() ->
Second; q.erase
(q.begin());
for (size_t j =0; j <g[v]. Size(); + +
J) { int to = g[v] [j]. First,
len = g[v] [j].
Second; if (d[v] + Len
<d[to]) {
    q.erase (make_pair (d[to], To));
    d[to] = D[v] + Len;
    p[to] = v;
    q.insert (make_pair (d[to], To));
}
}
}
}

```

Unlike conventional Dijkstra algorithm becomes unnecessary array. Its role as a function finding vertices with the smallest distance, performs `set`. Initially, he put the starting vertex `s` with its distance. The main loop of the algorithm is executed in a queue until there is at least one vertex. Is retrieved from the queue with the smallest distance to the vertex, and then executed from its relaxation. Before performing each successful relaxation we first remove from `set` old pair and then, after the relaxation back to attach a new pair (new Proximity $d[to]$).

priority_queue

Fundamentally different from here `set` No, except for the moment that any elements removed from the impossible `priority_queue` (although theoretically heap support such an operation, in the standard library is not implemented). Therefore it is necessary to make "workaround": the relaxation simply will not remove the old pair from the queue. As a result, the queue can be simultaneously several pairs of the same vertices (but at different distances). Among these pairs we are interested in only one, for which the element `first` equal, and all others are bogus. So

necessary to make a slight modification: the beginning of each iteration, when we learn from the next couple of turns, will check fictitious or not (it is enough to compare `first` u). It should be noted that

an important modification: if not make it, it will lead to a significant deterioration of the asymptotics (up $O(nm)$).

Yet it must be remembered that `priority_queue` arranges elements in descending order and not ascending, as usual. The easiest way to overcome this feature is not an indication of its comparison operator, but simply as an element of putting distance with a minus sign. As a result, the root elements of the heap will be provided

`first`

the smallest distance that we need.

```

const int INF = 1000000000;

int main() {
    int n;
    Reading ... n ...
    vector <vector <pair <int,int>>> G (n);
    Graph reading ... ...
    int s = ...; / / top of page

    vector <int> D (n, INF), P (n);
    d[s] = 0;
    priority_queue <pair <int,int>> Q;
    q.push (make_pair (0, S));
    while (! Q.empty()) {
        int v = q.top(). Second, cur_d ==q.top(). First;
        q.pop();
        if (cur_d > d[v]) continue;

        for (size_t j =0; j <g[v]. Size(); + +
J) { int to = g[v] [j]. First,

```

```

        len = g[v][j].
Second; if (d[v] + Len
<d[to]) {
    d[to] = D[v] + Len;
    p[to] = v;
    q.push (make_pair (-D[to], To));
}
}
}

```

As a rule, in practice version `priority_queue` is slightly faster version `set`.

Getting rid of pair

You can even slightly improve performance if in containers not yet store the pair, and only the numbers of vertices. In this case, it is clear, it is necessary to overload the comparison operator for the vertices: compare two nodes need to distances of `d[]`.

As a result of the relaxation value of the distance to the top of any changes, it should be understood that "by itself" data structure is not reconstructed. Therefore, although it may seem that the delete / add elements to the container in the relaxation process is not necessary, it will lead to the destruction of the data structure. Still before the relaxation should be removed from the top of the data structure `to`. And after relaxation insert it back - then no relations between the elements of the data structure are not violated.

And since you can delete items from `set`, but not from `priority_queue`. It turns out that this method is only applicable to `set`. In practice, it significantly increases the performance, especially when the distances are used to store large data types (such as `long long` or `double`).

Bellman-Ford algorithm

Let a directed weighted graph with n peaks and m ribs, and contains a vertex

v . Required to find the length of the shortest paths from the top v to all other vertices.

Unlike Dijkstra's algorithm This algorithm can also be applied to graphs containing negative edge

weight. However, if the graph contains a negative cycle, then, of course, the shortest path to some of the vertices may not exist (due to the fact that the weight of the shortest path must be equal to infinity); however, this algorithm can be modified to signal the presence of the negative cycle of the weight, or even bringing the cycle itself.

The algorithm is named after two American scientists: Richard **Bellman** (Richard Bellman) and Leicester **Ford** (Lester Ford). Ford actually invented this algorithm in 1956 in the study of other mathematical problem, which has been reduced to the subproblem of finding the shortest path in the graph, and Ford gave a sketch of the algorithm to solve this problem. Bellman in 1958 published an article devoted specifically to the problem of finding the shortest path, and in this article he clearly formulated the algorithm in the form in which we know it now.

Description of the algorithm

We believe that the graph does not contain a cycle of negative weight. Case of having a negative cycle will be discussed below in a separate section.

Head array distances $d[0 \dots n - 1]$ That after execution of the algorithm will contain the answer to the problem.

In early work, we fill it as follows: $d[v] = 0$ And all other elements $d[]$ equal infinity. ∞

The algorithm of Bellman-Ford is a few phases. Each phase scans all edges

graph, and the algorithm tries to **relaxation** (Relax, attenuation) along each edge (a, b) cost c . Relaxation along the edge - is an attempt to improve the value $d[b]$ value $d[a] + c$. In fact, this means that we try to improve response to the top, using an edge (a, b) and the current response for the top.

It is argued that it is sufficient $n - 1$ phase of the algorithm to correctly calculate the lengths of the shortest paths in the graph (again, we believe that no cycles of negative weight). For inaccessible peaks distance $d[]$ will be equal to infinity ∞ .

Implementation

Algorithm Bellman-Ford, unlike many other graph algorithms, more convenient to represent the graph as a single list of all the edges (and not the n lists of edges - the edges of each vertex). In here Starts implement data structure for an edge. The input data for the algorithm are the number n, m List e ribs and the room starting vertex v . All the numbers of vertices are numbered 0 by $n - 1$.

The simplest implementation

Constant **INF** denotes the number of "infinity" - it should be chosen so as to be clearly superior to all of the possible path lengths.

```
struct edge {
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve() {
    vector<int> D(n, INF);
    D[v] = 0;
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < m; ++j)
            if (D[e[j].A] < INF)
                D[e[j].B] = Min(D[e[j].B], D[e[j].A] + E);
}
```

[j]. Cost);

```
// O d, for example, to screen
```

Check "if ($d[e[j]. A] < INF$)" is needed only if the graph contains edges of negative weight: without such checks would place the relaxation of the vertices to which the path has not yet been found, and would appear incorrect distance from $\infty - 1, \infty - 2$ etc.

An improved

This algorithm can speed up a few: often the answer is already in several phases, and the remaining phases of any useful work does not happen, only wasted scans all edges. Therefore, we will keep the flag that something has changed in the current phase or not, and if at some stage, nothing happened, then the algorithm can be stopped. (This optimization does not improve the asymptotic behavior, ie, on some graphs will continue to need all $n - 1$ phase, but significantly accelerates the behavior of the algorithm "on average", ie on random graphs.)

With this enhancement becomes generally unnecessary to manually limit the number of phases of the algorithm number $n - 1$ - He stops after the desired number of phases.

```
void solve() {
    vector<int> D (n,
    INF); d[v] = 0;
    for (, ;) {
        bool any = false;
        for (int j = 0; j < m; ++ j)
            if (d[e[j]. A] < INF)
                if (d[e[j]. B] > D[e[j]. A] + E[j]. Cost) {
                    d[e[j]. B] = D[e[j]. A] + E[j].
                    Cost; any = true;
                }
        if (! Any) break;
    }
    // O d, for example, to screen
}
```

Restoration paths

Let us now consider how we can modify the algorithm of Bellman-Ford so that he not only found the length of the shortest paths, but also allows you to restore themselves shortest paths.

For this array's have another $p[0 \dots n - 1]$ in which each vertex will store its "ancestor", ie is the penultimate vertex in the shortest path leading to it. In fact, the shortest path to some vertex

a is the shortest path to some vertex $p[a]$, Which attributed to the end of the summit a .

Note that the algorithm of Bellman-Ford is working on the same logic: it is, assuming that the shortest distance to a vertex already counted, trying to improve the shortest distance to another vertex. Consequently, at the moment we just need to improve to remember in $p[]$, Of which vertex is the improvement occurred.

We present the implementation of the Bellman-Ford with the restoration of some ways to a given node t :

```
void solve() {
    vector<int> D (n,
    INF); d[v] = 0;
    vector<int> P (n, -1);
    for (, ;) {
        bool any = false;
        for (int j = 0; j < m; ++ j)
            if (d[e[j]. A] < INF)
                if (d[e[j]. B] > D[e[j]. A] + E[j]. Cost)
                    { d[e[j]. B] = D[e[j]. A] + E[j].
                    Cost;
                     p[e[j]. B] = E[j].
                     A; any = true;
                    }
        if (! Any) break;
    }

    if (d[t] == INF)
        cout << "No path from" << v << "To" << t << ".";
    else {
        vector<int> Path;
```

```

        for (int cur = t; cur != -1; cur =
            p[cur]) path.push_back
            (cur);
        reverse (path.begin(), Path.end());
    }

}

```

Here we first passed by the ancestors, starting from the top, and keep all the traversed path in the list

This list is obtained from the shortest path to t , but in reverse order, so we call it **path**.
reverse him

Proof of the algorithm

First, once we note that for inaccessible from v vertices algorithm will work correctly: they label as $d[]$ remain equal to infinity (since Bellman-Ford algorithm to find some way to all of the attainable s vertices and relaxation at all other vertices will not happen even once).

We now prove the following **approval**: After i phase-Bellman Ford algorithm correctly finds all shortest path whose length (number of edges) do not exceed i .

In other words, for every vertex a denote the number of edges in the shortest path to it (if there are several ways you can take any). Then this statement says that after this phase the shortest path will be found guaranteed.

Proof. Consider an arbitrary vertex a to which there is a path from the starting vertex v ,

Before the first phase of the shortest path and consider the shortest path to it ($p_0 = v, p_1, \dots, p_k = a$) to the top $p_0 = v$ found correctly. During the first phase of an edge (p_0, p_1) algorithm has been viewed Ford-Bellman equation, so that the distance to the top p_1 was correctly counted after the first phase. Repeating these statements again, we see that after the second phase of the distance to the top $p_k = a$ counted correctly, as required.

The last thing to note is that any shortest path can not have more $n - 1$ ribs. Consequently, the algorithm is sufficient to make only $n - 1$ phase. After that, no relaxation guaranteed improvement can not be completed until some vertex distance.

Case of a negative cycle

Above all, we felt that a negative cycle in the graph does not contain (more precise, we are interested in a negative cycle reachable from the starting vertex v). And unreachable cycles nothing in the above algorithm does not change).

If present, there are additional complexities associated with the fact that the distances to all the peaks in the same cycle as well as the distance to the reachable vertices of this cycle is not defined - they should be equal to minus infinity.

It is easy to understand that the algorithm Ford-Bellman equation can **infinitely do relaxation** among all vertices of this cycle and vertices reachable from it. Consequently, if not limit the number of phases by $n - 1$, The algorithm will run indefinitely, constantly improving the distances to these vertices.

Hence we obtain a **criterion for achievable cycle of negative weight**. If after $n - 1$ phase we will perform another phase, and it happens at least one relaxation, then the graph contains a cycle of negative weight, attainable from v ; Otherwise, this loop is not.

Moreover, if such a cycle is detected, the Bellman-Ford algorithm can be modified so that it is deduced that the cycle itself as a sequence of vertices contained in it. It is enough to remember the number of vertices x in which there was a relaxation on n . The second phase. This summit will either lie on a cycle of negative weight, or it can be reached from it. To obtain the peak, which is guaranteed

on a cycle, it is sufficient, for example, n just go to the ancestors, starting from the top x . Get a room y vertices lying on a cycle, it is necessary to go from the top of this ancestor, until we return to the same vertex y (And it will happen, because the relaxation cycle of negative weight occur in a circle).

Implementation:

```

void solve() {
    vector<int> D (n,
    INF); d[v] = 0;
}

```

```
vector <int> P (n, -  
1);int x;  
for (int i =0; i <n; + + I) {
```

```

x = -1;
for (int j = 0; j <m; ++j)
    if (d[e[j]. A] <INF)
        if (d[e[j]. B] > D[e[j]. A] + E[j]. Cost) {
            d[e[j]. B] = Max (-INF, d[e[j]. A] +
                E
[j]. Cost);
            p[e[j]. B] = E[j].
A; x = e[j]. B;
        }
    }

if (x == -1)
    cout << "No negative cycle from" << v;
else {
    int y = x;
    for (int i = 0; i <n; +
        + I) y = p[y];
    vector <int> Path;
    for (int cur = y; ; cur =
        p[cur]) {
        path.push_back (cur);
        if (cur == y && path.size() > 1) break;
    }
    reverse (path.begin(), Path.end());
    cout << "Negative cycle:";
    for (size_t i = 0; i <path.size(); +
        + I) cout << Path[i] << ' ';
}
}

```

Because when there is a negative cycle for n iterations distance could go far in the minus (apparently negative numbers to order -2^n), The code has taken additional measures against such an integer overflow:

```
d[e[j]. B] = Max (-INF, d[e[j]. A] + E[j]. Cost);
```

In the above implementation is sought negative cycle reachable from some starting vertex v ; However, the algorithm can be modified so that it was looking for just **any negative cycle** in the graph. To do this, we must put all distances equal to zero and not infinity - as if we looking for the shortest path from all vertices at the same time; correctness of the negative cycle detection is not affected.

Extras on this task - see separate article "[Search the negative cycle in the graph](#)".

Problem in online judges

List of tasks that can be solved using the algorithm of Bellman-Ford:

- E-OLIMP # 1453 "Ford-Bellman" [Difficulty: Easy]
- UVA # 423 "MPI Maelstrom" [Difficulty: low]
- # 534 "Frogger" [Difficulty: average]
- UVA # 10099 "The Tourist Guide" [Difficulty: Medium]
- UVA # 515 "King" [Difficulty: Medium]

See also the list of tasks in the article "[Search the negative cycle](#)".

Leviticus algorithm finding the shortest paths from a given vertex to all other vertices

Let a graph with N vertices and M edges, each of which lists its weight L_i . Also given starting vertex V_0 .

Required to find the shortest path from vertex V_0 to all other vertices.

Leviticus algorithm solves this problem very efficiently (about the asymptotic behavior and speeds see below).

Description

Let array $D[1..N]$ will contain the current shortest path lengths, ie D_i - is the current length of the shortest path from vertex to vertex V_0 i. Originally filled array D values "infinity", except $D_{V_0} = 0$. By

completion of the algorithm, this array will contain the final shortest distance.

Let array $P[1..N]$ contains the current ancestors, ie P_i - the pinnacle preceding vertex i in the shortest path from the vertex V_0 to i . As well as an array D, array P is gradually changed in the course of the algorithm, and it receives the end of the final values.

Now, actually, the algorithm Leviticus. At each step, supported by three sets of vertices:

- M_0 - tops, distance to which has already been calculated (but perhaps not completely);
- M_1 - vertex distance are calculated;
- M_2 - vertex distance are not yet calculated.

The vertices in the set M_1 is stored in the form of a bidirectional queue (deque).

Initially all vertices are placed in the set M_2 , except vertices V_0 , which is placed in the set M_1 .

At each step of the algorithm, we take the top of the set M_1 (pull out the top element of the queue). Let V - is the selected vertex. Translate this vertex in the set M_0 . Then review all edges emanating from that vertex. Let T - is the second end of the current edge (ie not equal to V), and L - is the length of the current edge.

- If T belongs to M_2 , then transferred to a T set M_1 in the queue. DT is set equal to $D_V + L$.
- If T belongs to M_1 , we try to improve the value of DT : $DT = \min(DT, DV + L)$. The very top of T does not move in the queue.
- If T is in M_0 , and if possible to improve DT ($DT > DV + L$), then improving DT , and top T return to the set M_1 , placing it in the top of the queue.

Of course, whenever you update the array D must be updated and the value in the array P.

Implementation Details

Create an array $ID[1..N]$, in which each vertex will be stored, what set it belongs: 0 - if M_2 (ie, the distance is equal to infinity), 1 - if M_1 (ie the vertex is in the queue) 2 and - if M_0 (some path has been found, the distance is less than infinity).

Queue processing can implement standard data structure deque. However, there is a more efficient way. First, obviously, in the queue at any one time will be a maximum of N elements stored. But, secondly, we can add elements in the beginning and the end of the queue. Consequently, we can arrange a place on the array size N , but you have to let it loop. Ie do array $Q[1..N]$, pointers (int) to the first element QH and the element after the last QT . The queue is empty when $QH == QT$. Adding to the end - just a record in $Q[QT] QT$ and increase by 1; if QT then went beyond the line ($QT == N$), then do the $QT = 0$. Adding the queue - reduce $QH-1$, if it has moved beyond the stage of ($QH == -1$), then do the $QH = N - 1$.

The algorithm implementing exactly the description above.

Asymptotics

I do not know more or less good asymptotic estimate of this algorithm. I have met only estimate $O(NM)$ of the similar algorithm.

However, in practice, the algorithm has proven itself very well: while I appreciate his work as **$O(M \log N)$** , although, I repeat, is exclusively **Experimental** appraisal.

Implementation

```
typedef pair <int,int> rib;
typedef vector <vector <rib>> graph;

const int inf = 1000 * 1000 * 1000;

int main ()
{
    int n, v1, v2;
    graph g (n);
    Graph reading ... ...

    vector <int> d (n,
    inf); d [v1] = 0;
    vector <int> id (n);
    deque <int> q;
    q.push_back (v1);
    vector <int> p (n, -
    1);

    while (! q.empty ())
    {
        int v = q.front (), q.pop_front
        (); id [v] = 1;
        for (size_t i = 0; i <g [v]. size (); + + i)
        {
            int to = g [v] [i]. first, len = g [v] [i].
            second; if (d [to]> d [v] + len)
            {
                d [to] = d [v] +
                len; if (id [to] ==
                0)
                    q.push_back (to);
                else if (id [to] == 1)
                    q.push_front (to);
                p [to] = v;
                id [to] = 1;
            }
        }
    }

    Result output ... ...
}
```

Floyd's Algorithm-Uorshella finding shortest paths between all pairs of vertices

Given a directed or undirected weighted graph with n vertices. Required to find the values of all variables d_{ij} . The length of the shortest path from vertex i to the top j .

It is assumed that the graph contains no cycles of negative weight (then answer between some pairs of vertices may simply not exist - it will be infinitely small).

This algorithm has been simultaneously published in the papers of Robert Floyd (Robert Floyd) and Stephen Uorshella (Stephen Warshall) (Stephen Warshall) in 1962, after whom this algorithm is called nowadays. However, in 1959, Bernard Roy (Bernard Roy) published essentially the same algorithm, but its publication unnoticed.

Description of the algorithm

The key idea of the algorithm - the process of finding a partition of the shortest paths on **phase**.

Before k The second phase ($k = 1 \dots n$) Considered that in the matrix of distances stored length of the shortest paths, which contain as internal vertices only the vertices of the ~~set we number~~ the vertices of the graph, starting with the unit).

In other words, before the k Second phase value equal to the length of the shortest path from vertex i to the top j if allowed to enter this path only in the top with numbers less k (Beginning and end of the path are not considered).

Easy to see that this property is to perform for the first phase, it is enough to record distance matrix adjacency matrix: - the cost of edges from vertex to vertex. Thus

$$d[i][j] = g[i][j]$$

if the edges between any vertices not, should record the value of "infinity" ∞ . From the vertex to itself should always record the value of 0. It is critical for the algorithm.

Suppose now that we are on k The second phase, and we want **recalculate** Matrix so that it already meets the requirements for the second phase. Fix some vertices i and j . We arises two fundamentally different cases:

- The shortest path from vertex i on top, which is allowed to pass through additional vertices $\{1, 2, \dots, k\}$ coincides with the shortest path that is allowed to pass through vertices $\{1, 2, \dots, k-1\}$.

In this case the $d[i][j]$ not change during the transition from k The second to $k+1$ -Th phase.

- "New" was the shortest path **better** The "old" way.

This means that the "new" shortest path passes through the vertex k . Just note that we do not lose generality, considering only more simple way (ie paths that do not pass on some vertex twice).

Then we note that if we break this "new" way top k two halves (one going $i \Rightarrow k$ another $k \Rightarrow j$). Then each of these halves is not comes to the top k . But then it turns out that the length of each these halves was deemed a further $k+1$ -Th phase or even earlier, and it is sufficient to simply take the sum $d[i][k] + d[k][j]$, And give it the length of the "new" shortest path.

Combining these two cases, we obtain that k The second phase is required to recalculate the length of the shortest paths between all pairs of vertices i and j follows:

```
new_d[i][j] = Min (d[i][j], D[i][k] + D[k][j]);
```

Thus, all the work that is required to produce the second phase - is to iterate over all pairs of vertices and recalculate the length of the shortest path between them. As a result, after the n The second phase in the matrix

will be recorded length of the shortest path between distances and $i \Rightarrow j$ ∞ . Or, if the path between the vertices does not exist $d[i][j]$

The last remark should be done - that can be **not to create a separate**

Second phase: all the changes can be made

Matrix new_d temporary matrix of shortest paths on k immediately in a matrix. In fact, if we have improved (reduced) a value in the matrix of distances, we could not degrade thereby length of the shortest path for some other pairs of vertices processed later.

$d[]$

Asymptotics algorithm clear is $O(n^3)$.

Implementation

Fed to the input of the program graph specified as the adjacency matrix - two-dimensional array size $n \times n$, in which each element specifies the length of the edges between the vertices.

It is required to satisfy $d[i][i] = 0$ for any i .

```
for (int k = 0; k <n; + + K)
    for (int i = 0; i <n; + + I)
        for (int j = 0; j <n; + + J)
            d[i] [j] = Min (d[i] [j], D[i] [k] + D[k] [j]);
```

It is assumed that if some vertices between two **there is no edge**, The adjacency matrix, was recorded for a large number (large enough that it is greater than the length of any path in the graph); then this edge will always be profitable to take, and the algorithm works correctly.

However, unless you take special measures, in the presence of edges in the graph **negative weight**, the resulting matrix of the form may appear $\infty - 1, \infty - 2$, Etc., which, of course, still means that the respective vertices between the no path at all. Therefore, in the presence of negative edges of the graph algorithm Floyd better write it so that it did not fulfill the transitions of the states which already is "no way":

```
for (int k = 0; k <n; + + K)
    for (int i = 0; i <n; + + I)
        for (int j = 0; j <n; + + J)
            if (d[i] [k] <INF && d[k] [j] <INF)
                d[i] [j] = Min (d[i] [j], D[i] [k] + D[k] [j]);
```

Recovery paths themselves

Easy to maintain additional information - so-called "ancestors", which can restore itself the shortest path between any two given vertices in

a sequence of vertices.

It is sufficient to maintain the distance matrix except as **matrix ancestors** Which for every pair of vertices will contain the number of phases, which was obtained the shortest distance between them. It is clear that this phase number is not more than the "average" tip Seeking the shortest path, and now we just need to find the shortest path between the vertices and, as well as between and. $i \ p[i][j] \ p[i][j] \ j$

This yields a simple recursive algorithm for reconstructing the shortest path.

Case of real weights

If the weights of edges are not integers and real numbers, it should be borne errors that inevitably arise when dealing with floating-point types.

Floyd's algorithm applied to unpleasant special effect of these errors becomes that found distance algorithm can go badly due to the minus **accumulated errors**. In fact, if the first phase has been a mistake Δ , On the second iteration of this error can already turn into 2Δ , The third - in 4Δ And so forth.

To avoid this, the comparison algorithm Floyd should be done taking into account the error:

```
if (d[i] [k] + D[k] [j] <D[i] [j] -
    EPS) d[i] [j] = D[i] [k] +
    D[k] [j];
```

The case of negative cycles

If the graph contains a cycle of negative weight, then formally Warshalla-Floyd algorithm is not applicable to such graph. In fact, for those pairs of vertices i and j between which it is impossible to go into a cycle of negative weight,

algorithm will work correctly.

For those pairs of vertices, the answer to which does not exist (because of the presence of a negative cycle in the path between them), Floyd's algorithm to find an answer to a number of (possibly highly negative, but not required). Nevertheless, it is possible to improve the algorithm of Floyd, he carefully processed to such a pair of vertices, and drew for them, for example, $-\infty$.

This can be done, for example, the following **criterion** "Is not the existence of the way." So, even for a given graph worked usual algorithm Floyd. Then between the vertices i and there is no shortest path if and only if there is a vertex t , Accessible from i and from which is achievable for which holds $d[t][t] < 0$.

In addition, when using the Floyd algorithm for graphs with negative cycles should remember that arise in the process of distance can go into much less exponentially with each phase. Therefore, measures should be taken against an integer overflow, limiting all distances below some value (eg, $-\text{INF}$).

More information about this task, see separate article: "[Finding a negative cycle in the graph](#)".

Shortcuts fixed length, number of tracks fixed length

The following describes these two objectives, built on the same idea: to reduce the problem to the construction of the power matrix (with the usual multiplication, and modified).

Number of paths of fixed length

Given a directed graph with unweighted n vertices, and given an integer. Required for each pair of vertices i and j to find the number of paths between these vertices consisting of exactly k edges. Paths wherein arbitrary addresses are not necessarily simple (i.e., vertices may be repeated any number of times).

We assume that the graph is given **adjacency matrix** size $n \times n$. Where each element $g[i][j]$ equals one if between these vertices is an edge, and zero if there is no edge. Described following algorithm works in the case of multiple edges if between some vertices and i is at once m ribs, the adjacency matrix should record this number m in the graph, if any.

Obviously, in this form, **adjacency matrix** graph is **answer to the problem at** $k = 1$. It contains a number of paths of length 1 between each pair of vertices.

The decision will build **iteratively**. Let the answer for some k is found, we show how to build it $k + 1$. We denote d_k matrix found answers to k . And through d_{k+1} - Matrix of responses that to build. Then clear the following formula:

Easy to see that recorded above formula - nothing more than the product of two matrices d_k and in the the usual sense:

Thus, **decision** This problem can be represented as follows:

$$d_k = \underbrace{g \cdot \dots \cdot g}_{k \text{ times}} = g^k.$$

It remains to note that the construction of the power matrix can be done efficiently using the algorithm **Binary exponentiation**.

Thus, the obtained solution has the asymptotic $O(n^3 \log k)$ and is binary exponentiation k th degree of adjacency matrix.

Shortcuts fixed length

Given a directed weighted graph with vertices, and given an integer. Required for each a pair of vertices

and find the length of the shortest path between these vertices, consisting of exactly k ribs.

We assume that the graph is given **adjacency matrix** size $n \times n$. Where each element $g[i][j]$ contains the length of the ribs from i to j . If the edges between any vertices not the corresponding element of the matrix to be equal to infinity ∞ .

Obviously, in this form, **adjacency matrix** graph is **answer to the problem at** $k = 1$. It contains the length of the shortest paths between each pair of vertices, or ∞ if the path length 1 does not exist.

The decision will build **iteratively**. Let the answer for some k is found, we show how to build it $k + 1$. We denote d_k matrix found answers for, and through d_{k+1} - the matrix of responses that to build. Then clear the following formula:

$$d_{k+1}[i][j] = \min_{p=1 \dots n} (d_k[i][p] + g[p][j]).$$

Look closely at this formula, it is easy to draw an analogy with the matrix multiplication: in fact, the matrix

d_k is multiplied by the matrix g . Only in the multiplication operation instead of the sum of all p minimum is taken over all p :

$$d_{k+1} = d_k \odot g,$$

where the operation \odot multiplication of two matrices is defined as follows:

$$A \odot B = C \iff C_{ij} = \min_{p=1 \dots n} (A_{ip} + B_{pj}).$$

Thus, **decision** This problem can be represented by this multiplication as follows:

$$d_k = \underbrace{g \odot \dots \odot g}_{k \text{ times}} = g^{\odot k}.$$

It remains to note that the construction of the power of this multiplication can be performed effectively by using the algorithm [Binary exponentiation](#) As the only required for his property - associativity of multiplication - obviously there.

Thus, the obtained solution has the asymptotic behavior $O(n^3 \log k)$ and is binary exponentiation k -Th power of the adjacency matrix of the graph with the modified matrix multiplication.

Minimum spanning tree. Prim's algorithm

Given a weighted undirected graph with n peaks and m ribs. Required to find a subtree of this graph, which would connect all the vertices, and thus has the lowest possible weight (ie, the sum of the weights of the edges). Subtree - a set of edges connecting vertices, and every vertex can reach any other by exactly one simple way.

This subtree is called a minimal spanning tree, or simply **the minimum spanning tree**. Easy to understand that any framework will necessarily contain $n - 1$ edge.

In natural setting This problem is as follows: there n cities, and for each pair of known connection cost them dear (or know that they can not connect). You want to connect all cities so that you can get from any city to another, and thus the cost of construction of roads would be minimal.

Prim's algorithm

This algorithm is named after the American mathematician Robert Primus (Robert Prim), which opened this algorithm in 1957, however, in 1930, this algorithm has been opened by the Czech mathematician Wojtek Jarnik (Vojtěch Jarník). Furthermore, Edgar Dijkstra (Edsger Dijkstra) in 1959 as invented this algorithm independently.

Description of the algorithm

Himself **algorithm** has a very simple form. Seeking a minimal skeleton is constructed gradually by adding to it the edges one by one. Originally skeleton relies consisting of a single vertex (it can be chosen arbitrarily). Then select the minimum weight edge emanating from the vertex, and is added to the minimum spanning tree. After this framework already contains two vertices, and now sought and added an edge of minimum weight, with one end of one of the two selected vertices, and the other - on the contrary, in all other than these two. And so on, i.e. whenever sought minimum weight edge, one end of which - is taken into the backbone of the vertex and the other end - not yet taken, and this edge is added to the skeleton (if several such edges, we can take any). This process is repeated until the frame will not yet contain all peaks (or, equivalently, $n - 1$ edge).

As a result, the skeleton will be built, which is minimal. If the graph was originally not connected, then the skeleton will not be found (the number of selected edges will be less $n - 1$).

Proof

Suppose that the graph was connected, ie answer exists. We denote the skeleton found Prim algorithm, and through S - Minimum core. Obviously, that really is the skeleton (ie, the subtree of the graph). Show that weight S and T coincide.

Consider the first time when it is added edge that is not in the best frame

We denote this edge through e . Ends it - through a and a plurality of incoming at that time in the skeleton vertices - by (according to the algorithm) b , c . Or vice versa). Optimally skeleton S and peaks G connected in some way; in this way we find any edge, one end of which lies in, and the other - instead of the edge, then this means that the not. Since Prim algorithm chosen edge e weight of the rib a greater than or equal to weight of the edge. P e g V .

Now remove e from the edge, and add an edge e . By just what to say, as a result of the weight of the core could not increase (decrease it too could not because e was optimal). Furthermore, S not ceased to be a skeleton (The fact that the connection a is not broken, it is easy to make: we closed the way to the ring, and then removed from this cycle one edge).

Thus we have shown that we can choose the optimum frame this procedure as many times, we find that we can choose the optimum frame s. Consequently, the weight of the constructed algorithm P T

thus it will comprise a rib. Repeating so that it matches S minimal, as required.

Implementation

Time of the algorithm depends essentially on how we search the minimum of the next edge of suitable edges. There may be different approaches lead to different asymptotic behavior and different implementations.

Trivial Pursuit: algorithms for $O(nm)$ and $O(n^2 + m \log n)$

If we look for an edge every time just browsing among all possible options, then asymptotically will be required to show $O(m)$ Ribs to find among all admissible edge with the least weight. Album asymptotic behavior of the algorithm in this case will be $\Theta(nm)$. That in the worst case, there is a too slow algorithm.

This algorithm can be improved if we view each time not all the edges, but only one edge of each vertex is selected. For this example, you can sort the edges of each vertex in the order of increasing weights, and store a pointer to the first valid edge (recall allowed only those edges that are not yet in the set of selected vertices). Then, if recalculate these pointers for each

add edge to the backbone, the total asymptotic behavior of the algorithm will be $O(n^2 + m \log n)$. But pre-need to sort all the edges in $O(m \log n)$. That in the worst case (for a dense graphs) gives an asymptotic $O(n^2 \log n)$.

Below we consider two slightly different algorithms: for dense and sparse graphs, received as a result significantly better asymptotic behavior.

Case of dense graphs: an algorithm for $O(n^2)$

$O(n)$

$O(n)$

Thus, we have an option of Prim's algorithm with the asymptotic $O(n^2)$.

In particular, such an implementation is particularly useful for solving the so-called **Euclidean problem on the minimum spanning tree**: when given n points on the plane, the distance between which is measured according to the standard Euclidean metric, and want to find the skeleton of minimum weight connecting them all (and add new vertices elsewhere is prohibited). This problem is solved by the algorithm described herein for $O(n^2)$ time and $O(n)$ memory, which will not work to achieve **Kruskal's algorithm**.

Implementation of Prim's algorithm for the graph given by the adjacency matrix $g[]$:

```
// input int n;
vector<vector<int>> G;
const int INF = 1000000000; // value "Infinity"

// algorithm
vector<bool> Used(n);
vector<int> Min_e(n, INF), Sel_e(n, -1); min_e[0] = 0;
for (int i = 0; i < n; ++i)
    if (Used[i]) { int v = -1;
        for (int j = 0; j < n; ++j)
            if (!Used[j] && (v == -1 || Min_e[j] < Min_e[v])) v = j;
        if (min_e[v] == INF) {
            cout << "No MST!";
            exit(0);
        }
        used[v] = true;
        if (Sel_e[v] == -1)
            cout << v << " " << Sel_e[v] << endl;
        for (int to = 0; to < n; ++to)
            if (g[v][to] < Min_e[to]) {
                min_e[to] = g[v][to];
                Sel_e[to] = v;
            }
    }
}
```

The input is the number of vertices n , matrix g size $n \times n$, which marked the edge weights, and are number m . If no corresponding rib. The algorithm supports three arrays: flag $used[i]$ means that the peak is included in the frame, the value $min_e[i]$ keeps the weight of the smallest admissible edge from vertex i . Element $sel_e[i]$ contains the lowest end of the rib (this is necessary for output edges of the reply). The algorithm makes n steps, each of which selects the top v with the smallest label min_e , Marks the $used$ And then looks at all the edges of this vertex, recounting their labels.

Case of sparse graphs: an algorithm for $O(m \log n)$

In the above algorithm can be seen finding the minimum standard operations in the set and change the values in this set. These two operations are classic, and perform many data structures, for example, implemented in C++, red-black tree set.

Within the meaning of the algorithm is exactly the same, but now we can find the minimum edge during

On the other hand, the time for

recalculation

pointers now be

That is worse than in

$O(\log n)$

the above algorithm.

If we consider that there will be m pointers and conversions $O(n)$ search for the minimum edge, then asymptotic behavior of the total amount $O(m \log n)$ - For sparse graphs is better than both above algorithm, but on dense graphs, this algorithm will be slower previous.

Implementation of Prim's algorithm for graph adjacency lists specified $g[]$:

```
// input int n;
vector<vector<pair<int,int>>> G;
const int INF = 1000000000; // value "Infinity"

// algorithm
vector<int> Min_e (n, INF), Sel_e (n, -1); min_e[0] = 0;
set<pair<int,int>> Q;
Q.insert (make_pair (0,0));
for (int i = 0; i <n; ++ I) {
    if (Q.empty ()) {
        cout << "No MST!";
        exit(0);
    }
    int v = Q.begin ()->
    Second; Q.erase
    (Q.begin ());
    if (sel_e[v] == -1)
        cout << v << "" << Sel_e[v] << endl;

    for (size_t j = 0; j < g[v].Size(); ++
        J) { int to = g[v][j].First,
        cost = g[v][j].Second; if (cost < min_e[to]) {
            Q.erase (make_pair (min_e[to], To));
            min_e[to] = Cost;
            sel_e[to] = v;
            Q.insert (make_pair (min_e[to], To));
        }
    }
}
}
```

The input is the number of vertices and the adjacency lists: $g[]$ - A list of all edges emanating from the vertex to as pairs (the second end of the edge Weight of the edge). The algorithm supports two arrays $min_e[i]$ stores value i the smallest possible weight from the top edge, and the element $sel_e[i]$ contains the lowest end of the rib (It is necessary to bring the edges of the reply). Additionally, a queue Q all of the nodes in increasing order of their tags. The algorithm makes n steps, each of which selects the top v with the smallest label min_e (Just removing it from the queue), and then looks at all the edges of this vertex, recounting their labels (when calculated from the queue, we remove the old value, and then put back a new one).

The analogy with the Dijkstra algorithm

In just two described algorithms can be traced quite clear analogy with [Dijkstra algorithm](#) He has the same structure ($n - 1$ phase, each of which selects an optimum first edge is added to the response and then converted values for all vertices not yet selected). Moreover, the Dijkstra's algorithm has also

two embodiments: in $O(n^2)$ and $\Theta(m \log n)$ (Of course we do not consider here the possibility the use of complex data structures to achieve even smaller asymptotics).

If you look at Prima and Dijkstra algorithms more formally, it turns out that they are generally identical to each other except **the weighting function** vertices: if Dijkstra each vertex supported length of the shortest path (ie the sum of the weights of some edges), the algorithm Prima each vertex attributed only to the minimum weight edge leading into the set of vertices already taken.

At the implementation level, this means that after the addition of the next vertex v in the set of selected vertices when we begin to view all the edges (v, to) from this vertex, the algorithm Prima pointer updated weight of the edge (v, to) , And Dijkstra mark distance $d[v]$ updated sum tags to and edge (v, to) . Otherwise, these two algorithms can be considered identical (although they decide to weight (v, to) completely different tasks).

Properties of the minimum spanning tree

- **Maximum** skeleton can also search algorithm Prima (E.g., replacing all the weights of the edges on the contrary: the algorithm does not require the non-negativity of edge weights).
- Minimum frame **is unique**, If the weights of all edges are distinct. Otherwise, there may be some minimum spanning tree (which one will be selected Prima algorithm depends on the order of viewing edges / vertices with the same weights / pointers)
- Minimum frame is also the core, **minimum for the product** all edges (assuming that all weights are positive). In fact, if we replace the weights of all edges in their logarithms, it is easy to notice that in the algorithm will not change anything, and will be found the same edge.
- Minimum frame is the skeleton of a minimum weight **the heaviest edge**. Most clearly this statement is clear if we consider the work [Kruskal's algorithm](#).
- **Minimality criterion** skeleton: backbone is minimal if and only if for any edges not belonging to the skeleton, the loop formed by this edge by adding to the core, no edges heavier than the edge. In fact, if for some edge turned out that it is easier for some ribs formed loop, the frame can be obtained with a smaller weight (an edge is added to the skeleton, and by removing the heaviest edge of the loop). If this condition is not fulfilled for any edge, then all these edges do not improve the weight of the core when they are added.

Minimum spanning tree. Kruskal's algorithm

Given a weighted undirected graph. Required to find a subtree of this graph, which would connect all the vertices, and thus has the least weight (ie, the sum of the weights of the edges) of all. This subtree is called a minimal spanning tree, or simply the minimum core.

It will discuss some important facts related to the minimum spanning tree, then be considered Kruskal's algorithm in its simplest implementation.

Properties of the minimum spanning tree

- Minimum frame **unique if the weights of all edges are distinct**. Otherwise, there may be several minimal cores (specific algorithms usually get one of the possible cores).
- Minimum frame is also **skeleton with minimal product** weights of the edges. (Prove it's easy enough to replace the weight of all edges on their logarithms)
- Minimum frame is also **backbone with a minimum weight of the heaviest edge**. (This follows from the validity of Kruskal's algorithm)
- **The skeleton of the maximum weight** sought similar skeleton of minimum weight, enough to change signs all edges reversed and perform any of the minimum spanning tree algorithm.

Kruskal's algorithm

This algorithm has been described by Kruskal (Kruskal) in 1956

Kruskal's algorithm initially assigns each vertex in your tree, and then gradually brings these trees, combining the two at each iteration some wood some edge. Before starting the algorithm, all the edges are sorted by weight (in decreasing order). Then begins the process of

Unions: get over all the edges from first to last (in the sort order), and if the current edge of its ends belong to different subtrees, these subtrees are combined, and the edge is added to the answer. At the end of sorting all edges vertices will belong to the same subtree, and the answer is found.

The simplest implementation

This code is directly implements the algorithm described above, and performed for **O (M log N + N²)**.

+ N²). Sort edges require O (M log N) operations. Vertices belonging to a particular subtree simply stored using an array tree_id - it is stored for each vertex tree number, which it belongs. For each edge we in O (1) define, whether it ends belong to different trees. Finally, the union of two trees is carried out at O (N) simply pass the array tree_id. Given that

all merge operations is N-1, we obtain the asymptotic behavior **O (M log N + N²)**.

```
int m;
vector<pair<int, pair<int,int>> g (m); // Weight - the top one - the top
2

int cost = 0;
vector<pair<int,int>> res;

sort (g.begin (), g.end
()); vector<int> tree_id
(n); for (int i = 0; i <n;
+ + i)
    tree_id [i] =
i; for (int i = 0; i
<m; + + i)
{
    int a = g [i]. second.first, b = g [i]. second.second, l = g [i].
first; if (tree_id [a] != tree_id [b])
    {
        cost + = l;
        res.push_back (make_pair (a, b));
        int old_id = tree_id [b], new_id = tree_id
[a]; for (int j = 0; j <n; + + j)
            if (tree_id [j] == old_id)
                tree_id [j] =
new_id;
    }
}
```

An improved

Using the data structure "The system of disjoint sets" You can write faster implementation [Kruskal's algorithm](#) with the asymptotic $O(M \log N)$.

Minimum spanning tree. Kruskal's algorithm with a system of disjoint sets

Description of the problem and Kruskal's algorithm, see [here](#).

This implementation will be discussed using the data structure "A system of disjoint sets "(DSU), which will reach the asymptotic behavior **O (M log N)**.

Description

Just as in the simple version of Kruskal's algorithm, we can sort all the edges in non-decreasing weight. Then put each node in your tree (ie its set) by calling the DSU MakeSet - it will take in the amount of $O (N)$. Loop through all the edges (in the sort order), and for each edge in $O (1)$ define, whether it ends belong to different trees (using two calls FindSet $O (1)$). Finally, the union of two trees will be calling the Union - also $O (1)$. Overall, we obtain the asymptotic behavior of $O (M \log N + N + M) = O (M \log N)$.

Implementation

To reduce the amount of code and carry out all operations are not as separate functions, and directly in the code of Kruskal's algorithm. Here we will use a randomized version of the DSU.

```
vector <int> p (n);

int dsu_get (int v) {
    return (v == p [v])? v: (p [v] = dsu_get (p [v]));
}

void dsu_unite (int a, int b)
{
    a = dsu_get (a);
    b = dsu_get (b);
    if (rand () & 1)
        swap (a, b);
    if (a! = b)
        p [a] = b;
}

... Function main (): ...

int m;
vector <pair <int, pair <int,int>> g; // Weight - the top one - the top 2
Graph reading ... ...

int cost = 0;
vector <pair <int,int>> res;

sort (g.begin (), g.end
()); p.resize (n);
for (int i = 0; i <n; +
    + i) p [i] = i;
for (int i = 0; i <m; + + i) {
    int a = g [i]. second.first, b = g [i]. second.second, l = g [i].
    first; if (dsu_get (a)! = dsu_get (b)) {
        cost + = l;
        res.push_back (g [i].
        second); dsu_unite (a, b);
    }
}
```

Kirchhoff matrix theorem. Finding the number of spanning trees

Asked connected undirected graph of its adjacency matrix. Multiple edges in the graph are allowed.

Required to count the number of different spanning trees of this graph.

The following formula belongs Kirchhoff (Kirchhoff), who proved it in 1847

Kirchhoff matrix theorem

Take the adjacency matrix of G, replace each element of this matrix is the opposite, and a diagonal element instead of A_{ii} , i put the degree of vertex i (if there are multiple edges, the vertex degree they are accounted for with its multiplicity). Then, according to Theorem Kirchhoff matrix, all the cofactors of this matrix are equal, and equal to the number of spanning trees of this graph. For example, you can delete the last row and last column of the matrix, and the modulus of its determinant is equal to the desired quantity.

Determinant of a matrix can be found in $O(N^3)$ using [Gauss method](#) or [Kraut method](#).

The proof of this theorem is rather complicated and is not presented here (see, for example, coming VB "dimer problem and the Kirchhoff theorem").

Communication with the laws of Kirchhoff circuit

Between the matrix and the Kirchhoff theorem Kirchhoff laws for circuit has a surprising connection. It can be shown (as a consequence of Ohm's law and Kirchhoff's first law), that the resistance R_{ij} between points i and j is equal to the electrical circuit:

$$R_{ij} = |T(i, j)| / |T_j|$$

where the matrix T is obtained from the matrix A inverse Conductor resistance (A_{ij} - opposite number to Resistance conductor between points i and j) transformation matrix described in the Kirchhoff's theorem, and designation of $T(i)$ denotes the deletion of the row and column number i , and $T(i, j)$ - the deletion of two rows and columns i and j . Kirchhoff theorem gives this formula geometric meaning.

Prüfer code. Cayley formula. The number of ways to make a graph connected

In this article we consider the so-called **Prüfer code** which represents a coding method for uniquely tagged using a sequence tree properties.

Using Prüfer codes demonstrates proof **Cayley's formula** (Specifying the number of spanning trees in a complete graph), as well as solving the problem of the number of ways to add to the given graph edges to turn it into a coherent.

Note. We will not consider trees consisting of a single vertex, - is a special case, where many of the statements degenerate.

Prüfer code

Prüfer code - a way bijective coding labeled trees with n vertices by a sequence $n - 2$ integers in the interval.

In other words, the code Prüfer - is $[1; n]$

bijection among all spanning trees of a complete graph and numerical sequences.

Although use Prüfer code for storing and manipulating trees impractical due to the specificity of representation codes Priifer find application in solving combinatorial problems.

Author - Heinz Prüfer (Heinz Prüfer) - suggested this code in 1918 as proof of Cayley's formula (see below).

Building code Priifer for that tree

Prüfer code is constructed as follows. Times will prodelyvat procedure: choose a tree leaf
c lowest number, remove it from the tree, and add the code Priifer node number, which has been linked
c this leaf. Eventually there will be only a tree 2 tops, and this completes the algorithm (number of
vertices explicitly in the code are not recorded).

Thus, for a given code Priifer tree - a sequence of $n - 2$ numbers where each number

- Node number associated with the smallest at the time sheet - ie is a number in the interval $[1; n]$.

Algorithm for computing Prüfer code is easy to implement with the asymptotic $O(n \log n)$ Simply maintaining
structure for extracting minimum data (e.g., set <> or priority_queue <> in C +
+) Containing a list of all current leaves:

```
const int MAXN = ...;
int n;
vector <int>
G[MAXN]; int
degree[MAXN]; bool
killed[MAXN];

vector <int> Prufer_code()
{ set <int> Leaves;
for (int i = 0; i < n; ++ I)
    degree[i] = (int) g[i].
    Size(); if (degree[i] == 1)
        leaves.insert (i);
    killed[i] = false;
}

vector <int> Result (n-2);
for (int iter = 0; iter < n-2; ++
    Iter) { int leaf = *
leaves.begin();
leaves.erase (leaves.begin());
killed[leaf] = true;
int v;
for (size_t i = 0; i < g[leaf]. Size(); +
    + I) if (! Killed[g[leaf] [i]])
        v = g[leaf]
[i]; result[iter] = v;
```

```

        if (- Degree[v] == 1)
            leaves.insert (v);
    }
    return result;
}

```

However, the construction of Prüfer code can be implemented in linear time and that is described in the next section.

Building code Priifer for that tree in linear time

We present here a simple algorithm, having the asymptotics $O(n)$.

The essence of the algorithm is to store **moving the pointer**. That will always be move only in the direction of increasing numbers of vertices.

At first glance, this is not possible, because in the process of building code Priifer rooms leaves can both increase and **decrease**. But it is easy to notice that the reduction occurs only in one case: if you remove the current code sheet his ancestor has a smaller number (this will be the ancestor of the minimum sheet and removed from the tree on the very next step Prüfer code). Thus,

reduce cases can be treated in time, and does not interfere with the construction of the algorithm

linear asymptotic behavior:

```

const int MAXN = ...;
int n;
vector <int> G[MAXN];
int parent[MAXN], Degree[MAXN];

void dfs (int v) {
    for (size_t i = 0; i < g[v].Size(); ++
        I) { int to = g[v][i];
        if (to != parent[v]) {
            parent[to] = v;
            dfs (to);
        }
    }
}

vector <int> Prufer_code()
{ parent[n-1] = -1;
    dfs (n-1);

    int ptr = -1;
    for (int i = 0; i < n; ++ I) {
        degree[i] = (int) g[i].
        Size(); if (degree[i] == 1 && Ptr
        == -1)
            ptr = i;
    }

    vector <int>
    Result; int leaf =
    ptr;
    for (int iter = 0; iter < n-2; + +
        Iter) { int next =
        parent[leaf];
        result.push_back (next); -
        Degree[next];
        if (degree[next] == 1 && Next < ptr)
            leaf = next;
        else {
            + + Ptr;
            while (ptr < n && degree[ptr] !=
                1) + + Ptr;
            leaf = ptr;
        }
    }
    return result;
}

```

Comment on this code. The main function here - `prufer_code()` which returns a code for Priifer tree set in global variables `n` (Number of vertices) and `g` (Adjacency lists specifying the graph). Initially

we find for each vertex of its ancestor $\text{parent}[i]$ - le ancestor of which this node will have
the time of disposal of wood (all we can find in advance, using the fact that the maximum peak $n - 1$

never removed from the tree). Also, we find for each vertex its degree. Variable $\text{degree}[]$ - This is the current sheet with a minimum number. Thus, each iteration increase. Variable ptr - This is the current sheet with a minimum number. Thus, each iteration in response, as well as making sure there was not less than the current candidate leaf If it turned out less, we simply assign $\text{leaf} = \text{parent}[\text{leaf}]$. And otherwise - move the pointer ptr before the next sheet.

As can be seen easily by the code, the asymptotic behavior of the algorithm is really a pointer ptr undergo only $O(n)$ changes, and all other parts of the algorithm is obviously work in linear time.

Some properties of codes Priifer

- Upon completion of construction of Prüfer code in the tree remain unremoved two vertices.

One of them will surely be the vertex with the maximum number - $n - 1$, But about the other vertex is nothing definite can be said.

- Each vertex occurs in code Priifer certain number of times equal to its power minus one.

This is easily understood if we note that the vertex is removed from the tree at a time when its degree is equal to one - ie at this point all edges adjacent to it, but one, has been removed. (For the two remaining vertices after building code this statement is also true.)

Recovering tree by its code Priifer

To restore the wood enough to see from the previous paragraph, that the degrees of all vertices in the target tree, we already know (and can calculate and save from an array). Consequently, we can find all leaves and, consequently, the smallest sheet number - has been removed at the first step. This sheet was connected to the top, the number of which is recorded in the first cell Prüfer code.

Thus, we find the first edge, remote Priifer code. Add this edge back, then reduce power $\text{degree}[]$ both ends of the rib.

We will repeat this operation until you have reviewed all the code Priifer: search with minimal vertex

$\text{degree} = 1$ Connecting it to the next vertex Prüfer code, reduce $\text{degree}[]$ at both ends.

In the end we are left with only two vertices
Prüfer left unremoved. Connect them an edge. $\text{degree} = 1$ - Those peaks, which algorithm

The algorithm is complete, the desired tree is built.

Realize This algorithm is easily during : Maintaining a data structure for extracting the minimum (e.g., `set <>` or `priority_queue <>`) numbers of all vertices with $\text{degree} = 1$ And removing from it every time at least.

Here are the appropriate implementation (where the function returns a list of the edges of the required tree):

```
vector<pair<int,int>> Prufer_decode (const vector<int> &Prufer_code) {
    int n = (int) prufer_code.size() + 2;
    vector<int> Degree (n,
    1);for (int i =0; i <n-2;
    + + I)
        + + Degree[prufer_code[i]];

    set<int> Leaves;
    for (int i =0; i <n; + + I)
        if (degree[i] == 1)
            leaves.insert (i);

    vector<pair<int,int>> Result;
    for (int i =0; i <n-2; + + I) {
        int leaf = * leaves.begin();
        leaves.erase (leaves.begin());
        int v = prufer_code[i];
        result.push_back (make_pair (leaf,
        v));if (- Degree[v] == 1)
            leaves.insert (v);
    }
}
```

```
result.push_back (make_pair (* Leaves.begin(), * - Leaves.  
End())); return result;  
}
```

Recovering tree code Priifer in linear time

For the algorithm with linear asymptotics can apply the same method that was used to obtain a linear algorithm for computing Prüfer code.

In fact, in order to find the lowest-numbered sheet optionally start the data structure to retrieve the minimum. Instead, you'll notice that after we find and treat the current sheet, it adds to the consideration of only one new vertex. Consequently, we can get one with a moving pointer variable, over a minimum current list:

```
vector<pair<int,int>> Prufer_decode_linear (const vector<int> &
Prufer_code) {
    int n = (int) prufer_code.size() + 2;
    vector<int> Degree(n, 1);
    for (int i = 0; i < n-2; ++ I) ++
        Degree[prufer_code[i]];
    int ptr = 0;
    while (ptr < n && degree[ptr] != 1)
        ++ Ptr;
    int leaf = ptr;

    vector<pair<int,int>> Result;
    for (int i = 0; i < n-2; ++ I) {
        int v = prufer_code[i];
        result.push_back (make_pair (leaf, v));
        -- Degree[leaf];
        if (-- Degree[v] == 1 && v < ptr)
            leaf = v;
        else {
            ++ Ptr;
            while (ptr < n && degree[ptr] != 1)
                ++ Ptr;
            leaf = ptr;
        }
    }
    for (int v = 0; v < n-1; ++ V)
        if (degree[v] == 1)
            result.push_back (make_pair (v, n-1));
    return result;
}
```

One correspondence between trees and Prüfer codes

One hand, for every tree there is exactly one code Prüfer, corresponding (this follows from the definition of Prüfer code).

On the other hand, the correctness of the algorithm for reconstructing the tree code Priifer that any Prüfer code (ie sequence of $n - 2$ numbers where each number lies in the interval $[1; n]$) corresponds to a tree.

Thus, all the trees and all the codes Priifer form **bijection**.

Cayley formula

Cayley's formula states that **number of spanning trees in full labeled graph** of n vertices is equal to:

$$n^{n-2}.$$

There are many **evidence** this formula, but the proof using codes Priifer clearly and constructively.

In fact, any set of $n - 2$ numbers from the interval corresponds uniquely to a tree of n vertices. Unique codes Priifer n^{n-2} . As in the case of a complete graph of n vertices as

core fits any tree, the number of spanning trees and power n^{n-2} , As required.

The number of ways to make a graph connected

Power Prüfer codes is that they allow you to get a more general formula than formula Cayley.

Thus, a graph of n and peaks m ribs; let k the number of connected components in this graph.

Required to find the number of ways to add an edge to the graph become connected (obviously rib - the minimum number of edges to make the graph connected).

Derive ready formula for solving this problem.

We denote s_1, \dots, s_k sizes of connected components of the graph. Since adding edges connected components inside smoking, it turns out that the problem is very similar to the number of search spanning trees in a complete graph of the vertices: but the difference here is that each vertex has a "weight" s_i :

Each edge adjacent to i Th vertex multiplies response s_i .

Thus, for counting the number of ways is important what degree are all vertices in the backbone. To obtain equations for the problem must be summed over all possible answers powers.

Let d_1, \dots, d_k - The degree of the vertices in the backbone. Sum of the vertex degrees is equal to twice the number of edges, so:

If i -vertex has d_i degree, then it enters the code Prüfer $d_i - 1$ times. Prüfer Code for the tree of vertices has a length $k - 2$. Number of ways to choose a set of $k - 2$ numbers, where the number occurs exactly $d_i - 1$ times power **multinomial coefficient** (By analogy with **binomial coefficient**)

$$\binom{k-2}{d_1-1, d_2-1, \dots, d_k-1} = \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}.$$

Given the fact that each edge adjacent to the i Th vertex multiplies response s_i We find that the answer provided that the degrees of vertices are d_1, \dots, d_k Is equal to:

$$s_1^{d_1} \cdot s_2^{d_2} \cdot \dots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}.$$

To answer this problem we must sum over all possible valid formula sets $\{d_i\}_{i=1}^{i=k}$:

$$\sum_{\substack{d_i \geq 1, \\ \sum_{i=1}^k d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdot \dots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}.$$

To minimize this formula we use the definition of multinomial coefficient:

$$(x_1 + \dots + x_m)^p = \sum_{\substack{c_i \geq 0, \\ \sum_{i=1}^m c_i = p}} x_1^{c_1} \cdot x_2^{c_2} \cdot \dots \cdot x_m^{c_m} \cdot \binom{m}{c_1, c_2, \dots, c_k}.$$

Comparing this with the previous formula, we find that if we introduce the notation $e_i = d_i - 1$:

$$\sum_{\substack{e_i \geq 0, \\ \sum_{i=1}^k e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdot \dots \cdot s_k^{e_k+1} \cdot \frac{(k-2)!}{e_1! e_2! \dots e_k!},$$

after folding **answer to the problem** is equal to:

$$s_1 \cdot s_2 \cdot \dots \cdot s_k \cdot (s_1 + s_2 + \dots + s_k)^{k-2} = s_1 \cdot s_2 \cdot \dots \cdot s_k \cdot n^{k-2}.$$

(This formula holds for $k = 1$ Although formal proof of it should not have.)

Problem in online judges

Problem in online judges, which use codes Prüfer:

- UVA # 10843 "Anne's game" [Difficulty: Easy]
- TIMUS # 1069 "Code Priifer" [Difficulty: Easy]
- CODEFORCES 110D "Clues" [Difficulty: Medium]
- TopCoder SRM 460 "TheCitiesAndRoadsDivTwo" [Difficulty: Medium]

Finding a negative cycle in the graph

Given a directed weighted graph with n nodes and m edges. You want to find in it any **cycle of negative weight**, if available.

When another formulation of the problem - you want to find **all pairs of vertices** such that there exists a path between any number of small length.

These two variants of the problem can be conveniently solved by different algorithms so below will be considered both of them.

One of the common "life" performances of this problem - the following: known **Currency Rates** courses transfer from one currency to another. You want to know whether a certain sequence of exchanges to benefit, ie started with one unit of a currency, receive as a result of more than one unit of the same currency.

The decision by the algorithm of Bellman-Ford

Bellman-Ford algorithm allows you to check the presence or absence of negative weight cycle in the graph, when its presence - a one of such cycles.

We will not go into details here (which are described in [Article algorithm Bellman-Ford](#)), And give only the result - how the algorithm works.

Done n iterations of Bellman-Ford, and if at the last iteration has been no change - then the negative cycle in the graph no.

Otherwise, take the vertex, the distance to which the change, and we will go from her ancestor, while not going into the cycle; this cycle will be the desired negative cycle.

Implementation:

```
struct edge {
    int a, b, cost;
};

int n, m;
vector<edge>
e;
const int INF = 1000000000;

void solve() { vector<int> D(n);
    vector<int> P(n, -1);
    int x;
    for (int i = 0; i < n; ++i)
        if (D[e[i].B] > D[e[i].A] + E[i].Cost) {
            D[e[i].B] = Max(-INF, D[e[i].A] + E[i].Cost);
            P[e[i].B] = E[i].A;
            x = e[i].B;
        }
    }

    if (x == -1)
        cout << "No negative cycle found.";
    else {
        int y = x;
        for (int i = 0; i < n; ++i)
            if (y == p[y])
                break;
        vector<int> Path;
        for (int cur = y; cur == p[cur];)
            path.push_back(cur);
        reverse(path.begin(), Path.end());
    }
}
```

```
cout << "Negative cycle:";  
for (size_t i =0; i <path.size(); + + I)
```

```
    cout << Path[i] << ' ';
```

```
}
```

The decision by the algorithm Floyd-Uorshella

Floyd's Algorithm-Uorshella solves the second formulation of the problem - when you have to find all pairs of vertices

(i, j) Between which there exists a shortest path (i.e., it has an infinitesimal amount).

Again, more detailed explanations contained in [description of the algorithm Floyd-](#)

[Uorshella](#) And here we present only the results.

After Floyd's algorithm-Uorshella work for the input graph, iterate over all pairs of vertices
for each such pair, check infinitesimal shortest path from to or not. To do this, let's look over the third
top, and if it has turned $d[t][t] < 0$ (ie, it lies in a cycle of negative weight), and she is achievable
and out of reach - the (i, j) path (i, j) could have infinitesimal length.

Implementation:

```
for (int i = 0; i <n; + + I)
    for (int j = 0; j <n; + + J)
        for (int t = 0; t <n; + + T)
            if (d[i] [t] <INF && d[t] [t] < 0 && D[t] [j]
                <INF) d[i] [j] =-INF;
```

Problem in online judges

List of tasks that require search cycle of negative weight:

- [UVA # 499 "Wormholes"](#) [Difficulty: low]

- [UVA # 104 "Arbitrage"](#) [Difficulty: average].

- [UVA # 10557 "XYZZY"](#) [Difficulty: average]

Finding Euler path for O (M)

Euler path - A path in the graph, passing through all its edges. Euler tour - is the Euler path is a cycle. The challenge is to find a way into the Euler **undirected multigraph with loops**.

Algorithm

Please check whether there is an Euler path. Then we find all simple cycles and combine them into one - it will be an Euler tour. If the graph is that the Euler path is not a cycle, then add the missing edge, find an Euler tour, then remove the extra edge.

To test whether there is an Euler path, you need to use the following theorem. Euler cycle exists if and only if the degrees of all vertices are even. Eulerian path exists if and only if the number of vertices equals the odd powers of two (or zero if there is Euler cycle).

Also, of course, the graph must be sufficiently connected (ie, if we remove from it all isolated vertices, then should get a connected graph).

Search all the loops and will combine them a recursive procedure:

```
procedure FindEulerPath (V)
    1. iterate over all edges emanating from the
       vertex V; each such edge is removed
       from the graph, and
       FindEulerPath call from the second end of the rib;
    2. add a vertex V in response.
```

The complexity of this algorithm is obviously linear in the number of edges. But the

same algorithm can be written in **non-recursive** version:

```
stack St;
in St put any vertex (top of page); while St
is not empty
    Let V - value on top of St; if
    the power (V) = 0, then
        add V to the answer;
        remove V from the top
        of St;
    otherwise
        find any edge going from V; remove
        it from the graph;
        the second end of the rib put in St;
```

It is easy to verify the equivalence of these two forms of the algorithm. However, the second shape is obviously faster running, the code will not be greater.

The problem of the domino

We present here a classic problem in the Euler cycle - the problem of dominoes.

Means N dominoes are known at both ends of dominoes recorded one number (typically 1 to 6, but in this case is not important). You want to publish all the dominoes in a row so that any two adjacent dominoes numbers written on their common side match. Dominoes are allowed to turn.

Reformulate the problem. Let the number recorded on donimoshkah - top graph, and the dominoes - edge of the graph (each domino with numbers (a, b) - this edge (a, b), and (b, a)). Then our task **reduces to** the problem of finding **Euler path** in this graph.

Implementation

The following program searches for and displays Euler cycle or path in the graph, or displays -1 if it does not exist.

First, the program checks the degree of vertices if the vertices with odd degree is not, then the graph has an Euler tour if there are two vertices with odd degree, then the graph has an Euler path only (no Euler cycle), if such heights greater than 2, then graph no Euler cycle or Euler path. To find an Euler path (not cycle), proceed as follows: if V1 and V2 - that two vertices of odd degree, then just add an edge (V1, V2), in the resulting graph will find an Euler tour (he obviously will exist), and then remove from the answer "fictitious" rib (V1, V2). Euler tour

will look exactly as described above (non-recursive version), and at the same time at the end of this algorithm check was connected graph or not (if the graph is not connected, then

termination of the algorithm, some remain in the column ribs, and in this case we must derive -1). Finally, the program takes into account that in the graph can be isolated vertices.

```
int main () {

    int n;
    vector <vector <int>> g (n, vector <int> (n));
    Reading ... graph adjacency matrix ...

    vector <int> deg (n);
    for (int i = 0; i <n; +
        + i)
        for (int j = 0; j <n; + +
            j) deg [i] += g
            [i] [j];

    int first = 0;
    while (! deg [first]) ++ first;

    int v1 = -1, v2 = -1;
    bool bad = false;
    for (int i = 0; i <n; +
        + i) if (deg
        [i] & 1)
            if (v1 == -1)
                v1 = i;
            else if (v2 == -1)
                v2 = i;
            else
                bad = true;

    if (v1! = -1)
        + + G [v1] [v2], + + g [v2] [v1];

    stack <int> st;
    st.push (first);
    vector <int> res;
    while (! st.empty
    ())
    {
        int v = st.top
        (); int i;
        for (i = 0; i <n; +
            + i) if (g
            [v] [i])
                break;
        if (i == n)
        {
            res.push_back (v);
            st.pop ();
        }
        else
        {
            - G [v] [i];
            - G [i] [v];
            st.push (i);
        }
    }

    if (v1! = -1)
        for (size_t i = 0; i +1 <res.size (); + + i)
            if (res [i] == v1 && res [i +1] == v2 || res [i] == v2
&& Res [i +1] == v1)
            {
                vector <int> res2;
                for (size_t j = i +1; j <res.size ();
                    + + j) res2.push_back (res
                    [j]);
                for (size_t j = 1; j <= i; + +
                    j) res2.push_back (res
                    [j]);
            }
    }
```

```
    res = res2;
    break;
}

for (int i = 0; i <n; + + i)
    for (int j = 0; j <n; +
        + j) if (g [i]
[j])
```

```
bad = true;  
  
if (bad)  
    puts ("-1");  
else  
    for (size_t i = 0; i < res.size ();  
         + + i) printf ("% d", res  
                      [i] +1);  
}
```

Checking on the acyclic graph and finding cycle

Let a directed or undirected graph without loops and multiple edges. Want to check whether it is acyclic, and if not, then find any cycle.

Solve this problem by using [DFS](#) in $O(M)$.

Algorithm

We carry out a series of searches in depth in the graph. Ie from each vertex, which we had never come, run dfs that at the entrance to the top will paint it gray, and on leaving - in black. And if dfs tries to go into a gray top, it means that we have found a cycle (if undirected graph, the cases where the depth-first search of some vertex tries to go to the parent are not considered). The cycle itself can be restored over an array of ancestors.

Implementation

Here is an implementation for the case of a directed graph.

```
int n;
vector <vector <int>> g;
vector <char> cl; vector
<int> p;
int cycle_st, cycle_end;

bool dfs (int v)
    {cl [v] =
     1;
    for (size_t i = 0; i <g [v]. size ();
         + + i) {int to = g [v] [i];
              if (cl [to] == 0)
                  {p [to] =
                   v;
                    if (dfs (to)) return true;
                  }
              else if (cl [to] == 1)
                  {cycle_end = v;
                   cycle_st = to;
                   return true;
                  }
            }
    cl [v] = 2;
    return false;
}

int main () {
    Graph reading ... ...

    p.assign (n, -1);
    cl.assign (n, 0);
    cycle_st = -1;
    for (int i = 0; i <n; +
         + i) if (dfs
                  (i))
                     break;

    if (cycle_st == -1)
        puts ("Acyclic");
    else {
        puts ("Cyclic"); vector
        <int> cycle;
        cycle.push_back (cycle_st);
        for (int v = cycle_end; v! = cycle_st; v =
             p [v]) cycle.push_back (v);
        cycle.push_back (cycle_st);
```

```
reverse (cycle.begin (), cycle.end
()); for (size_t i = 0; i <cycle.size
(); ++ i)
```

```
    printf ("%d", cycle [i] +1);  
}  
}
```

Lowest common ancestor. For finding O (sqrt (N)) and O (log N) preprocessing with O (N)

Suppose we are given a tree G. The input receives requests form (V1, V2), for each request is required to find their least common ancestor, ie vertex V, which lies on the path from the root to the V1, the path from the root to the V2, and from all such vertices should choose the lowest. In other words, the required vertex V - ancestor and V1, and V2, and among all such common ancestors selected lower. It is obvious that the lowest common ancestor of vertices V1 and V2 - it is their common ancestor, which lies on the shortest path from V1 to V2. In particular, for example, if V1 is the ancestor of V2, then V1 is their least common ancestor.

In English, this problem is called LCA - Least Common Ancestor.

Idea of the algorithm

Before responding, perform the so-called **preprocessing**. Start the tour in depth from the root, which will build a list of vertices visit Order (current node is added to the list at the entrance to this mountain, and after each return of her son), you will notice that the final size of the list is O (N). And build an array of First [1 .. N], in which each vertex is specified position in the array Order, in which there is such a vertex, ie Order [First [I]] = I for all I. Also, using the depth-first search will find the height of each vertex (distance from the root to it) - H [1 .. N].

As it is now responding? Let there be the current request - a pair of vertices V1 and V2. Consider the list of Order between indices First [V1] and First [V2]. It is easy to notice that in this range will be required and LCA (V1, V2), as well as many other peaks. However, LCA (V1, V2) is different from the other peaks in that it will peak with the lowest height.

Therefore, to answer the query, we simply **find the vertex with the lowest height** Order array comprised between First [V1] and the First [V2]. Thus, **task LCA reduces to RMQ problem** ("Minimum interval"). And the last problem is solved by data structures (Cf. task [RMQ](#)).

If you use **sqrt-decomposition** It is possible to obtain a solution that responds to the request for **O (sqrt (N))** and performs preprocessing for the **O (N)**.

If you use **segment tree** It is possible to obtain a solution that responds to the request for **O (log (N))** and performs preprocessing for the **O (N)**.

Implementation

Here will be given ready LCA implementation using wood segments:

```
typedef vector <vector <int>> graph;
typedef vector <int> :: const_iterator const_graph_iter;

vector <int> lca_h, lca_dfs_list, lca_first,
lca_tree; vector <char> lca_dfs_used;

void lca_dfs (const graph & g, int v, int h = 1)
{
    lca_dfs_used [v] = true;
    lca_h [v] = h;
    lca_dfs_list.push_back (v);
    for (const_graph_iter i = g [v]. begin (); i != g [v]. end
        (); ++ i) if (! lca_dfs_used [* i])
    {
        lca_dfs (g, * i, h +1);
        lca_dfs_list.push_back (v);
    }
}

void lca_build_tree (int i, int l, int r)
{
    if (l == r)
        lca_tree [i] = lca_dfs_list [l];
    else
    {
        int m = (l + r) >> 1;
```

```

        lca_build_tree (i + i, l, m);
        lca_build_tree (i + i +1, m +1,
        r);
        if (lca_h [lca_tree [i + i]] <lca_h [lca_tree [i +
        i +1]]) lca_tree [i] = lca_tree [i + i];
        else
            lca_tree [i] = lca_tree [i + i +1];
    }
}

void lca_prepare (const graph & g, int root)
{
    int n = (int) g.size ();
    lca_h.resize (n);
    lca_dfs_list.reserve (n *
2); lca_dfs_used.assign (n,
0);
    lca_dfs (g, root);

    int m = (int) lca_dfs_list.size ();
    lca_tree.assign (lca_dfs_list.size () * 4 + 1, -
1); lca_build_tree (1, 0, m-1);
    lca_first.assign (n, -1);
    for (int i = 0; i <m; + +
i)
    {
        int v = lca_dfs_list [i];
        if (lca_first [v] == -1)
            lca_first [v] = i;
    }
}

int lca_tree_min (int i, int sl, int sr, int l, int r)
{
    if (sl == l && sr == r)
        return lca_tree
        [i];
    int sm = (sl + sr) >> 1;
    if (r <= sm)
        return lca_tree_min (i + i, sl, sm, l,
r); if (l> sm)
        return lca_tree_min (i + i +1, sm +1, sr, l,
r); int ans1 = lca_tree_min (i + i, sl, sm, l, sm);
    int ans2 = lca_tree_min (i + i +1, sm +1, sr, sm
+1, r); return lca_h [ans1] <lca_h [ans2]? ans1:
ans2;
}

int lca (int a, int b)
{
    int left = lca_first [a],
        right = lca_first
        [b];
    if (left> right) swap (left, right);
    return lca_tree_min (1, 0, (int) lca_dfs_list.size () -1, left, right);
}

int main ()
{
    graph g;
    int root;
    Graph reading ... ...
    lca_prepare (g, root);
    for (; ;)

```

```
{  
    int v1, v2; // Received a request  
    int v = lca (v1, v2); // Response to a request  
}
```

Lowest common ancestor. Being in O (log N) (binary lifting method)

Suppose we are given a tree G. The input receives requests form (V1, V2), for each request is required to find their least common ancestor, ie vertex V, which lies on the path from the root to the V1, the path from the root to the V2, and from all such vertices should choose the lowest. In other words, the required vertex V - ancestor and V1, and V2, and among all such common ancestors selected lower. It is obvious that the lowest common ancestor of vertices V1 and V2 - it is their common ancestor, which lies on the shortest path from V1 to V2. In particular, for example, if V1 is the ancestor of V2, then V1 is their least common ancestor.

In English, this problem is called LCA - Least Common Ancestor.

This algorithm will be discussed, which is written much faster than the one described [here](#).

Asymptotics of this algorithm will be equal for preprocessing **O (N log N)** and the response to each request for **O (Log N)**.

Algorithm

Predposchitaem for each vertex of its ancestor 1st, 2nd ancestor, 4th, etc. Denote the array through P, ie P [i] [j] - it 2j-th ancestor vertex i, i = 1 .. N, j = 0 .. $\lfloor \log_2 N \rfloor$. Also, for each vertex find sunset times in and output depth-first search (see "Depth-First Search") - That we need to determine in O (1) whether one vertex ancestor of the other (not necessarily direct). This preprocessing can be done in **O (N log N)**.

Suppose now entered another request - a pair of vertices (A, B). Immediately check whether one vertex is an ancestor of the other - in this case, it is the result. If A is not an ancestor of B, and B is not an ancestor of A, then we climb the ancestors A, until we find the highest (ie, closest to the root) of the vertex, which is still

is not an ancestor (not necessarily direct) B (ie, a vertex X, that X is not an ancestor of B, and P [X] [0] - the ancestor of B). At the same time finding the vertex X is in O ($\log N$), using an array of P.

We describe this process in more detail. Let $L = \lfloor \log_2 N \rfloor$. Suppose first that $I = L$. If P [A] [I] is not an ancestor of B, then assign $A = P [A] [I]$, and reduce I. If P [A] [I] is an ancestor of B, then just reduce I. Obviously, when I would be less than zero, the vertex A just and will be required vertex - ie such that A is not an ancestor of B, but P [A] [0]

- Ancestor of B.

Now, obviously, the answer to the LCA will be P [A] [0] - ie smallest vertex among the ancestors of the original vertices A, which is also the ancestor of B.

Asymptotics. The whole algorithm is responding to a request from the change of $I \leftarrow L = \lfloor \log_2 N \rfloor$ to 0, and check each step in O (1) whether one vertex is an ancestor of the other. Consequently, for each query answer is found in O ($\log N$).

Implementation

```
int n, l;
vector <vector <int>> g;
vector <int> tin, tout;
int timer;
vector <vector <int>> up;

void dfs (int v, int p = 0)
    {tin [v] = + + timer;
    up [v] [0] = p;
    for (int i = 1; i <= l; + + i)
        up [v] [i] = up [up [v] [i-1]]
[i-1]; for (size_t i = 0; i < g [v].
size (); + + i) {
            int to = g [v]
[i]; if (to! = p)
                dfs (to, v);
        }
    tout [v] = + + timer;
}

bool upper (int a, int b) {
    return tin [a] <= tin [b] && tout [a]> = tout [b];
}

int lca (int a, int b) {
```

```
if (upper (a, b)) return a;
if (upper (b, a)) return b;
for (int i = l; i> = 0; - i)
    if (! upper (up [a] [i],
                 b)) a = up [a]
                     [i];
return up [a] [0];
}

int main () {

    Reading ... n and g ...

    tin.resize (n), tout.resize (n), up.resize (n); l
    = 1;
    while ((l << 1) <= n) + + l;
    for (int i = 0; i <n; + + i) up [i]. resize
        (l +1); dfs (0);

    for (; ; ) {
        int a, b; / / Current request
        int res = lca (a, b); / / Response to a
        request
    }
}
```

Lowest common ancestor. Finding the O (1) with preprocessing O (N) (algorithm Farah-Colton and Bender)

Suppose we are given a tree G. The input receives requests form (V1, V2), for each request is required to find their least common ancestor, ie vertex V, which lies on the path from the root to the V1, the path from the root to the V2, and from all such vertices should choose the lowest. In other words, the required vertex V - ancestor and V1, and V2, and among all such common ancestors selected lower. It is obvious that the lowest common ancestor of vertices V1 and V2 - it is their common ancestor, which lies on the shortest path from V1 to V2. In particular, for example, if V1 is the ancestor of V2, then V1 is their least common ancestor.

In English, this problem is called LCA - Least Common Ancestor.

Farah algorithm described here-Colton and Bender (Farach-Colton, Bender) is asymptotically optimal, and thus a relatively simple (compared to other algorithms, for example, gate-Vishkina).

Algorithm

Use the classical reducing the problem of LCA [to the problem RMQ](#) (Minimum interval) (for more details see [Lowest common ancestor. Finding for O \(sqrt \(N\)\) and O \(log N\) with preprocessing O \(N\)](#)). Now learn to

RMQ solve the problem in this particular case with preprocessing O (N) and O (1) on request.

Note that the problem RMQ, to which we have reduced the problem LCA, is very specific: any two adjacent elements in the array **differ by exactly one** (As array elements - it is nothing like the height of vertices visited in the traversal, and we either go to the descendant, then the next item will be 1 more or go to the ancestor, then the next item will be 1 less). Actually algorithm Farah-Colton and Bender just is a solution of this problem RMQ.

Let A array over which queries are running RMQ, and N - size of the array.

We first construct an algorithm that solves this problem **with preprocessing O (N log N) and O (1) on inquiry**. It's easy: create a so-called Sparse Table T [l, i], where each element T [l, i] equal to the minimum A on the interval [l; l + 2ⁱ]. Obviously, 0 <= i <= ⌊log N⌋, and therefore the size of Sparse Table is O (N log N). Build it also easy for the O (N log N), if we note that T [l, i] = min (T [l, i-1], T [l + 2ⁱ⁻¹, i-1]). How now to respond to every request RMQ O (1)? Let received a request (l, r), then the answer would be min (T [l, sz], T [r-2sz + 1, sz]), where sz - the largest power of two not exceeding r-l + 1. Indeed, we seem to take the interval (l, r) and cover it with two runs of length 2sz - one starting at l, and the other ending in r (and these segments overlap, which in this case does not bother us). To really achieve the asymptotics O (1) on the request, we must predposchitat sz values for all possible lengths from 1 to N.

We will now describe **to improve** this algorithm to the asymptotic behavior of O (N).

We divide the array A into blocks of size K = 0.5 log2 N. For each block, calculate the minimum element in it and his position (as for the solution of LCA are important to us not the lows, and their positions). Let B - is an array of size N / K, composed of these minima in each block. Construct the array B Sparse Table, as described above, the size of Sparse Table and time of its construction will be:

$$\begin{aligned} N / K \log N / K &= (2N / \log N) \log (2N / \log N) = \\ &= (2N / \log N) (1 + \log (N / \log N)) \leq 2N / \log N + 2N = O (N) \end{aligned}$$

Now we just need to learn how to quickly respond to requests RMQ **within each block**. In fact, if the request comes RMQ (l, r), then if r and l are located in different blocks, the answer will be a minimum of the following values: at least a block l, starting from l to the end of the block, then the minimum units after l to r and (Not included), and finally the minimum block r, from the beginning of the block to r. At the prompt "at least in the" we can be responsible for O (1) using the Sparse Table, there were only questions RMQ in blocks.

Here we use the "+ - 1 property." Note that, if within each block of each of its elements take the first element, then all blocks will be uniquely determined by the sequence of length K-1, consisting of the numbers + - 1. Consequently, the amount of the various blocks will be:

$$2K-1 = 20.5 \log N - 1 = 0.5 \sqrt{N}$$

Thus, the number of different blocks will be O (sqrt (N)), and therefore we can predposchitat results inside RMQ all the different blocks in O (sqrt (N) K2) = O (sqrt (N) log2 N) = O (N). From an implementation standpoint, we can characterize each block bitmask length K-1 (which obviously fits into standard type int), and store predposchitannye RMQ from an array R [mask, l, r] size O (sqrt (N) log2 N).

So, we have learned predposchityvat RMQ results within each block, as well as by the RMQ over blocks all for a total of O (N), and to respond to every request RMQ O (1) - using only the precomputed values in the worst case four: in block l, in block r, and on the blocks between l and r is not inclusive.

Implementation

At the beginning of the program are constants MAXN, LOG_MAXLIST and SQRT_MAXLIST, determine the maximum number of vertices in the graph, which, if necessary, should be increased.

```

const int MAXN = 100 * 1000;
const int MAXLIST = MAXN * 2;
const int LOG_MAXLIST = 18;
const int SQRT_MAXLIST = 447;
const int MAXBLOCKS = MAXLIST / ((LOG_MAXLIST +1) / 2) + 1;

int n, root; vector<int> g [MAXN];
int h [MAXN]; // Vertex height vector <int> a; //
Dfs list
int a_pos [MAXN]; // Positions in dfs list
int block; // Block size = 0.5 log A.size()
()
int bt [MAXBLOCKS] [LOG_MAXLIST +1]; // Sparse table on blocks
(relative minimum positions in blocks)
int bhash [MAXBLOCKS]; // Block hashes
int brmq [SQRT_MAXLIST] [LOG_MAXLIST / 2] [LOG_MAXLIST / 2]; // Rmq
inside each block, indexed by block hash
int log2 [2 * MAXN]; // Precalced logarithms (floored values)

// Walk graph
void dfs (int v, int curh)
{
    h [v] = curh;
    a_pos [v] = (int) a.size();
    a.push_back (v);
    for (size_t i = 0; i < g [v].size();
        ++ i) if (h [g [v] [i]] == -1) {
        dfs (g [v] [i], curh +1); a.push_back (v);
    }
}

int log (int n) {int res = 1;
    while (1 << res <n) ++
    res; return res;
}

// compares two indices in a
inline int min_h (int i, int j) {
    return h [a [i]] < h [a [j]]? i: j;
}

// O (N) preprocessing
void build_lca () {
    int sz = (int) a.size ();
    block = (log (sz) + 1) /
    2;
    int blocks = sz / block + (sz % block? 1: 0);

    // Precalc in each block and build sparse
    table memset (bt, 255, sizeof bt);
    for (int i = 0, bl = 0, j = 0; i < sz; ++
        i, ++ j) {if (j == block)
        j = 0, ++ bl;
        if (bt [bl] [0] == -1 || min_h (i, bt [bl] [0])
            == i) bt [bl] [0] = i;
    }
    for (int j = 1; j <= log (sz); ++ j)

```

```
for (int i = 0; i <blocks; + +
    i) {int ni = i + (1 <<
(j-1)); if (ni> =
blocks)
    bt [i] [j] = bt [i] [j-1];
```

```

        else
            bt [i] [j] = min_h (bt [i] [j-1], bt [ni] [j-1]);
    }

    / / Calc hashes of blocks
    memset (bhash, 0, sizeof bhash);
    for (int i = 0, bl = 0, j = 0; i <sz | | j <block;
        + + i, + + j) {if (j == block)
        j = 0, + + bl;
    if (j> 0 && (i>= sz | | min_h (i-1, i) == i-1))
        bhash [bl] += 1 << (j-1);
}

/ / Precalc RMQ inside each unique
block memset (brmq, 255, sizeof brmq);
for (int i = 0; i <blocks; + +
i) {int id = bhash
[i];
if (brmq [id] [0] [0] != -1)
continue; for (int l = 0; l <block;
+ + l) {
    brmq [id] [l] [l] = l;
    for (int r = l +1; r <block; + + r) {brmq
[id] [l] [r] = brmq [id] [l] [r-
1]; if (i * block + r <sz)
    brmq [id] [l] [r] =
        min_h (i * block + brmq
[id] [l]
[R], i * block + r) - i * block;
    }
}
}

/ / Precalc logarithms
for (int i = 0, j = 0; i <sz; +
+ i) {if (1 << (j +1) <=
i) + + j; log2 [i] = j;
}

// answers RMQ in block # bl [l; r] in O (1)
inline int lca_in_block (int bl, int l, int r) {
    return brmq [bhash [bl]] [l] [r] + bl * block;
}

// answers LCA in O (1)
int lca (int v1, int v2) {
    int l = a_pos [v1], r = a_pos
[v2]; if (l> r) swap (l, r);
    int bl = l / block, br = r /
block; if (bl == br)
        return a [lca_in_block (bl, l% block, r%
block)]; int ans1 = lca_in_block (bl, l% block,
block-1);
    int ans2 = lca_in_block (br, 0, r%
block); int ans = min_h (ans1, ans2);
    if (bl <br - 1) {
        int pw2 = log2 [br-bl-1];
        int ans3 = bt [bl +1]
[pw2];
        int ans4 = bt [br-(1 << pw2)] [pw2];
        ans = min_h (ans, min_h (ans3, ans4));
    }
    return a [ans];
}

```

Problem RMQ (Range Minimum Query - at least in the interval). Solution in O (1) with preprocessing O (N)

Given an array A [1 .. N]. Receives requests form (L, R), for each query requires a minimum in the array A, starting at position L and ending at R. Array A change in the process can not, ie The solution described here static problem RMQ.

Described herein asimptoticheski optimal solution. It stands apart from several other algorithms for solving the RMQ, because it is very different from them: it reduces the problem to the RMQ problem LCA, and then uses [Farah algorithm-Colton](#) and [Bender](#)Which reduces the problem back to the LCA RMQ (but a particular form) and solves it.

Algorithm

Construct the array A Cartesian tree where each node is a key position i, and priority - the sheer number of A [i] (it is assumed that in the Cartesian tree priorities are ordered from smaller to larger radically). Such tree can be constructed in O (N). Then inquiry RMQ (l, r) is equivalent to the request LCA (l', r'), where l' - the vertex corresponding to the element A [l], r' - corresponding to the A [r]. Indeed, LCA find a vertex which is the key lies between l', r', i.e. by position in the array A will be between l and r, and wherein the vertex closest to the top, i.e. with the lowest priority, ie the lowest value.

LCA problem we can solve in O (1) preprocessing with O (N) using [Farah algorithm-Colton](#) and [Bender](#)Which, interestingly, reduces the problem back to the LCA problem RMQ, but a special form.

Lowest common ancestor. Finding for $O(1)$ offline (Tarjan algorithm)

Given tree with vertices and given form queries of the required
find the lowest common ancestor of vertices and that a vertex, which is farthest from the root, required
and this is the ancestor of both peaks and .
We consider the problem in the offline mode, ie Considering that all requests are known in advance. Described
following algorithm allows to answer all requests for the total time $O(n + m)$ ie at sufficiently
large m $O(1)$ on request.

Tarjan's algorithm

The basis for the algorithm is a data structure "The system of disjoint sets" Which was invented Tarjanne (Tarjan).

Algorithm is actually a detour into the depths of the root of the tree, in the which are gradually responding to requests. Namely, the answer to the query is when traversal depth is at the top (v, u)
 u And the apex v already has had, or vice versa.

So let traversal depth is at the top v (And have already been implemented in the transitions of her sons), and found that for some query (v, u) vertex already has had a circuit in depth. Then learn how to find LCA

Note that $LCA(v, u)$ is either the very top, or one of its ancestors. So, we need to find the lowest vertex of ancestors (including herself), for which the peak u is a descendant.
Note that for fixed on such grounds, what the lowest ancestor v and what is the ancestor-
the vertices) tree top tree split into a set of disjoint classes. For each ancestor $\neq v$ vertices v her class contains the vertex itself and all subtrees with roots in those of its sons, who are "left" of the way up v (i.e., that have been processed earlier than was achieved v).

We need to learn how to effectively support all of these classes, for which we apply the data structure "system of disjoint sets." Each class will meet in the structure set, and ANCESTOR
representative for this set, we define the quantity - that top p , Which defines this class.

We consider in detail the implementation of traversal depth. Suppose we are at some vertex . Place it in the separate class in the structure of disjoint sets, . As usual in the bypass depth, we sort all outgoing edges . For each such we first need to call the traversal depth of this peak, and then add the top with all its subtree in the class top. It operation is realized data structures, "a system of disjoint sets", followed by v installation Union for a representative set (after the merger because the class representative could change). Finally, after processing all the edges, we iterate through all the queries of the And if there form (v, u) was u marked as visiting a circuit in depth, the response to this request will be top $LCA(v, u) = \text{ANCESTOR}[\text{FindSet}(u)]$. It is easy to see that for each request this condition (that one vertex is the current request, and the other has had before) is executed exactly once.

We estimate the **asymptotics**. It consists of several parts. First, it is the asymptotic behavior of traversal depth, which in this case is. Secondly, this business combination $O(n)$ amounting to expend all reasonable $O(n)$ operations. Thirdly, it is for each request verification conditions (Twice on request) and the definition of the result (once at your request), each, again, for all reasonable n performed for $O(1)$. Total asymptotics obtained $O(n + m)$ That is sufficiently large m ($n = O(m)$) Response for $O(1)$ one request.

Implementation

We present a complete implementation of this algorithm, including a slightly modified (with support ANCESTOR) implementation of disjoint sets (randomized version).

```
const int MAXN = maximum number of vertices in the
graph; vector <int> G[MAXN], Q[MAXN]; / graph and
all requests int dsu[MAXN], Ancestor[MAXN];
bool u[MAXN];
```

```

int dsu_get (int v) {
    return v == dsu[v] ? v: dsu[v] = Dsu_get (dsu[v]);
}

void dsu_unite (int a, int b, int new_ancestor) {
    a = dsu_get (a), B = dsu_get (b);
    if (rand() & 1) swap (a, b);
    dsu[a] = B, ancestor[b] = New_ancestor;
}

void dfs (int v) {
    dsu[v] = V, ancestor[v] = V;
    u[v] = true;
    for (size_t i =0; i <g[v].Size(); +
        + I) if (! U[g[v][i]]) {
        dfs (g[v][i]);
        dsu_unite (v, g[v][i], V);
    }
    for (size_t i =0; i <q[v].Size(); +
        + I) if (u[q[v][i]]) {
        printf ("% D% d ->% d \ n", V +1, Q[v]
            [i]+1, ancestor[ dsu_get(q[v]
            [i])]+1);
    }
}

int main() {
    Graph reading ... ...

    // read requests
    for (,; ) {
        int a, b = .... / / another request
        - A, - b; q[a].
        Push_back (b);
        q[b]. Push_back
        (a);
    }

    // traversal depth and response to requests
    dfs (0);
}

```

Maximum flow Edmonds-Karp method for O (N M2)

Let a graph G, which revealed two peaks: the source S and drain T, and each edge is defined bandwidth C_{uv} , v . The flow F may be represented as a stream of a substance which could pass through the network from source to drain when viewed as a network pipe graph with certain bandwidth. Ie Feed - function F_u ,

Defined on the set of edges.

The task is to find the maximum flow. It will discuss the method of Edmonds-Karp, who works for the O (N M2), or (another estimate) O (FM), where F - the value of the desired flow. Algorithm was proposed in 1972.

Algorithm

Residual bandwidth called bandwidth edges minus current flow along this edge. It must be remembered that if a stream flows through the oriented edge, then the so-called inverse edge (directed in the opposite direction), which will have zero capacity, and which will take place the same value on the stream, but with a minus sign. If the rib was unoriented, as it would be oriented into two edges with the same bandwidth, and each of these edges is the inverse of the other (if one proceeds flow F, then proceeds differently-F).

General schematic **Edmonds-Karp algorithm** is as follows. Please believe the flow is zero. Then look for complementary way, ie simple path from S to T on those edges whose residual bandwidth strictly positive. If supplementing the way had been found, then zooms the current flow along this path. If the same path is not found, then the current flow is highest. For supplementing the search path can be used as [Crawling in a width](#) And [Crawling in depth](#).

Consider a more accurate procedure to increase the flow. Suppose we found some complementary way, then let C - the smallest of the residual capacities of the edges of this path. The procedure is to increase the flow as follows: for each edge (u, v) perform complementary ways: $F_u, v + = C$, and $F_v, u = -F_u, v$ (or, equivalently, $F_v, u - = C$).

The flux is the sum of all non-negative values of F_{Sv} , where v - any vertex connected to the source.

Implementation

```
const int inf = 1000 * 1000 * 1000;

typedef vector <int> graf_line;
typedef vector <graf_line>
graf;
typedef vector <int> vint;
typedef vector <vint>
vvint;

int main ()
{
    int n;
    cin >> n;
    vvint c (n, vint (n));
    for (int i = 0; i <n; i
        + +)
        for (int j = 0; j <n;
            j++) cin>> c
            [i] [j];
    // Source - vertex 0, Stock - top n-1

    vvint f (n, vint
    (n)); for (; ; )
    {

        vint from (n, -1);
        vint q (n);
        int h = 0, t = 0;
```

```

q [t + +] =
0; from [0]
= 0;
for (int cur; h <t ;)
{
    cur = q [h + +];
    for (int v = 0; v <n; v + +)
        if (from [v] == -1 && c [cur]
            [v]-f [cur] [v]> 0)
        {
            q [t + +] = v;
            from [v] =
            cur;
        }
}

if (from [n-1] == -
    1) break;
int cf = inf;
for (int cur = n-1; cur! = 0;)
{
    int prev = from [cur];
    cf = min (cf, c [prev] [cur]-f [prev]
    [cur]); cur = prev;
}

for (int cur = n-1; cur! = 0;)
{
    int prev = from
    [cur]; f [prev] [cur]
    += cf; f [cur]
    [prev] -= cf; cur =
    prev;
}

int flow = 0;
for (int i = 0; i <n; i
    + +) if (c [0]
    [i])
    flow += f [0] [i];

cout << flow;
}

```

Maximum flow by pushing predpotoka

in O (N4)

Let a graph G, which revealed two peaks: the source S and drain T, and each edge is defined bandwidth $C_{u,v}$. The flow F may be represented as a stream of a substance which could pass through the network from source to drain when viewed as a network pipe graph with certain bandwidth. Ie Feed - function F_u ,

Defined on the set of edges.

The task is to find the maximum flow. It will discuss the method of pushing predpotoka working at O (N4), or, more precisely, for the O (N2 M). The algorithm was proposed by Goldberg in 1985.

Algorithm

The general scheme of the algorithm is as follows. At each step, we will consider some predpotok - ie function that resembles the properties of the flow, but not necessarily satisfy the law of conservation of flux. At each step, we try to apply any of the two operations: pushing or lifting the top of the flow. If at some stage it will be impossible to use any of the two operations, we have found the required flow.

Each vertex is defined by its height H_u , and $HS = N$, $HT = 0$, and for any remaining edge (u, v) have H_u

$\leq H_v + 1$.

For each vertex (except S) can determine its excess: $E_u = F_v - u$. Top with positive excess is called crowded.

Operation push Push (u, v) is applicable if the vertex u is full, the residual bandwidth $C_{u,v} > 0$ and $H_u = H_v + 1$. Operation push is to maximize the flow from u to v ,

E_u and limited excess residual bandwidth $C_{u,v}$.

Operation lift Lift (u) raises crowded vertex u to the maximum allowable height. Ie $H_u = 1 + \min\{H_v\}$, where (u, v) - the residual rib.

It remains only to consider the initialization stream. It is necessary to initialize only the following values:

$FS, v = CS, v, Fu, S = - Cu, S$, the remaining values equal to zero.

Implementation

```
const int inf = 1000 * 1000 * 1000;

typedef vector<int> graf_line;
typedef vector<graf_line>
graf;
typedef vector<int> vint;
typedef vector<vint>
vvint;

void push (int u, int v, vvint & f, vint & e, const vvint & c)
{
    int d = min (e [u], c [u] [v] - f [u]
    [v]); f [u] [v] += d;
    f [v] [u] = - f [u]
    [v]; e [u] -= d;
    e [v] += d;
}

void lift (int u, vint & h, const vvint & f, const vvint & c)
{
    int d = inf;
    for (int i = 0; i <(int) f.size (); i +
    ) if (c [u] [i]-f [u] [i]> 0)
        d = min (d, h [i]);
```

```

        if (d == inf)
            return;
        h [u] = d + 1;
    }

int main ()
{
    int n;
    cin >> n;
    vvint c (n, vint (n));
    for (int i = 0; i <n; i
        +++)
        for (int j = 0; j <n;
            j++) cin>> c
                [i] [j];
    // Source - vertex 0, Stock - top n-1

    vvint f (n, vint (n));
    for (int i = 1; i <n; i
        +++)
    {
        f [0] [i] = c [0]
            [i]; f [i] [0] =-c
            [0] [i];
    }

    vint h (n);
    h [0] = n;

    vint e (n);
    for (int i = 1; i <n; i
        +++) e [i] = f
            [0] [i];

    for (;;) {
        int i;
        for (i = 1; i <n-1;
            i++) if (e[i]> 0)
                break;

        if (i == n-1)
            break;

        int j;
        for (j = 0; j <n; j + +)
            if (c [i] [j]-f [i] [j]> 0 && h [i] == h
                [j] +1) break;
        if (j <n)
            push (i, j, f, e, c);
        else
            lift (i, h, f, c);
    }

    int flow = 0;
    for (int i = 0; i <n; i
        +++) if (c [0]
            [i])
        flow += f [0]
            [i]; cout << max (flow, 0);
}

```

Modification of the method for finding the push predpotoka maximum flow in O (N3)

It is assumed that you have already read [Method push predpotoka for finding the maximum flow O \(N4\)](#).

Description

Modification is very simple: at each iteration among all crowded vertices we select only those vertices that have **Greatest height** and apply a pushing / lifting only to those heights. Moreover, to select nodes with a maximum height we do not need any data structures, just keep a list of nodes with a maximum height and just recalculate it, if all the vertices of this list have been processed (then the list will be added to the top of already lower height), and when new tops crowded with greater height than the list, clear the list and add the top of the list.

Despite its simplicity, this modification reduces the asymptotic behavior of the whole order. To be precise asymptotic behavior of the algorithm is received $O(NM + N^2 \sqrt{M})$. That is, in the worst case $O(N^3)$.

This modification was proposed Cheriyanom (Cheriyan) and Maheshwari (Maheshvari) in 1989

Implementation

Here is a ready-made implementation of this algorithm.

Unlike conventional algorithm push - only available array maxh, which will be stored crowded rooms with a maximum height of vertices. The array size is specified in the variable sz. If at some iteration is that the array is empty ($sz == 0$), then we fill it (just passing through all the vertices); if after this array is still empty, the vertices not crowded, and the algorithm stops. Otherwise, we go over the tops in the list applying to them pushing or lifting. After the operation, push the current node may cease to be crowded, in this case, remove it from the list maxh. If, after some operations lifting height of the current node becomes larger than the height of vertices in the list maxh, then we clear the list ($sz = 0$), and immediately proceed to the next iteration of pushing (which will be built on the new list maxh).

```
const int INF = 1000 * 1000 * 1000;

int main () {

    int n;
    vector <vector <int>> c (n, vector <int> (n));
    int s, t;
    Reading ... n, c, s, t ...

    vector <int> e
    (n); vector <int>
    h (n); h [s] = n-
    1;
    vector <vector <int>> f (n, vector <int> (n));

    for (int i = 0; i <n; + +
        i) {f [s] [i] = c
            [s] [i]; f [i] [s]
            =-f [s] [i]; e [i]
            = c [s] [i];
    }

    vector <int> maxh
    (n); int sz = 0;
    for (; ; ) {
        if (! sz)
            for (int i = 0; i <n; + + i)
                if (i! = s && i! = t && e [i]> 0)
                    {if (sz && h [i]> h [maxh
                        [0]])
                        sz = 0;
                     if (! sz | | h [i] == h [maxh
                        [0]]) maxh [sz + +] =
                        i;
```

```

        }
        if (! sz) break;
        while (sz) {
            int i = maxh [sz-1];
            bool pushed = false;
            for (int j = 0; j <n && e [i]; + + j)
                if (c [i] [j]-f [i] [j]> 0 && h [i] == h [j]
                    +1) {pushed = true;
                int addf = min (c [i] [j]-f [i] [j], e [i]);
                f [i] [j] += addf, f [j] [i] -= addf;
                e [i] -= addf, e [j] += addf;
                if (e [i] == 0) - sz;
            }
            if (! pushed) {
                h [i] = INF;
                for (int j = 0; j <n; + + j)
                    if (c [i] [j]-f [i] [j]> 0 && h [j] +1
                        <
                        h [i])
                        h [i] = h [j]
                    +1; if (h [i]> h [maxh [0]]) {
                        sz = 0;
                        break;
                    }
            }
        }
    }

Output flow ... f ...
}

```

Finding the flow in a graph in which each edge indicate the minimum and maximum flow

Let a graph G, where each edge in addition to the capacity (maximum flow along this edge) and a minimum value specified flow to pass along this edge.

Here we consider two problems: 1) the need to find an arbitrary stream of satisfying all constraints, and 2) requires a minimum flow satisfying all constraints.

Solution of 1

We denote the minimum value of L_i flow that can pass through the i -th edge, and by R_i - its maximum value.

Made in the following graph **changes**. Add a new source S 'and stock T'. Consider all the edges, which L_i is nonzero. Let i - the number of such edges. Let the ends of the rib (oriented) - the vertices A_i and B_i . Add an edge (S', B_i) , which $L = 0$, $R = L_i$, add an edge (A_i, T') , in which $L = 0$, $R = L_i$, and at most i -th ribs put $R_i = R_i - L_i$, and $L_i = 0$. Finally, we add to the graph edge from T to S (old drain and source) that $L = 0$, $R = \text{INF}$.

After these conversions all edges will have $L_i = 0$, ie we reduced this problem to the usual

the problem of finding the maximum flow (but in a modified graph with the new source and the drain) (to understand why the maximum - read the following explanation).

Correctness of these transformations is more difficult to understand. Informal **explanation** such. Each edge which L_i is nonzero, we replace two edges: one with a capacity of L_i , and the other - with $R_i - L_i$. We need find a thread that would definitely satiated the first edge of the pair (ie, the flow along this edge must be equal to L_i); second edge we care less - flow along it can be anything, as long as it does not exceed its bandwidth. So we need to find a thread that would definitely satiated a set of edges. Consider each such edge, and perform such an operation: sum to its end edge from the new source S' , sum up the edge of his start to the drain of T' , remove the rib itself, and from the old to the old Photo T the source S will hold an edge of infinite bandwidth. With these actions, we proimitiruem the fact that this edge is saturated - the rib will follow L_i flow units (we simulate it using a new source that delivers at the end of the ribs desired flow amount), and will flow into it again L_i flow units (but instead of ribs this thread gets a new stock). Flow from the source of the new flows through one portion of the graph to the old dotekaet Photo T, it flows from the source to the old S, then flows through another portion of the graph, and finally comes to the beginning of this rib and into the new sink T' . That is, if we find in this modified graph maximum flow (and the drain gets the right amount of flow, ie the sum of all the values of L_i - otherwise, the flux will be smaller and the response simply does not exist), we simultaneously find the flow in the original graph, which will satisfy all the minimum restrictions, and, of course, all the limitations of the maximum.

Solution of 2

Note that along the edge of the old drain in old source with bandwidth INF flows all the old stream, ie, the bandwidth of the edge effect on the magnitude of the old stream. For sufficiently large values of the bandwidth of the edge (ie, INF) old flow is not restricted. If we reduce the bandwidth, then, at some point, and will decrease the value of the old stream. But when values are too small, the flux will be insufficient to ensure that the restrictions (the minimum value of the flow along the edges). Obviously, you can apply **binary search** meaningfully INF, and it is a lowest value, wherein all constraints have will be met, but the old thread will have a minimum value.

Minimum cost flow (min-cost-flow).

Algorithm enhancing ways

Given network G, consisting of N vertices and M edges. Each edge (generally oriented, but on this occasion, see below) contains the bandwidth (a positive integer), and the cost per unit of flow along this edge (an integer). Column Set source S and drain T. gives some value K flux is required to find the flow of this magnitude, and among all flows of this magnitude choose the stream with the lowest cost ("the task of min-cost-flow").

Sometimes the task of putting a little differently: it is required to find the maximum flow (Least Cost "problem min-cost-max-flow").

Both these problems are solved effectively increasing the algorithm described below ways.

Description

The algorithm is very similar to [Edmonds-Karp algorithm for computing the maximum flow](#).

The simplest case

Consider, to begin the simplest case where the graph - oriented, and between any pair of vertices is at most one edge (if there is an edge (i, j) , then the edges (j, i) should not be).

Let U_{ij} - bandwidth edges (i, j) , if there is an edge. Let C_{ij} - cost per unit of flow along the edge (i, j) . Let F_{ij} - the flux along the edge (i, j) , initially all fluxes are zero.

Modify network as follows: for each edge (i, j) Add to your network so-called **reverse** edge (j, i) a bandwidth $U_{ji} = 0$ and cost $C_{ji} = -C_{ij}$. Since our assumption ribs (j, i) to this network was not modified in such a way that the network is still not a multigraph. In addition, throughout the algorithm will maintain the best condition: $F_{ji} = -F_{ij}$.

Define **residual network** for a fixed flow F as follows (actually, just as in the Ford-Fulkerson algorithm) residual network edge belong only unsaturated (i.e. which $F_{ij} < U_{ij}$), and the residual capacity of each such edge as $U_{Pij} = U_{ij} - F_{ij}$.

Properly **algorithm** min-cost-flow is as follows. At each iteration, the algorithm finds the shortest path in the residual network from S to T (the shortest relative cost C_{ij}). If the path is not found, then the algorithm returns, the thread F - desired. If the path has been found, we are increasing the flow along it as far as possible (ie, pass along this path, we find the minimum residual bandwidth MIN_UPI edges of this path, and then increase the flow along each edge of the path on the value MIN_UPI , do not forget to reduce the same amount of reverse flow along the edges). If at any time the flux reaches the value K (given to us by the condition of the flux), we also stop the algorithm (note that if the last iteration of the algorithm when the flow along the path need to increase the flow by an amount that the final flow surpassed K, but it is easily done).

Easy to see that if we put K to infinity, the algorithm finds the minimum cost maximum flow, ie the same algorithm solves both unchanged problem min-cost-flow and min-cost-max-flow.

Case of undirected graphs, multigraphs

Case of undirected graphs and multigraphs conceptually no different from above, so the actual algorithm will work on such graphs. However, there are some difficulties in the implementation, which should be noted.

Undirected edge (i, j) - it is actually two oriented edges (i, j) and (j, i) with same bandwidth and cost. Since the above described algorithm min-cost-flow needs for each undirected edges to create inverse edge, the result is that the non-oriented edge is split into four oriented edges, and we actually get the case **multigraph**.

What problems cause **multiple edges**? First, the flow on each of multiple edges should be maintained separately. Secondly, when searching for the shortest path, be aware that what is important is what kind of multiple edges to select the reduction path ancestor. Ie instead of the usual array of ancestors for each vertex we have to keep the top of the ancestor and the room edges on which we have come out of it. Thirdly, when the flow along the edges of a need according to an algorithm inverse to reduce flow along the rib. Since we may have multiple edges, you have to keep the number of each rib ribs reverse it.

Other difficulties with undirected graphs and multigraphs not.

Analysis of Time

By analogy with the analysis of Edmonds-Karp algorithm, we obtain the following estimate: $O(NM) * T(N, M)$, where $T(N, M)$ - the time required for finding the shortest path in a graph with N vertices and M edges. If this is implemented using [simplest version of Dijkstra's algorithm](#), Then the whole algorithm min-cost-flow will score $O(N^3 M)$.

(N3 M), however, Dijkstra's algorithm will be modified to make it work on graphs with negative weights (called Dijkstra's algorithm with potentials).

Instead, you can use [Leviticus algorithm](#) That, although asymptotically much worse, but in practice is very fast

(in about the same time as the Dijkstra algorithm).

Implementation

Here is an implementation of the algorithm min-cost-flow, based on [Leviticus algorithm](#).

Algorithm is applied to the input network (undirected multigraph) with N vertices and edges of M and K - the quantity of flow that you want to find. The algorithm finds the flow of K minimum cost, if one exists. Otherwise, it finds the maximum value of the flow minimum cost.

The program has a special feature to add an oriented edge. If need be add undirected edge, then this function must be called for each edge (i, j) twice: from (i, j) and from (j, i) .

```
const int INF = 1000 * 1000 * 1000;

struct rib {
    int b, u, c, f;
    size_t back;
};

void add_rib (vector <vector <rib>> & g, int a, int b, int u, int c)
{
    rib r1 = {b, u, c, 0, g [b]. size ()};
    rib r2 = {a, 0, -c, 0, g [a]. size ()};
    g [a]. push_back (r1);
    g [b]. push_back (r2);
}

int main ()
{
    int n, m, k;
    vector <vector <rib>> g (n);
    int s, t;
    Graph reading ... ...

    int flow = 0, cost = 0;
    while (flow <k) {
        vector <int> id (n, 0);
        vector <int> d (n, INF);
        vector <int> q (n);
        vector <int> p (n);
        vector <size_t> p_rib
        (n); int qh = 0, qt = 0;
        q [qt + +] =
        s; d [s] =
        0;
        while (qh! = qt) {
            int v = q [qh +
            +]; id [v] = 2;
            if (qh == n) qh = 0;
            for (size_t i = 0; i <g [v]. size ();
                + + i) {rib & r = g [v] [i];
                if (rf <ru && d [v] + rc <d [rb]) {d
                    [rb] = d [v] + rc;
                    if (id [r.b] == 0) {q
                        [qt + +] =
                        r.b;
                        if (qt == n) qt = 0;
                    }
                    else if (id [r.b] == 2) {
                        if (- qh == -1) qh = n-1;
                        q [qh] = r.b;
                    }
                }
            }
        }
    }
}
```

```
id [r.b] =  
1; p [r.b] =  
v;
```

```
        p_rib [r.b] = i;
    }
}

if (d [t] == INF) break;
int addflow = k - flow;
for (int v = t; v != s; v = p [v]) {
    int pv = p [v]; size_t pr = p_rib [v];
    addflow = min (addflow, g [pv] [pr]. u - g [pv] [pr]. f);
}
for (int v = t; v != s; v = p [v]) {
    int pv = p [v]; size_t pr = p_rib [v], r = g [pv]
[Pr]. Back;
    g [pv] [pr]. f += addflow; g [v] [r]. f -= addflow;
    cost += g [pv] [pr]. c * addflow;
}
flow += addflow;
}
```

Result output

}

Assignment problem. Solution with a min-cost-flow

The problem has two equivalent statement:

- Given a square matrix $A [1 \dots N, 1 \dots N]$. It is necessary to select N elements so that each row and column has been selected exactly one element, and the sum of these elements is a minimum.
- There orders N and N machines. About every order known its manufacturing cost for each machine. On each machine can perform only one order. Is required to distribute all orders for the machines so as to minimize the total cost.

Here we consider the solution of the problem based on the algorithm [finding the minimum cost flow \(Min-cost-flow\)](#), solving the problem of appointments for **O (N5)**.

Description

Construct bipartite network: has a source S , runoff T , in the first part are N vertices (Corresponding rows of the matrix or orders), the second - also N vertices (corresponding to the columns of a matrix or machines).

Between each vertex i of the first part and each vertex j of the second part will hold an edge with capacity 1 and the value of A_{ij} . From the source S will hold the edges to all vertices i of the first part with a capacity of 1 and value of 0. From each vertex of the second part to the drain T j draw an edge with capacity 1 and cost 0.

Find in the resulting network maximum flow minimum cost. Obviously, the flux will be equal to N . Furthermore, it is obvious that for each vertex i of the first part there is exactly one vertex j from the second part, such that the flow $F_{ij} = 1$. Finally, it is obvious that one-to-one correspondence between the vertices of the first part and vertices of the second part is the solution of the problem (as found flow has a minimum value, the sum of the costs of the selected edges will be the least possible, which is the optimality criterion).

Asymptotics of solutions of the assignment depends on how the algorithm searches for the maximum flow of minimum cost. Asymptotics will **O (N3)** using Dijkstra's algorithm or **O (N4)** using the algorithm of Bellman-Ford.

Implementation

Present here is the implementation longish possible, it can be reduced significantly.

```
typedef vector <int> vint;
typedef vector <vint>
vvint;
const int INF = 1000 * 1000 * 1000;

int main ()
{
    int n;
    vvint a (n, vint (n));
    Reading ... a ...

    int m = n * 2 + 2;
    vvint f (m, vint (m));
    int s = m-2, t = m-1;
    int cost = 0;
    for (; ;)
    {
        vector <int> dist (m,
        INF); vector <int> p (m);
        vector <int> type (m, 2);
        deque <int> q;
        dist [s] = 0; p
        [s] = -1; type
        [s] = 1;
        q.push_back (s);
        for (; ! q.empty ();)
        {
            int v = q.front (); q.pop_front
            (); type [v] = 0;
            for (int i = 0; i < n; i++)
            {
                if (a [i] [s] > 0)
                {
                    if (dist [i] >= INF)
                        dist [i] = INF;
                    else
                        dist [i] = min (dist [i], a [i] [s]);
                    p [i] = s;
                }
            }
            for (int i = 0; i < n; i++)
            {
                if (a [t] [i] > 0)
                {
                    if (dist [i] >= INF)
                        dist [i] = INF;
                    else
                        dist [i] = min (dist [i], a [t] [i]);
                    p [i] = t;
                }
            }
        }
        for (int i = 0; i < n; i++)
        {
            if (p [i] == -1)
            {
                if (type [i] == 1)
                    cost += dist [i];
                type [i] = 2;
            }
        }
        for (int i = 0; i < n; i++)
        {
            if (p [i] == -1)
            {
                if (type [i] == 2)
                    cost -= dist [i];
                type [i] = 1;
            }
        }
        for (int i = 0; i < n; i++)
        {
            if (p [i] == -1)
            {
                if (type [i] == 1)
                    cost += dist [i];
                type [i] = 2;
            }
        }
        for (int i = 0; i < n; i++)
        {
            if (p [i] == -1)
            {
                if (type [i] == 2)
                    cost -= dist [i];
                type [i] = 1;
            }
        }
    }
}
```

```

        if (v == s)
        {
            for (int i = 0; i <n; + +
                i) if (f [s] [i]
                == 0)
            {
                dist [i] = 0; p
                [i] = s; type
                [i] = 1;
                q.push_back (i);
            }
        }
    else
    {
        if (v <n)
        {
            for (int j = n; j <n + n; + + j)
                if (f [v] [j] <1 && dist [j]
> Dist [v] + a [v] [j-n])
                {
                    dist [j] = dist [v] + a
[v] [j-n];
                    p [j] = v;
                    if (type [j] ==
0) q.
push_front (j);
                    else if (type [j] == 2)
                        q.push_back (j);
                    type [j] = 1;
                }
        }
        else
        {
            for (int j = 0; j <n; + + j)
                if (f [v] [j] <0 && dist [j]
> Dist [v] - a [j] [v-n])
                {
                    dist [j] = dist [v] - a
[j] [v-n];
                    p [j] = v;
                    if (type [j] ==
0) q.
push_front (j);
                    else if (type [j] == 2)
                        q.push_back (j);
                    type [j] = 1;
                }
        }
    }
}

int curcost = INF;
for (int i = n; i <n + n; + + i)
    if (f [i] [t] == 0 && dist [i] <curcost)
    {
        curcost = dist
[i]; p [t] = i;
    }
if (curcost == INF) break;
cost += curcost;
for (int cur = t; cur! = -1; cur = p [cur])
{
    int prev = p
[cur]; if (prev!
-1)
        f [cur] [prev] = - (f [prev] [cur] = 1);
}
}

```

```
printf ("%d \n",
cost); for (int i = 0;
i <n; + + i)
    for (int j = 0; j <n; + + j)
```

}

```
if (f [i] [j + n] == 1)
    printf ("% d", j +1);
```

Hungarian algorithm for solving the assignment problem

Formulation of assignment problem

The assignment problem is put quite naturally.

Here are a few **setting options** (As is easily seen, they are all equivalent to each other):

- Yes n workers and n assignments. For each worker knows how much money he will ask for the fulfillment of a task. Each worker can afford to take only one job. Required to distribute work assignments so as to minimize the total cost.
- Given a matrix a size $n \times n$. Requires each row select one number, so that any column was also selected in exactly the same number and the sum of selected properties could be minimal.
- Given a matrix a size $n \times n$. Required to find a permutation p length n That the quantity $\sum a[i][p[i]]$
 - Minimal.
- Provides a complete bipartite graph with n vertices; each edge is assigned some weight.
Want to find a perfect matching of minimum weight.

Note that the above statement "**are square**": They both dimensions are always the same (and equal n). In practice, often similar "**rectangular**" Setting when select $\min(n, m)$ elements. However, as is easy to see from the "square" of the problem you can always go to "square", adding rows / columns with zero / infinite values, respectively.

Also note that, by analogy with the search **minimum** solutions also pose the problem of search **maximum** solutions. However, these two problems are equivalent to each other, all the weight is enough to multiply -1 .

Hungarian algorithm

Historical Background

The algorithm was developed and published by Harold **Kuhn** (Harold Kuhn) in 1955 gave himself Kun algorithm called "Hungarian" because it was largely based on the earlier works of two Hungarian mathematicians: Denes **Koenig** (Dénes König) and Eigen **Egervary** (Jenő Egerváry).

In 1957, James **Mankres** (James Munkres) showed that this algorithm works for the (strict) polynomial time (ie the time of the order of a polynomial n That does not depend on the magnitude of costs).

Therefore, in the literature, this algorithm is known not only as the "Hungarian", but as "algorithm Kuna Mankresa" or "algorithm Mankresa."

However, recently (2006) revealed that the same algorithm was invented **a century before the Kuna** German mathematician Carl Gustav **Jacobi** (Carl Gustav Jacobi). The fact that his work "About the research of the order of a system of arbitrary ordinary differential equations", printed posthumously in 1890, contained, among other results, and a polynomial algorithm for solving the assignment problem, was written in Latin, and its publication **gone unnoticed** among mathematicians.

Also worth noting that the original algorithm Kuhn had asymptotic and only later Jack **Edmonds** (Jack Edmonds) and Richard **Carp** (Richard Karp) (and, independently **Tomidzava** (Tomizawa)) shown how to improve its asymptotics $O(n^3)$.

Construction of an algorithm for $O(n^4)$

Just note in order to avoid ambiguities, we mainly consider here the task assignment in a matrix formulation (ie, given a matrix a And it is necessary to choose from n cells that are in different rows and columns). Indexing arrays, we start with unity, ie, for example, the matrix a has indices $a[1 \dots n][1 \dots n]$.

Also, we assume that all the numbers in the matrix a **nonnegative** (If not, you can always go to a non-negative matrix, adding all the numbers to a number).

We call **potential** two arbitrary array of numbers $u[1 \dots n]$ and $v[1 \dots n]$ such that the following condition:

$$u[i] + v[j] \leq a[i][j] \quad (i = 1 \dots n, \ j = 1 \dots n).$$

(As can be seen, the number $u[i]$ correspond to the lines, and the number

$v[j]$ - Columns of the matrix.) We call **value** f **building** the sum of its properties:

$$f = \sum_{i=1}^n u[i] + \sum_{i=1}^n v[i].$$

On the one hand, it is noted that the cost of the desired solution sol **not less** the value of any building:

(PROOF desired solution of the problem consists of a matrix of cells, and each of them condition $u[i] + v[j] \leq a[i][j]$. Since all the elements are in different rows and columns, then summing these inequalities for all selected $a[i][j]$ in the left-hand side we obtain, and the right - sol , QED.)

On the other hand, it appears that there is always a solution and the potential at which this inequality **becomes an equality**. The Hungarian algorithm is described below, will be constructive proof of this fact. In the meantime, only pay attention to the fact that if a solution has a value equal to the value of any potential, the decision - **optimally**.

Fix some potential. We call an edge (i, j) **tough**, if satisfied:

$$u[i] + v[j] = a[i][j].$$

Think of the alternative formulation of the assignment problem, using a bipartite graph. We denote H bipartite graph composed only of hard edges. In fact, the Hungarian algorithm support for the current building **the maximum number of edges on matching** M column H : And as soon as it becomes contain matching n Ribs, ribs and this matching will be the desired optimal solution.

Proceed directly to the **description of the algorithm**.

- At the beginning of the algorithm the potential is set to zero $u[i] = v[i] = 0$ And matching M relies empty.
- Further, at each step of the algorithm we try without changing the potential to increase the power of the current matching M per unit (recall is sought in the graph matching hard edges H).

This is actually used for ordinary [Kuhn search algorithm maximum matchings in bipartite graphs](#).

Recall here the algorithm.

All edges of the matchings M oriented in the direction from the first to the second part, all other edges H oriented in the opposite direction.

Recall (search matchings of terminology) that the vertex is called saturated if it is adjacent an edge of the current matching. Vertex, which is not adjacent to any edge of the current matching is called unsaturated. Path of odd length, which does not belong to the first edge matching, and for

all subsequent edges there is an alternation (belongs / does not belong) - called by magnifying.

Of all unsaturated vertices of the first part starts bypassing **depth/across**. If the bypass managed to reach the top of the second unsaturated share, it means that we have found a way of increasing the share of the first to the second. If procheredovat edges along this path (ie, the first rib include matching, eliminate second, the third turn, and so on), we thereby increase the power matching unit.

If increasing the path was not, then this means that the current matching M - Maximum in the graph H So in this case, proceed to the next step.

- If the current step is not able to increase output current matching, then made a conversion capacity so that the next steps there are more opportunities for larger matchings.

We denote the set of Z_1 vertices of the first part, which were visited traversal algorithm when trying to find Kuhn increasing the chain; through - many visited vertices of the second part. We denote Z_2 set of unsaturated vertices of the first part, through R_2 - The set of unsaturated vertices of the second part.

Calculate the value Δ :

$$\Delta = \min_{i \in Z_1, j \notin Z_2} \{a[i][j] - u[i] - v[j]\}.$$

This value is positive.

(PROOF Suppose that $\Delta = 0$. Then there is the hard edges (i, j) , Wherein $i \in Z_1$ and $j \notin Z_2$.

This implies that the edge (i, j) must have been directed from the first to the second part, i.e. this tough edge (i, j) should be part of a matching M into saturated top i . Except passing along the edge of j in i . A contradiction, so $\Delta > 0$.)

Now **recalculate potential** as follows: for all vertices
all vertices - Do . The resulting potential will remain
correct potential. $i \in Z_1$ $u[i] + = \Delta$
 $j \in Z_2$ $v[j] - = \Delta$

(PROOF For this we must show that it still holds and for all:

For cases where
the has not changed. $i \in Z_1 \& j \in Z_2$ or $i \notin Z_1 \& j \notin Z_2$ - So as for their programming $u[i] + v[j] \leq a[i][j]$
When $i \notin Z_1 \& j \in Z_2$ inequality only usilililos. Finally, for the case $i \in Z_1 \& j \notin Z_2$
- Although the left-hand side and increasing but still inequality persists, since the value of Δ According to its definition - it is just the maximum increase, which does not lead to a violation of the inequality.)

In addition, the old matching M of hard edges can be left, i.e. matching all edges remain tight.

(PROOF To some stiff edge (i, j) ceased to be tough as a result of changes potential, it is necessary that the equality $u[i] + v[j] = a[i][j]$ turned into inequality $u[i] + v[j] < a[i][j]$. However, the left side could be reduced only in one case: when $i \notin Z_1 \& j \in Z_2$
But times $i \notin Z_1$, It means that the edge (i, j) could not be an edge matching, as required.)

Finally, to show that the potential change **can not be infinitely**, We note that for each such potential varies the number of vertices reachable bypass, ie , strictly increasing. (This is not to say that an increasing number of hard edges.) (PROOF First, any node that was achievable, attainable, and will remain. Indeed,

if a vertex is reachable, then it has to be a path of vertices reachable starting top of the first in the unsaturated fraction; and as for the edges of the form $(i, j), i \in Z_1 \& j \in Z_2$ amount $u[i] + v[j]$ does not change, then continue all the way and after the change capacity, as required. Second, we show that the translation of potential there is at least one new vertex achievable. But

it is almost obvious if you go back to the definition of Δ : In fact, if the minimum has been reached on the edge (i, j) then this edge now becomes rigid, and therefore, the apex j become achievable thanks to the edges and vertices i .

Thus, the total can not be more n building conversions before being discovered increasing the power chain and matching M will be increased.

Thus, sooner or later be found potential, which corresponds to a perfect matching M Being the answer to the problem.

If we talk about **asymptotics** algorithm, it is because all must occur $O(n^3)$
 n increases matchings, before each of which occurs not more than n building conversions, each of which is carried out during $O(n^2)$.

For implementation $O(n^4)$ we shall not give here, because it still does not get shorter than described below for implementation $O(n^3)$.

Construction of an algorithm

for $O(n^3)$ $O(n^2m)$)

Now learn how to implement the same algorithm for the asymptotic behavior $O(n^3)$ (For rectangular tasks $n \times m$ - $O(n^2m)$).

The key idea: we will now **adding into consideration matrix row one after another**, a not treat them all at once. Thus, the algorithm described above will be:

- Add into consideration the next row of the matrix a .
- Yet increasing the chain starting in this line, recalculate potential.
- As soon as the chain increases, alternate matching along it (thus including the last line of matching), and go to the top (to the next line).

To achieve the required asymptotic behavior, it is necessary to implement steps 2-3 for each line running matrix during $O(n^2)$ (For rectangular tasks - for $O(nm)$).

To do this, we recall two facts proved above us:

- When the potential peaks that were achievable bypass Kuhn, achievable and will remain.
- Total could happen only $O(n)$ building conversions before increasing the chain is found.

This implies **key ideas**That achieve the desired asymptotic behavior:

- To check which increases chain no need to run again after bypassing Kuhn each conversion potential. Instead, you can get a bypass Kuhn **iterative** form after each conversion potential we look to add a hard edge, and if they were left ends

achievable, mark their right ends as achievable and keep bypassing of them.

- Developing this idea further, we can come to such a representation of the algorithm: a cycle, at each step, which is converted first potential, then there is a column that has become achievable (and there will always be such as restated building there are always new reachable vertices), and if this unsaturated column was then found increasing the chain, and if the column was saturated - then the corresponding row in the matching problem also becomes achievable.

$$j \quad O(n^2)$$

As is easily seen, the unknown quantity Δ therethrough expressed as follows:

$$\Delta = \min_{j \notin Z_2} \{minv[j]\}.$$

Thus, finding Δ Now you can make for $O(n)$.

Maintain the array $minv[]$ necessary when new rows visited. This can obviously be made for $O(n)$ one string to be added (for a total give $O(n^2)$). Also update the array $minv[]$ must in terms of capacity, which is also done during $O(n)$. One conversion potential (as $minv[]$ changes only for the unreach while columns: namely, reduced by Δ).

Thus, the algorithm takes the form in the outer loop, we add in the consideration of one row of the matrix after another. Each row is processed in time, as this could only happen

$O(n)$ building conversions (each - for the time $O(n)$ which for the time $O(n^2)$) supported array $minv[]$: algorithm is run for a total of HRK during $O(n^2)$ (Since it is in the form $O(n)$ iterations each of which is visited by a new column).

The resulting asymptotic behavior is $O(n^3)$ - Or, if the problem is rectangular, $O(n^2m)$.

Implementation of the Hungarian algorithm for $O(n^3)$ ($O(n^2m)$)

The implementation shown was actually developed **Andrey Lopatin** a few years ago. It features amazing conciseness: the whole algorithm is placed in **30 code lines**.

This implementation is looking for a solution for a rectangular input matrix $a[1 \dots n][1 \dots m]$ where $n \leq m$.

The matrix is stored in 1-Indexing for convenience and brevity code. The fact that in this implementation are introduced fictitious zero row and zero column that allows you to write many cycles in general terms, without further checks.

Arrays and store potential $a[0 \dots n][0 \dots m]$ initially it is zero, which is true for the matrix consisting of zero lines. (Note that this implementation is not important whether or not there in the matrix $a[]$ negative numbers.)

Array $p[0 \dots m]$ contains matching: for each column $i = 1 \dots m$ it stores the corresponding to the selected row $p[i]$ (Or, until nothing is selected). In this $p[0]$ for convenience implementation is assumed to be the current number of qualifying rows.

Array $minv[1 \dots m]$ contains for each column supporting the minimum necessary to Fast conversion capacity:

Array $way[1 \dots m]$ contains information about where these minima are obtained, we then were able to restore magnifying chain. At first glance it seems that the array $way[]$ for each column, line number to be stored, as well as make another array: for each row to remember the number of the column from which we came into it. Instead, it can be seen that the algorithm Kuhn always gets in line, passing through the edge of matching columns, so the line numbers to restore the chain can always be taken out of matchings (ie, from an array).

Thus for each column, j contains $p[]$ number preceding column (or 0 if none).

The algorithm is a external **loop through the rows of the matrix** Inside which takes into consideration the addition of the second row of the matrix. The inner part is a series of "do-while ($p[j0]! = 0$ ", which works, until it finds a free column visited the new column with the number of 0 is, we start we dummy column), as well as a new line and initially at $j0 = 0$ taken $i0$ th row). Because of the advent of a new row visited $i0$ necessary

properly counted array

An $\minv[]$, At the same time we find in it at least - value δ

which column j_1 this minimum has been reached (note that in this implementation δ could be zero, which means that the potential at the current step may not be changed: new attainable column has already).

After this conversion capacity, the corresponding change in the array.

$\minv[]$

When the cycle is "do-while" we found a magnifying chain terminating in a column, "row"

which can, using an array of ancestors $way[]$.

Constant INF - Is "infinity", i.e. some number greater than all known possible number of

input matrix $a[][]$.

```
vector <int> U (n +1), V (m +1), P (m +1), Way (m +1);
for (int i =1; i <= n; + + I) {
    p[0] = I;
    int j0 = 0;
    vector <int> Minv (m +1, INF);
    vector <char> Used (m +1, false);
    do {
        used[j0] = true;
        int i0 = p[j0], delta = INF, j1;
        for (int j =1; j <= m; + + J)
            if (! Used[j]) {
                int cur = a[i0][j]-U[i0]-V[j];
                if (cur < minv[j])
                    minv[j] = Cur, way[j] = J0;
                if (minv[j] < Delta)
                    delta = minv[j], j1 = j;
            }
        for (int j =0; j <= m; + + J)
            if (used[j])
                u[p[j]] += Delta, v[j] -= Delta;
            else
                minv[j] -= Delta;
        j0 = j1;
    }while (p[j0] != 0);
    do {
        int j1 = way[j0];
        p[j0] = P[j1]; j0
        = j1;
    }while (j0);
}
```

Answer to reset a usual manner, i.e. finding for each row

it Column

$ans[i]$, Is as follows:

number of selected

$i = 1 \dots n$

```
vector <int> Ans (n
+1);
for (int j =1; j <= m; + + J)
    ans[p[j]] = J;
```

Cost found matching you can just take the potential of zero column (taken from

opposite sign). In fact, it is easy to trace through the code $-v[0]$ contains the sum of all units δ le the total change in potential. While at each change of potential change could several variables $u[i]$ and j . The total change in value of the potential is exactly equal δ , since there are no magnifying circuit, the number of rows achievable exactly one more than the number of achievable column (only the current line i has no "couples" as visited column):

```
int cost =-v[0];
```

Examples of problems

Here are a few examples in the task assignment: from the very trivial, and ending with less obvious problems:

- Given bipartite graph, find it matching **maximum minimum weight matching** (i.e., primarily maximized size matchings the second - the cost is minimized).

To solve the problem of building a simple assignment, putting in place the number of missing ribs "infinity". After that we solve the problem by the Hungarian algorithm, and remove the ribs from the response of infinite weight (they could enter

in response, if the problem has no solution in the form of a perfect matching).

- Given bipartite graph, find it matching **maximum maximum weight matching**.

Decision again obvious, but all the weight must be multiplied by minus one (or in the Hungarian algorithm to replace all the minima on the highs, and infinity - at minus infinity).

- Task detecting moving objects from images:** Was made two shots, following which there were two set of coordinates. Required to relate the objects in the first and second picture, ie, to determine for each point of the second shot, the first shot at what it was adequate. It is required to minimize the sum of the distances between the mapped points (ie, we seek a solution in which objects are passed smallest total path).

To solve we just build and solve the problem of assignment, where the weights of the edges are the Euclidean distance between points.

- Task detecting moving objects on the radar:** There are two locator who can not determine the position of an object in space, but only the direction to it. With both locators (at different points) came in the form of information n such directions. Required to determine the position of objects, ie determine the expected position of objects and their corresponding pairs of directions so as to minimize the sum of distances from objects to ray-directions.

Decision - again, just build and solve the problem of assignment, where the vertices of the first part are n directions from the first locator vertices of the second part - n directions from the second locator and edge weights

- The distance between the beams.

- Cover directed acyclic graph paths:** A directed acyclic graph, find the smallest number of paths (with equal - with the smallest total weight) to each vertex of the graph would lie in exactly one way.

Solution - build on the graph corresponding bipartite graph, and find the maximal matching of minimum weight. For more details see [separate article](#).

- Coloring wood.** Given a tree, wherein each vertex, but leaves, has exactly $k-1$ sons. You want to select for each vertex of some color colors so that no two adjacent vertices have the same color. Furthermore, for each vertex of each color and known dyeing this peak value in the color and required to minimize the total cost.

Solutions for use by dynamic programming. Namely, learn to consider the value

Whe
rein - Number of vertices - number of colors, and the value $d[v][c]$ - This is the minimum cost
 $d[v][c]$ coloring vertices together with its descendants, the vertex itself v has color. To calculate this
, it is necessary to allocate the c
value $d[v][c]$ remaining $k-1$ flowers on top of the sons, and for this it is necessary to build
and solve the problem of assignment (in which the vertices of one share - the colors, the other vertices share - tops-sons,
and weights of the edges - This value corresponds to the
speaker $d[][]$).

Thus, each value $d[v][c]$ considered by solving the assignment problem, with the result that
gives an asymptotic $O(nk^4)$.

- If the assignment problem is not given weight of the ribs, and at the tops, and only **at the tops of one share**, It is possible to dispense with the Hungarian algorithm, but rather a sort weight vertices and start normal [Kuhn algorithm](#) (See more details [separate article](#)).

- Consider the following **special case**. Let each vertex of the first part is attributed to a number of $\alpha[i]$, Suppose that the weight and each vertex of the second part - $\beta[i]$ of any edge equal (Number and we know). Solve the problem of appointments.

For solutions without the Hungarian algorithm, we first consider the case in which both shares two vertices. In this case, as is easily seen, it is advantageous to connect the vertices in reverse order: the top with less $\alpha[i]$ join

. This rule can be easily generalized to an arbitrary number of vertices: it is a vertex with larger $\alpha[i]$ necessary to join. sort the vertices of the first part in order of increasing $\alpha[i]$, The second part - in order of decreasing $\beta[i]$, and connecting the vertices in pairs in that order. Thus, we obtain a solution with the asymptotic $O(n \log n)$.

- The problem of the potentials.** Given a matrix $a[1 \dots n][1 \dots m]$. Required to find the two arrays $u[1 \dots n]$ and $v[1 \dots m]$ such that, for any i, j and running $a[i][j] \leq u[i] + v[j]$. But the sum of $u[1 \dots n]$ array elements $u[]$ and $v[]$ maximum. $u[i] + v[j] \leq a[i][j]$

Knowing the Hungarian algorithm, this task will not be difficult: the Hungarian algorithm just finds $u[], v[]$ Which satisfies the condition of the problem. On the other hand, without knowledge it is so much potential $u[], v[]$ Hungarian algorithm to solve this problem seems almost impossible.

Literature

- Ravindra Ahuja, Thomas Magnanti, James Orlin. **Network Flows** [1993]
- Harold Kuhn. **The Hungarian Method for the Assignment Problem** [1955]
- James Munkres. **Algorithms for Assignment and Transportation Problems** [1957]

Problem in online judges

A list of tasks in the task assignment:

- UVA # 10746 "**Crime Wave - The Sequel**" [Difficulty: Easy]
- UVA # 10888 "**Warehouse**" [Difficulty: Medium]
- SGU # 210 "**Beloved Sons**" [Difficulty: Medium]
- UVA # 3276 "**The Great Wall Game**" [Difficulty: high]
- UVA # 10296 "**Jogging Trails**" [Difficulty: high]

Finding minimal cut. Stohr-Wagner algorithm

Statement of the Problem

Given an undirected weighted graph with n vertices and m edges. Cut is called a subset of vertices (in fact, the cut - partition the vertices into two sets belonging to everyone else). Weight cut is the sum of the weights of the edges passing through the slit, i.e. such edges, exactly one end of which is owned C :

$$w(C) = \sum_{\substack{(v, u) \in E, \\ u \in C, v \notin C}} c(v, u),$$

where by E denotes the set of all edges G and through $c(v, u)$ - weight of the edge (v, u) .

Required to find **cut the minimum weight**.

Sometimes this problem is called a "global minimum cut" - in contrast to the task when given the vertex and the source-drain and requires a minimal incision containing runoff and does not contain the source.

Global minimal incision equal to the minimum of the minimum cost cuts over all pairs source-drain.

Although this problem can be solved by using an algorithm for finding the maximum flow (running it $O(n^2)$ time for all pairs of source and drain), but the following describes a much simpler and faster algorithm proposed Matilda

Stohr (Mechthild Stoer) and Frank Wagner (Frank Wagner) in 1994

Generally allowed loops and multiple edges, although, of course, hinges absolutely no influence on the result, and all multiple edges can be replaced by a single edge with their total weight. Therefore, for simplicity, we shall assume that the input graph loops and multiple edges are missing.

Description of the algorithm

The basic idea algorithm is very simple. Will iteratively repeating the following process: find minimum cut between any pair of vertices and then combine these two peaks one iteration, the graph is compressed into a single vertex (Connecting the adjacency lists). Eventually, after s and found iteration, the graph is compressed into a single vertex s .
process will stop. After that, the answer would be minimal among all cuts. Indeed, on every second stage found minimal incision i either not desired global minimum cut, or, on the contrary, the tops found between the peaks and C_i and unprofitable s_i classified

either not desired global minimum cut, or, on the contrary, the tops different sets, so we did not deteriorate, combining these two vertices into one.
Thus, we have another problem: given a graph to find **minimal incision between some, arbitrary pair of vertices s and t** . To solve this problem, the following has been proposed also an iterative process. Introduce a set of vertices A which initially contains a single arbitrary vertex. At each step, there is a vertex most strongly associated a plurality, i.e. top $v \notin A$ for which the next maximum value:

(i.e., the maximum sum of the weights of edges, one end of which v and the other belongs A).

Again, this process is completed through $n - 1$ iteration, when all the vertices will move in many A (Incidentally, this process is very similar [Prim's algorithm](#)). Then, as the **Stohr-Wagner theorem**, if we denote by s and t recently the two written in A peaks, the minimum cut between vertices s and t will consist of a single vertex - t . The proof of this theorem will be given in the next section (as is often the case, by itself it does not contribute to the understanding of the algorithm).

Thus, total chart Curtains-Wagner is. The algorithm consists of $n - 1$ phase. Each phase set A relies first consisting of a vertex; counted starting weight vertices $w(v, A)$. Then there $n - 1$ iteration, each of which is selected vertex u with highest value $w(v, A)$ and added to the set A . Then converted values w

for the remaining vertices (which, obviously, need to go through all the edges of the selected vertex adjacency list u). After all the iterations we remember in s and the numbers of the last two vertices added, and As the cost of minimal cut found between and can take the value t $w(t, A \setminus t)$. Then found to be compared with the current minimum cut answer if less, then update the answer. Go to the next phase.

If you do not use any complex data structures, the most critical part is to find the vertex with the highest value. If you make it for $O(n)$ Then, given that all phases $n - 1$ And by $- 1$ iteration each, total asymptotic behavior of the algorithm obtained $O(n^3)$.

If finding the top with the highest value w use Fibonacci heap
(Which allow to increase the value of the key for $O(1)$ in the middle and make the most for $O(\log n)$ on average) all operations related to a plurality of executed on one phase for $O(m + n \log n)$. Summary asymptotic behavior of the algorithm in this case is $O(n + n^2 \log n)$.

Proof Stohr-Wagner

Recall the condition of this theorem. If we add to the set A turns all the vertices, each time adding a vertex, most strongly associated with this set, then we added the penultimate summit via s And last - through t . Then the minimal $s-t$ incision consists of a single vertex - t .

To prove this, consider an arbitrary $s-t$ incision and show that its weight can not be less than the weight section, consisting of a single vertex t :

For this we prove the following fact. Let A_v - Set the state immediately before the addition of top. Let $v \in C_v$ - Cut set $A_v \cup v$ Induced by a cut (in other words, equal to the intersection of these two sets of vertices). Next, the top v called an active (with respect to the slit) when the vertex v and added to the previous vertex belong to different parts of the section. Then, it is claimed for any active vertices v the inequality:

$$w(v, A_v) \leq w(C_v).$$

In particular, t is an active vertex (since before it adds vertices s), And $v = t$ this inequality becomes an assertion:

$$w(t, A_t) = w(\{t\}) \leq w(C_t) = w(C).$$

So, we will prove the inequality, for which we use the method of mathematical induction.

For the first active vertex v this inequality is true (in fact, it becomes an equality) - because all the vertices A_v belong to the same part of the section, and v - Other.

Suppose now that this inequality holds for all active vertices until some vertex v . Prove it for the next active vertex u . To do this, transform the left side:

$$w(u, A_u) \equiv w(u, A_v) + w(u, A_u \setminus A_v).$$

First, we note that:

- This follows from the fact that when a plurality of power was A_v , It was just added top, and not, so she had the highest value. w

Further, since $w(v, A_v) \leq w(C_v)$ By the induction hypothesis, we get:

$$w(u, A_v) \leq w(C_v),$$

Hence we have:

$w(u, A_u) \leq w(u, A_v) + w(u, A_u \setminus A_v)$
Now, note that the top u and all vertices u are different parts of the section, so this value $w(u, A_u \setminus A_v)$ $w(C_u)$ included in $w(C_u)$ But has not yet been taken into account in $w(C_v)$, Which yields:

QED.

We have proved the relation $w(v, A_v) < w(C_v)$ And from there, as mentioned above, it should be and all theorem.

Implementation

For the most simple and clear implementation (with the asymptotic adjacency matrix. The answer is stored in the variable $O(n^2)$ and cut themselves vertices contained in it).) Was chosen as a graph representation (The desired value of the minimum cut best_cost best_cut For each vertex in the array other vertex. The list for each vertex compressed stored, whether it exists or it was associated with some the vertex. v[i] i The algorithm consists of phase (in the variable cycle outside the set, for which the array in_a connectedness of $n - ph$ iteration is the summit sel with the highest value if necessary, updated, and the penultimate prev and the last sel If not the last iteration, the sel A). At each phase of the first all vertices are w all vertices are zero. Each w . If this is the last iteration, the answer selected vertices are merged into one.

g

```
const int MAXN = 500;
int n, g[MAXN];
vector<int> Best_cut;
int best_cost = 1000000000;
vector<int> v[MAXN];
bool exist[MAXN], In_a[MAXN];
int w[MAXN];
int ph = 0;
for (int i = 0; i < n; ++i)
    v[i].Assign(1, i);
int sel = -1;
for (int ph = 0; ph < n-1; ++ph) {
    memset(in_a, false, sizeof(in_a));
    memset(w, 0, sizeof(w));
    for (int it = 0, Prev; it < n-ph; ++it) {
        for (int i = 0; i < n; ++i)
            if (exist[i] && !In_a[i] && (sel == -1 || w[i] > w[sel]))
                sel = i;
        if (it == n-ph-1) {
            if (w[sel] < best_cost)
                best_cost = w[sel], Best_cut = v[sel];
            v[prev].Insert(v[prev].End(), v[sel]);
            Begin();
            for (int i = 0; i < n; ++i)
                g[prev][i] = G[i][prev] += G[sel][i];
            exist[sel] = false;
        } else {
            in_a[sel] = true;
            for (int i = 0; i < n; ++i)
                w[i] += G[sel][i];
            prev = sel;
        }
    }
}
```

Literature

- Mechthild Stoer, Frank Wagner. **A Simple Min-Cut Algorithm** [1997]
- Kurt Mehlhorn, Christian Uhrig. **The minimum cut algorithm of Stoer and Wagner** [1995]



Minimum cost flow, circulation of minimum cost.

Algorithm for removing cycles of negative weight

Setting objectives

Let - net (network), that is a directed graph in which the selected vertex-source s Denote the set of vertices, the set of edges - through V . Each edge E and runoff. its capacity $u_{ij} \geq 0$ and the cost per unit flow c_{ij} . If some ribs $(i, j) \in E$ compared s and t , the graph does it is assumed that $u_{ij} = c_{ij} = 0$. (i, j) not,

Flow (Flow) network G called such a real-valued function which assigns to each pair vertices (i, j) flow f_{ij} between them, and satisfies the following three conditions:

- Bandwidth Limit (holds for any $i, j \in V$)

$$f_{ij} \leq u_{ij}$$

- Antisymmetry (holds for any $i, j \in V$)

$$f_{ij} = -f_{ji}$$

- Flux conservation (holds for any $i \in V$ Except $i = s, i = t$)

$$\sum_{j \in V} f_{ij} = 0$$

The flux is the quantity

$$|f| = \sum_{i \in V} f_{si}$$

Value stream is the quantity

$$z(f) = \sum_{i, j \in V} c_{ij} f_{ij}$$

The problem of finding **minimal cost flow** is that for a given volume flow is required to find a stream with a minimum cost. Should pay attention to the fact

that the value assigned to the edges are responsible for the cost per unit of flow along this edge; problem sometimes occurs when the edges are mapped value stream flow along this edge (ie, if the flow proceeds of any size, then this cost will be charged, regardless of the value stream) - this problem has nothing to do with being considered here and, moreover, is NP-complete.

The problem of finding **maximum flow of minimum cost** is to find the flow of the highest value, and all of these - with minimal cost. In the special case when the weights of all edges are the same, the problem becomes equivalent to the usual problem of maximum flow.

The problem of finding **minimum cost circulation** is to find a value of zero flow to a minimum value. If all non-negative value, then, of course, the answer is zero flow; if there is a negative edge weights (or rather, the weight of the negative cycle), even

$f_{ij} = 0$ at zero flow stream may find a negative value. The problem of finding a minimum cost circulation can, of course, and put on the network without the source and drain, as no semantic load they carry (however, in this graph, you can add the source and drain in the form of isolated vertices and

get a regular on the wording of the task). Sometimes the task of finding the maximum value of the circulation

- Clear enough to change the value of the ribs on the opposite and get the problem of finding a minimum cost circulation already.

All these problems, of course, can be extended to undirected graphs. However, the move from undirected graph to easily oriented: each undirected edge

with

a
n
d

Bandwidth

and cost

should be replaced by two directed edges

u_{ij}

c_{ij}

(i,j)
 $(i,j) \ (j,i)$

with the same bandwidth and cost.

Residual Network

Concept **residual network** based on the following simple idea. Suppose there is some flow f ; along each edge $(i, j) \in E$ takes some flow G^f . Then along this edge can be (Theoretically) to put more $u_{ij} - f_{ij}$ flow units; this value and call **residual Bandwidth**:

The cost of these additional units of flow will be the same:

In the residual However, apart from this, **direct** ribs (i, j) network G^f appears and **back edge** (j, i) . Intuitive sense of the edge that we can in the future to cancel part of the stream that flowed through the edge (i, j) . Accordingly, the flow passing along the edges of the reverse (j, i) actually, formally, means a decrease in the flow along the edge (i, j) . Back edge has a bandwidth equal to zero (To, for example, $f_{ij} = 0$ and back edges would have been impossible to miss the stream; at positive value $f_{ij} > 0$ reverse edges to become the property of the antisymmetry $f_{ji} < 0$). That less $c_{ji}^f = 0$ (ie you can skip some return flow along the edges), the residual capacity - equal to flow along a straight edge, and cost - the opposite (because after the cancellation of the flow, we must be reduced accordingly and the price):

$$\begin{aligned} u_{ji}^f &= 0 \\ r_{ji}^f &= f_{ij} \\ c_{ji}^f &= -c_{ij} \end{aligned}$$

Thus, each oriented edge corresponds to two oriented edges in the residual network G^f . And each edge of the residual network, an additional characteristic - residual bandwidth. However, you will notice that the expressions for residual capacity essentially the same for both the forward and reverse edges, ie we can write for any edge (i, j) by residual network:

By the way, the implementation of this property prevents storing residual capacity, and simply calculate them when needed for an edge.

It should be noted that the residual network removes all edges with zero residual capacity ability. Residual Network G^f must be **only edges with positive residual bandwidth** r_{ij}^f .

Here you should pay attention to this important point: if the network G^f simultaneously both edges (i, j) and (j, i) , then the residual network, each of them will appear on the reverse side, and the result will **multiple edges**. For example, this situation often occurs when the network is built on an undirected graph (and, it turns out each undirected edge will eventually lead to the emergence of four edges in the residual network). This feature must always be remembered, it leads to a slight complication of programming, although in general does not change anything. Furthermore, the designation rib (i, j) in this case becomes ambiguous, so here we shall everywhere assume that the situation in the network is not (solely for the purpose of simplicity and correctness of descriptions, and on the correctness of the ideas is not affected).

Optimality criterion for the presence of cycles of negative weight

Theorem. Some flow is optimal (i.e., has the lowest cost of all such flows same value) if and only if the remaining network G^f contains no cycles of negative weight.

Proof: the need. Suppose that the flow is optimal. Assume that the residual network contains a cycle of negative weight. Take this cycle of negative weight and choose the least residual capacity of the edges of this cycle (to be greater than zero). But then we can increase the flow along each edge of the cycle on the magnitude, while no flow properties are not violated, the flux does not change, but the value of the flow is reduced (reduced by the cost of the cycle, multiplied by). Thus, if there is a cycle of negative weight, f may not be optimal, QED

Proof: The sufficiency. To do this, we first prove some auxiliary facts.

Lemma 1 (Decomposition of the stream) any flow can be represented as a set of paths from the source to the drain and cycles, all - have a positive flow. Prove this lemma is constructive: we show how to break into a plurality of flow paths and cycles. If the thread has a nonzero value, then obviously, the source of s out at least one edge with positive flow; pass along this edge, find ourselves in some vertex v_1 . If this vertex, then stop - have found a way of s in. Otherwise, according to the property to maintain the flow of v_1 must extend at least one edge with positive flow; Go through it at some vertex v_2 . Repeating this process, we will either come in stock t , Or else come to some vertex a second time. In the first case we find a way of s to, in the second - cycle. Found path / cycle will have a positive flow (minimum flow of the edges of this path / cycle). Then reduce the flow along each edge of the path / cycle by an amount due to get re-flow, which again apply this process. Sooner or later, all the edges along the stream becomes zero, and we find him on the way and decomposition cycles.

Lemma 2 (On the difference between the fluxes) For any two values of flow and

a () Flow can be
 f g . Indeed, consider the difference between

represented as a stream, plus a few cycles in the residual network G^f these flows $g - f$ (Subtraction flows - this term by term subtraction, ie subtraction flows along each edge). It is easily seen that the result will be some flow zero value, ie circulation. We make the decomposition of this circulation by the previous lemma. Obviously, it can not be decomposed paths (because the presence s, t). Path positive flow means that the quantity of flow in the network is positive). Thus, the difference in flow g and f can be represented as the sum of network cycles G .

Moreover, it will cycle in the residual network G^f . Since $g_{ij} - f_{ij} \leq u_{ij} - f_{ij} = r_{ij}^f$, QED

Now, armed with these lemmas, we can easily **prove the sufficiency**. So consider an arbitrary flow in the residual network that does not have a negative cost cycles. Consider

also stream the same size, but the minimum cost; prove that f^* equally worth. According to Lemma 2, the flow f^* can be represented as the sum of the flow f^* and several cycles. But times value of all non-negative cycles, then the value of the stream f^* can not be less than the value stream f . On the other hand, because $f^* \geq z(f^*)$ the flow f^* is optimal, then its value can not be higher value of the flow f . Thus, $z(f) = z(f^*)$, QED

Algorithm for removing cycles of negative weight

Just proved theorem gives us a simple **algorithm** for finding the minimum cost flow: if we have some thread, then build him a residual network, check whether there is a negative cycle in it weight. If this cycle is not, then the flow is optimal (has the lowest cost of all such flows same magnitude). If it was found negative cost cycle, then calculate flow, which you can skip further through this cycle (this is equal to the minimum of the residual capacities of the edges of the cycle). Increasing the flow on each edge along the cycle, we obviously do not disturb the flow properties do not change the value of the flux, but will reduce the cost of this flow by getting a new stream for which we must repeat the process.

So, to start the process of improving the value of the flow, we need to find a pre- **arbitrary stream of the desired value** (Some standard algorithm determination of the maximum flow, see, e.g., [Edmonds-Karp algorithm](#)). In particular, if you want to find the lowest cost circulation, then you can just start with zero flow.

We estimate the **asymptotics** algorithm. Search a negative cost cycle in a graph with n and peaks m fins made for (see [relevant article](#)). If we denote the greatest values of edges through - most of bandwidth, the maximum value does not exceed the value stream. If all bandwidths and cost - are integers, each iteration of the algorithm reduces the value of the stream by at least one; therefore, all make the algorithm $O(mCU)$ iterations, and the final asymptotic behavior will be:

$$O(nm^2CU)$$

This asymptotic behavior - not strictly polynomial (strong polynomial), because it depends on amount of bandwidth and cost.

However, if the search is not an arbitrary negative cycle, and use some kind of a clear approach, the asymptotic behavior can be significantly reduced. For example, if every time look for a cycle with minimum average cost (which can also be adjusted for), while the work of the entire algorithm is reduced to $O(nm^2 \log n)$ And this asymptotic behavior is strictly polynomial.

Implementation

We first introduce the data structures and functions to store the graph. Each edge is stored in a separate structure

edge All edges lie in the general list **edges**. And for each vertex i numbers are stored in the vector **edges** emanating from it. This organization makes it easy to find a number, reverse rib line straight edges - they are in the list of neighbors, and the number one can be obtained by calling another operation " $\wedge 1$ " (it inverts the LSB). Adding oriented edge in the graph performs the function **add_edge**. Which immediately adds forward and reverse edges.

```
const int MAXN = 100*2;
int n;
struct edge {
    int v, to, u, f, c;
};
vector<edge> edges;
vector<int>
G[MAXN];
void add_edge (int v, int to, int cap, int cost) {
    edge e1 = { v, to, cap, 0, Cost };
    edge e2 = { to, v, 0, 0, -Cost }; g[v].
    Push_back ((int) edges.size());
    edges.push_back (e1); g[to].
    Push_back ((int) edges.size());
    edges.push_back (e2);
}
```

In the main program after reading the graph is an infinite loop, inside which runs the algorithm of Bellman-Ford, and if it detects a negative cost cycle, then this cycle along the flow increases. As the residual network may be a disconnected graph, the algorithm runs the Bellman-Ford from each not yet reached the top. In order to optimize the algorithm uses the queue (the current queue **q** and the new queue **nq**). So as not to touch on all the edges of each step. Detected along the cycle each time the unit is pushed exactly flow, although, of course, in order to optimize the flow quantity can be defined as the minimum residual capacity.

```
const int INF = 1000000000;
for (, i) {
    bool found = false;

    vector<int> D (n, INF);
    vector<int> Par (n, -1);
    for (int i = 0; i < n; ++ I)
        if (d[i] == INF) {
            d[i] = 0;
            vector<int> Q,
            nq; q.push_back
            (i);
            for (int it = 0; it < n && q.size(); ++ it)
                if (nq.clear());
                sort (q.begin(), Q.end());
                q.erase (unique (q.begin(), Q.end()),
                Q.end()); for (size_t j = 0; j < q.size(); ++ J)
                {
                    int v = q[j];
                    for (size_t k = 0; k < g[v].Size(); ++ K)
                        if (edges[id].F < edges[id].U)
                            if (d[v] + Edges[id].
                                d[edges[id].to]) {
                                to = D[v] + Edges[id].c;
                                par[edges[id].to] = v;
                                push_back (edges[id].to);
                            }
                }
                swap (q, nq);
            }
            if (q.size()) {
```

```
int leaf = q[0];
vector <int>
Path;
for (int v = leaf; v! = -1; v = par[v])
    if (find (path.begin (), Path.end (),
```

```

v) == Path.end())
    path.push_back (v);
else {
    path.erase (path.begin(),
                break;
}
for (size_t j =0; j <path.size(); ++ J) {
    int to = path[j], v = path[(j +1)% path.size()];
    for (size_t k =0; k <g[v].Size(); ++ K)
        if (edges[ g[v] [k]]. To == to)
            { int id = g[v] [k];
            edges[id]. F += 1;
            edges[id ^1]. F -= 1;
}
found = true;
}
if (! Found) break;
}

```

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Construction and analysis** [2005]
- Ravindra Ahuja, Thomas Magnanti, James Orlin. **Network flows** [1993]
- Andrew Goldberg, Robert Tarjan. **Finding Minimum-Cost Circulations by Cancelling Negative Cycles** [1989]

Algorithm Diniz

Statement of the Problem

Suppose we are given a network, ie directed graph G in which each edge (u, v) assigned bandwidth c_{uv} And two peaks are highlighted - Source s and stock t .

To find in this network flow f_{uv} from the source s in stock t maximum.

A little history

This algorithm was published by the Soviet (Israeli) scientists Yefim **Diniz** (Yefim Dinic, sometimes written as "Dinitz") in 1970, ie even two years before the publication of the Edmonds-Karp algorithm (however, both algorithms were independently discovered in 1968).

Furthermore, it should be noted that some of the algorithm were made to simplify Shimon Ewen (Shimon **Even**) and his student Alon Itai (Alon **Itai**) in 1979 of Because of them got his algorithm modern look: they used to the idea of the concept of blocking flows Diniz Karzanova Alexander (Alexander Karzanov, 1974) and reformulated the algorithm to the combination of bypass in width and in depth, which is now the algorithm and presents everywhere.

Development of ideas in relation to streaming algorithms extremely interesting to consider, taking into account "**Iron Curtain**" of those years, divided the USSR and the West. Evident sometimes similar ideas were almost simultaneously (as in the case of the algorithm Diniz and Edmonds-Karp algorithm), however, while having different efficiencies (algorithm Diniz one order of magnitude faster); sometimes, on the contrary, the emergence of ideas on one side "curtain" ahead of a similar move on the other side more than a decade (the algorithm Karzanova push in 1974, and the algorithm of Goldberg (Goldberg) push in 1985).

Necessary definitions

We introduce three definitions needed (each of them is independent from the others), which are then used in the algorithm Diniz.

Residual network with respect to the network G . Some streams and it is called a network in which each edge $(u, v) \in G$ a bandwidth c_{uv} and stream f_{uv} correspond to two edges

- (u, v) a bandwidth $c_{uv}^R = c_{uv} - f_{uv}$
- (v, u) a bandwidth $c_{vu}^R = f_{uv}$

It is noteworthy that with this definition in the remaining network may appear multiple edges: if the original network were as rib (u, v) And (v, u) .

Residual edge can be intuitively understood as a measure of how still possible to increase the flow along some ribs. In fact, if an edge (u, v) a bandwidth c_{uv} stream flows f_{uv} Then potentially You can skip over it yet $c_{uv} - f_{uv}$ units of flow and in the opposite direction to be skipped f_{uv} flow units, which would mean flux cancellation in the initial direction.

Blocking flow in this network is called a torrent that any path from the source s in stock t contains saturated this stream edge. In other words, the network can not find such a path from the source to the drain, along which you can easily increase the flow.

Blocking flow is not necessarily maximal. Ford-Fulkerson theorem says that the flow will be maximal if and only if in the residual network is not there $s - t$ the way; in blocking same thread says nothing about the existence of the path along the edges that appear in the residual network.

Layered network for the network is constructed as follows. First the length of the shortest paths from the source to all other vertices; call level $\text{level}[v]$ top of its distance from the source. Then layered network include all those edges (u, v) the source network which lead from one level to another, (Why in this case the difference of distances is later, the level, i.e. $\text{level}[u] + 1 = \text{level}[v]$ not can exceed unity follows from the properties of the shortest distances). Thus, all the edges are removed, located entirely within the levels, as well as the edges leading back to previous levels.

Obviously, a layered network is acyclic. In addition, any $s - t$ path in a layered network is the shortest path

in the home network.

Build a layered network of the network is very easy: you need to run wide detour along the edges of the network, thereby considering each vertex value $\text{level}[v]$. And then make a layered network is suitable ribs.

Note. The term "layered network" in Russian literature is not used; this design is usually referred to simply as "auxiliary graph." However, the English term commonly used "Layered network".

Algorithm

Diagram

The algorithm consists of several **phases**. Each phase is constructed first, the residual network, and then in relation to it being built layered network (preorder traversal), and it sought an arbitrary blocking flow.

Found blocking the flow added to the current flow, and that the next iteration ends.

This algorithm is similar to the Edmonds-Karp algorithm, but the main difference can be understood as follows: at each iteration, the flow does not increase along one of the shortest $s - t$ path, and along the whole set of such paths (because it is in such ways are the ways in blocking the flow of a layered network).

Correctness of the algorithm

We show that if the algorithm terminates, the output stream he gets it maximum value.

In fact, suppose that at some point in the layered network built for the residual network, failed to find a blocking flow. This means that stock t generally not achievable in a layered network of the source s . But as a layered network contains all the shortest paths from the source in the residual network, which in turn means that there is no residual network path from the source to the drain. Consequently, applying the Ford-Fulkerson theorem, we see that the current flow in fact maximal.

Estimating the number of phases

We show that the algorithm always performs Diniz less n phases. For this we prove two lemmas:

Lemma 1. The shortest distance from the source to each vertex does not decrease with the implementation of each iteration, ie

$$\text{level}_{i+1}[v] \geq \text{level}_i[v]$$

where the subscript denotes the number of phases, which are taken to the values of these variables.

Proof. Fix an arbitrary phase i and an arbitrary vertex v and consider any shortest $s - v$ path P . Online (recall so we denote the residual network, taken before

implementation $i + 1$ (The second phase). Obviously, the path length P equal $\text{level}_{i+1}[v]$.

Note that in the residual network may include only the edges of the And ribs, the ribs of the inverse

(This follows from the definition of the residual network). Consider two cases: G^R contains only the edges of G^R . Then, of course, the path

- Path the length greater than or equal to (Because that P by definition G^R the length of the shortest path) which means that the inequality.
- Path v comprises at least one rib that is not contained in the (But the opposite edge of some). Consider the first such edge; let it be an edge

We can apply our lemma to the top, because it falls under the first case; So we we obtain

$$\text{level}_{i+1}[u] \geq \text{level}_i[u]$$

Now, note that since the rib (u, w) appeared in the residual network only after phase, it follows that along the edge was further omitted some thread; Consequently, the ribs belonged to a layered network before the second phase, and

(w, u) therefore $\text{level}_i[w] = \text{level}_i[u] + 1$. Consider that property on the shortest paths $\text{level}_{i+1}[w] = \text{level}_{i+1}[u] + 1$ And combining this with two previous inequalities, we obtain:

$$\text{level}_{i+1}[w] \geq \text{level}_i[w] + 2.$$

Now we can apply the same reasoning to the remaining path to v (ie, that each adds to the inverted rib level at least two), and finally obtain the required inequality.

Lemma 2. The distance between the source and the drain increases strongly after each phase of the algorithm, ie:

$\text{level}'[t] > \text{level}[t]$,

where the prime marked value obtained on the next phase of the algorithm.

Proof: by contradiction. Assume that after the current phase appeared, that. Consider the shortest path

from the source to the drain by assumption, its length

should remain unchanged. However, the residual network for the next phase contains only the edges of the residual network before executing the current phase or reverse them. Thus, a contradiction: found $s - t$ a path that does not contain saturated edges and has the same length as the shortest path. This path should be "locked" blocking the flow of what had happened, and what is a contradiction, as required.

This lemma can be understood intuitively as follows: on i The second phase of the algorithm identifies and Diniz pervades all $s - t$ path length i .

Since the length of the shortest path from s in t can not exceed $n - 1$ Then, consequently, the algorithm performs Dinitz **max $n - 1$ phase**.

Search blocking flow

To complete the construction of an algorithm Diniz, must describe the algorithm for finding a blocking flow in a layered network - a key place algorithm.

We consider three possible embodiment search blocking flow:

- Search $s - t$ way one by one until such ways are. Way can be found for a circuit in depth, and all such paths is (because each path saturates at least one edge). Total asymptotics $O(m)$.
Search a blocking flow amount $O(m^2)$.
- Similarly to the previous idea, but removed during traversal of the graph in depth all the "extra" edge, ie edges along which will not reach the drain.

It is very easy to implement: just delete the edge after we viewed it in the crawl depth (except in the case when we walked along the edge and found a way to drain). In terms of implementation, it is necessary simply to maintain the adjacency list of each vertex pointer to the first undeleted edge and increase this point in the cycle within the traversal depth.

We estimate the asymptotic behavior of the solutions. Each tour ends in depth or saturation of at least one edge (if the bypass flow reached) or moving forward at least one direction (otherwise

case). One can understand that one start crawling into the depths of the main program runs for $O(k + n)$ Where - by advances pointers. Given that only runs deep traversal within one search

blocking flow will $O(p)$ Where - the number of edges, it blocks the flow of saturated, then the whole

search algorithm for blocking the flow of exhaust $O(pk + pn)$ of

gone the distance $O(m)$, Gives an asymptotic $O(n + pn)$. In the worst case, when the blocking flow saturates

all edges asymptotics obtained $O(nm)$; this asymptotic behavior and will be used on.

We can say that this method of finding a blocking flow is extremely effective in the sense that increasing the search one way he spends an average of operations. Therein lies the difference on

an order of effectiveness of the algorithm Diniz and Edmonds-Karp (who is looking for a way for increasing $O(m)$).

This method of solution is still simple to implement, but effective enough, and therefore most often used in practice.

- You can apply a special data structure - dynamic Sletora trees (Sleator) and Tarjan (Tarjan)). Then each blocking flow can be found in time $O(m \log n)$.

Asymptotics

Thus, the entire algorithm runs in Diniz $O(n^2m)$ If blocking the flow of search described the above method for $O(nm)$. Implementation using dynamic trees Sletora Tarjan and will work during $O(nm \log n)$.

Single network

Unit called such a network in which all capacities are equal to 0 or 1, and each vertex either incoming or outgoing edge only.

This case is important enough, because the problem of finding **maximum matchings** network is built exactly the unit.

We prove that the algorithm for single networks Diniz even a simple implementation (which is an arbitrary graph fulfills for $O(nm)$) works in time, reaching to the problem of finding maximum matching of the best known algorithms - Hopcroft-Karp algorithm. To prove

this assessment, it is necessary to consider two cases:

- If the value does not exceed the desired flow

In this case, we find that the number of phases and starts crawling in depth is the value one phase in this embodiment operates in $O(\sqrt{n})$. Recalling that asymptotics $O(\sqrt{nm} + \sqrt{nn}) = O(\sqrt{nm})$. Obtain final $O(\sqrt{n})$.

- If the

desired flow more

$$|f| > \sqrt{n}$$

Note that flow in a single network can be represented as a sum of vertex-disjoint paths, and because the maximum length path has a value $|f|$. Given that one phase of the algorithm Diniz whole processes all the way any length, we again find that the number of phases is the value $O(\sqrt{n})$. Summarizing the asymptotic behavior of one phase all phases obtain $O(\sqrt{nm} + n^2) = O(\sqrt{nm})$. That

QED.

Implementation

We present two implementations of the algorithm for Running on the network, given the adjacency matrices and adjacency lists, respectively.

Implementation on graphs as adjacency matrices

```

const int MAXN = ...; // the number of vertices
const int INF = 1000000000; // constant-Infinity

int n, c[MAXN][MAXN], F[MAXN][MAXN], S, t, d[MAXN], Ptr[MAXN], Q[MAXN];

bool bfs() {
    int qh = 0, qt = 0; q[qt] = S;
    memset(d, -1, N * sizeof d[0]);
    d[S] = 0;
    while (qh < qt) {
        int v = q[qh];
        for (int to = 0; to < n; ++to)
            if (d[to] == -1 && F[v][to] < c[v][to])
                { q[qt] = to; d[to] = d[v] + 1; }
    }
    return d[t] != -1;
}

int dfs(int v, int flow) {
    if (!Flow) return 0;
    if (v == t) return flow;
    for (int & To = ptr[v]; to < n; ++To) {
        if (d[To] != D[v] + 1) continue;
        int pushed = dfs(to, min(flow, c[v][to] - F[v][to]));
        if (pushed) {
            f[v][to] += pushed;
            f[to][v] -= pushed;
            return pushed;
        }
    }
    return 0;
}

int dinic() {
    int flow = 0;
    for (;;) {
        if (!Bfs()) break;
        memset(ptr, 0, N * sizeof ptr[0]);
        while (int pushed = dfs(s, INF))
    }
}

```

```
    flow += pushed;
}
return flow;
```

The network must be pre-read: must be specified variables
matrix of capacities c_{ij} - As well as read
found the maximum flow value.

The main function of the decision $n \leftarrow s \rightarrow t$
 $\text{dinic}()$

Which returns

Implementation on graphs as adjacency lists

```

const int MAXN = ...; // the number of vertices
const int INF = 10000000000; // constant-Infinity

struct edge {
    int a, b, cap, flow;
};

int n, s, t, d[MAXN], Ptr[MAXN], Q[MAXN];
vector<edge> e;
vector<int> G[MAXN];

void add_edge (int a, int b, int cap) {
    edge e1 = { a, b, cap, 0 }; edge
    e2 = { b, a, 0, 0 }; g[a].
    Push_back ((int) e.size());
    e.push_back (e1);
    g[b]. Push_back ((int)
    e.size()); e.push_back (e2);
}

bool bfs() {
    int qh = 0, qt
    = 0; q[qt + +] =
    S;
    memset (d, -1, N * sizeof d[0]);
    d[s] = 0;
    while (qh < qt && d[t] == -1) {
        int v = q[qh + +];
        for (size_t i = 0; i < g[v]. Size(); + +
            I) { int id = g[v] [i],
            to = e[id]. B;
            if (d[to] == -1 && E[id]. Flow < e[id]. Cap)
                { q[qt + +] = To;
                d[to] = D[v] + 1;
                }
            }
        }
    return d[t] , = -1;
}

int dfs (int v, int flow) {
    if (! Flow) return 0;
    if (v == t) return flow;
    for (; ptr[v] < (int)g[v]. Size(); + +
        Ptr[v]) { int id = g[v] [ptr[v]],
        to = e[id]. B;
        if (d[to] != D[v] + 1) continue;
        int pushed = dfs (to, min (flow, e[id]. Cap - e[id].
        Flow)); if (pushed) {
            e[id]. Flow += pushed;
            e[id ^ 1]. Flow -=
            pushed; return pushed;
            }
        }
    return 0;
}

int dinic() {
    int flow = 0;
    for (;;) {
        if (! Bfs()) break;

```

```
memset (ptr, 0, N * sizeof ptr[0]);
```

```
        while (int pushed = dfs (s, INF) )
              flow += pushed;
    }
    return flow;
}
```

The network must be pre-read: must be specified variables n, s, t . And adding all edges (oriented) using function calls `add_edge` The main function of the `main()` which returns the value of the maximum flow found.

Kuhn's algorithm for finding the greatest matching in a bipartite graph

G

Dan bipartite graph containing n and peaks m Ribs. Required to find a maximum matching, ie select as many edges to none selected edge had no common vertex with any other selected edge.

Description of the algorithm

Necessary definitions

A matching M is a set of pairwise non-adjacent edges of the graph (in other words, any vertex of the graph must be incident to at most one edge of the set M). Power matching call number of edges in it. The greatest (or maximum) matching is called matching, whose power is maximum among all possible matchings in this graph. All those vertices that have adjacent edges of matchings (ie, which have exactly one degree in the subgraph formed M), This is called a matching saturated.

Chain length called some simple way (I.e., does not contain duplicate vertices or edges) comprising k Ribs.

Alternating chain (In the bipartite graph, with respect to some matching) call chain, in ribs which in turn owns / not belong to matching.

Increasing the chain (In the bipartite graph, with respect to some matching) call alternating chain, whose initial and final vertices do not belong to matching.

Berge's theorem

Formulation. Matching the maximum if and only if When there is no increasing chain relative thereto.

Proof of necessity. Show if matching M maximum, then there is a relatively magnifying circuit. This proof will be constructive: we show how to increase by this magnifying circuit P Power matchings M unit.

To do this, perform the so-called alternating matchings along the chain P . We remember that, by definition, the first edge of the path P does not belong to matching, the second - belongs to the third - again does not belong to the fourth - owned, etc. Let's change the status of all edges along the chain: those edges that are not included P b matching (the first, third and so on until the last) are included in the matching, and the edges that were previously b matching (the second, fourth and so on until the penultimate) - remove from him.

It is clear that the power of matching thus increased by one (because it was added to one edge more than deleted). It remains to verify that we have built a correct matching, ie that no vertex has no right two adjacent edges of this matching. For all vertices alternating chain P Except the first and last, it follows from the alternation of the algorithm: first, we have removed the top of each such adjacent edges, then added. For the first and last vertex chain P and nothing could broken as to interleave they had to be unsaturated. Finally, for all other vertices

- Non-chain P - Obviously, nothing has changed. Thus, we actually built a matching, and one greater power than the old one, which completes the proof of necessity.

The proof of sufficiency. We prove if under some matchings M No increase the ways it - the maximum.

Proof by contradiction. Suppose there is a matching having more power than M . Consider the symmetric difference of these two matchings, ie leave all the edges included in M or M' , But not both simultaneously.

It is clear that the set of edges - is certainly not matching. Consider what kind of a set of edges has; for convenience we will treat it as a graph. In this graph, each vertex obviously has degree 2 (because each node can have a maximum of two adjacent edges - one matchings and from another). Easy to understand that if this graph consists only of cycles or paths, and neither one nor the other do not intersect with each other.

Now, note that the way in this column may not be any, but only even length. In fact, any path in the graph alternate ribs: the ribs of the Q post M edge goes from M' And vice versa. Now, if we look at some way of odd length in the graph Q , It turns out that in the original graph G it will increase the chain or

for matchings M , Or for M' . But this could not be, because in the case of matching M this contradicts with the proviso in the case M' . At its maximum (in fact we have already proved the necessity of the theorem from which it follows that the existence of increasing the matching circuit can not be maximal).

We now prove the analogous statement for cycles and all cycles in the graph can only have an even length. It is quite easy to prove: it is clear that the cycle edges also must alternate (owned by turns M , That is), but this condition can not be executed in a cycle of odd length - it sure there are two adjacent edges of a matching, which contradicts the definition matching.

Thus, all the paths and cycles of Q are of even length. Therefore, the graph contains an equal number of edges of M and M' . But given that $Q = M \oplus M'$ contains all the edges of M and M' , except for their common edges, it follows that the power $d_M = d_{M'}$ coincide. We have a contradiction: by assumption, matching M was not the maximum, then the theorem is proved.

Kuhn algorithm

Kuhn's algorithm - the direct application of the theorem of Berge. It can be briefly described as follows: first, take the empty matching, and then - until the graph can not find magnifying chain - will perform alternating matchings along the chain, and repeat the process increases the search chain. Once such a chain can not find - the process stops - the current matching is maximum.

You're finding a way to increase the detail circuits. **Kuhn algorithm** - Just looking for any of these circuits using **traversal depth or across**. Kuhn's algorithm looks at all the vertices graph in turn, starting from each tour, trying to find a magnifying circuit, starting at this vertex.

Convenient to describe the algorithm, assuming that the graph is split into two parts (though in fact the algorithm can be implemented and so that he was not given the input graph is clearly divided into two lobes).

The algorithm looks at all the vertices v the first part of the graph. If the current vertex v current matching is already saturated (ie already selected some edge adjacent to it), then skip this summit. Otherwise

- The algorithm tries to satiate this summit, which starts searching magnifying chain, starting with this vertex.

Search magnifying circuit by means of a special bypass in depth or width (usually for ease of implementation is used to bypass the depth). Originally worth a detour to the depth in

current unsaturated top v the first part. We examine all the edges of the vertex, let the current edge - this edge (v, p) . If the top v matching is not yet saturated, it means that we were able to find magnifying circuit; it consists of a single edge (v, p) ; in this case, simply turn this edge (v, p) already saturated with to stop and search matching magnifying chain of peaks. Otherwise - if the edge (v, p) is already saturated, how-then edge (p, to) . Then try to pass along this edge: thus we will try to find a magnifying chain extending through the first (v, p) to the second. To do this, simply go to our crawl to the top - now we try to find the magnifying circuit from this vertex.

One can understand that as a result of this tour, running from the top v or will magnifying circuit and thereby saturate top v or as a magnifying circuit not find (and, hence, this peak v is not able to become saturated).

Once all vertices $v = 1 \dots n_1$ will be reviewed, the current matching is maximized.

Operation time

Thus, the algorithm can be represented as HRK series of n starts crawling depth / width of all column. Consequently,

only the algorithm is executed during $O(nm)$. That in the worst case, there is $O(n^3)$.

However, this estimate may be a little **improve**. It turns out that it is important for the algorithm Kuhn what portion is selected for the first, and which - in the second. In fact, in the above implementation starts crawling depth /

width of the peaks occur only the first part, so the whole algorithm is executed during $O(n_1 n_2)$. Whe rein n_1 (Wh - Number of vertices of the first share).

- The number of vertices of the first part. In the worst case, it is $O(n_1^2 n_2)$ share. This shows that more profitable when the first fraction contains fewer vertices than the second. Very unbalanced graphs (when n_1 and n_2 differ), this translates into a significant difference at work.

Implementation

We present here the implementation of the above algorithm, based on a detour into the depths, and the host in the form of a bipartite graph clearly broken into two parts of the graph. This implementation is very concise, and perhaps it is worth remembering in this form.

Here n - The number of vertices in the first part, k - In the second part - from the top of a list of edges v the first part (ie, a list of numbers of vertices that are of these edges v). Vertices in both lobes are numbered independently, ie first share - with numbers $1 \dots n$, The second - with numbers $1 \dots k$.

Then there are two auxiliary array: **mt** and **used**. First - **mt** contains information about the current matching. For programming convenience, this information is contained in only the second for the vertices

- Is the number of vertices of the first part, associated with the top edge of the second part
share **mt[i]** (or no edges of matchings not in). The second array - **used** - The usual array of "visit" vertices
Bypassed in depth (need it just to bypass the depth does not extend into one vertex twice).

Function **try_kuhn** - And there is a detour to the depth. It returns **true**, if she could find magnifying circuit from the top **v** When it is considered that this function has already made alternate matchings found along the chain.

Inside the function scans all the edges emanating from the vertex **v** the first part, and then checks if it unsaturated rib leads into the top, or if the vertex **to** saturated, but can not find the magnifying recursive chain starting from **mt[to]**. Then we say that we found the magnifying circuit and before returning with the result of the function **try_kuhn**, produce alternating current in Fin redirect edge adjacent to, on top. **v**

In the main program first indicates that the current matching - empty (list **-1**). Then moves the vertex of the first part of it and started crawling in depth **try_kuhn**, pre-zeroed array **used**.

It should be noted that the size of matchings easily get the number of calls **try_kuhn** in the main program who returned result **true**. Deposit required maximal matching in the array **mt**.

```
int n, k;
vector <vector <int>> G;
vector <int> Mt; vector
<char> Used;

bool try_kuhn (int v) {
    if (Used[v]) return false;
    Used[v] = true;
    for (size_t i = 0; i < g[v].Size(); ++
        I) { int to = g[v][i];
        if (Mt[to] == -1 || Try_kuhn (Mt[to])) {
            Mt[to] = v;
            return true;
        }
    }
    return false;
}

int main() {
    Graph reading ... ...

    mt.assign (k, -1);
    for (int v = 0; v < n; ++ V) {
        used.assign (n, false);
        try_kuhn (v);
    }

    for (int i = 0; i < k; ++
        I) if (Mt[i] == -1)
        printf ("% D% d \ n", Mt[i]+1, I +1);
}
```

Once again, the algorithm is easy to implement and Kuhn so that he worked on graphs for which we know that they are bipartite, but their clear division into two lobes found. In this case it is necessary to abandon the convenient partition into two parts, and all the information stored for all vertices. For this array of lists **g** now set not only for the vertices of the first part, but for all the vertices of the graph (of course, now tops both lobes are numbered in total numbering - from **1** to **n**). Arrays **mt** and now also defined for the vertices of both shares and, therefore, they must be maintained in this state.

An improved

We modify the algorithm as follows. Prior to the main loop of the algorithm will find some simple algorithm **arbitrary matching** (Simple **heuristic algorithm**), And only then will perform a cycle with function calls **kuhn ()**, which will improve this matching. In

, the algorithm will run much faster on random graphs - because in most graphs can easily dial matching sufficiently large weight through heuristics, and then improve to the maximum found matching algorithm has the usual Kuhn. Thus we will save on

starts crawling depth of the vertices that we include with the heuristics in the current matching.

For example, you can just iterate through all the vertices of the first part and for each of them to find any edge that can be added to matching, and add it. Even such a simple heuristic algorithm is able to accelerate Kuhn several times.

It should be noted that the main loop will have to slightly modify. Since the function call in the main loop is assumed that the current node is not yet included in the matching, then
try_kuhn
need to add an appropriate test.

In implementing change only the code in the function main ():

```
int main() {
    Graph reading ... ...

    mt.assign (k, -1);
    vector <char> Used1
    (n); for (int i =0; i
    <n; + + I)
        for (size_t j =0; j <g[i]. Size(); +
            + J) if (mt[g[i] [j]] == -1)
        {
            mt[g[i] [j]] =
            I; used1[i] =
            true;
            break;
        }
    for (int i =0; i <n; + + I) {
        if (used1[i]) continue;
        used.assign (n, false);
        try_kuhn (i);
    }

    for (int i =0; i <k; + +
        I) if (mt[i] ,
        -1)
        printf ("% D% d \ n", Mt[i]+1, I +1);
}
```

Another good heuristics is the following. At each step, you will seek the top least likely (but not isolated), choose any of her edge and add it to the matching,
then removing both these vertices with all their incident edges of the graph. Such greed works very well for random graphs, even in most cases builds a maximal matching (though against her there is a test in which it finds a matching value significantly lower than the maximum).

Checking graph on bipartition and split into two parts

Suppose we are given an undirected graph. Want to check whether it is bipartite, ie whether it is possible to divide it into two parts, the vertices so that there are no edges connecting two vertices of one share. If the graph is bipartite, then withdraw themselves share.

Solve this problem by using search width in O (M).

Sign dipartition

Theorem. A graph is bipartite if and only if all its simple cycles are of even length.

However, from a practical point of view to look all simple cycles uncomfortable. Much easier to test graph on bipartition the following algorithm:

Algorithm

We carry out a series of searches in width. Ie will run breadth-first search from each unvisited vertex. That vertex from which we start walking, we put in the first part. In searching the width, if we go to some new vertex, then we put it in a fraction different from the current share of the top. If we try to pass on to the top edge, which has already had, then we check to the vertex and the current vertex were in different proportions. Otherwise, the graph is not bipartite.

At the end of the algorithm, we either find that the graph is not bipartite, or find a partition of vertices into two parts.

Implementation

```
int n;
vector <vector <int>> g;
Graph reading ... ...

vector <char> part (n, -1); bool ok = true; vector <int> q (n);
for (int st = 0; st <n; + +
    st) if (part [st] == -1) {
    int h = 0, t = 0; q [t + +] = st; part [st] = 0;
    while (h <t)
    {
        int v = q [h + +];
        for (size_t i = 0; i <g [v]. size ();
            + + i) {int to = g [v] [i];
        if (part [to] == -1)
            part [to] = ! part [v], q [t + +] = to;
        else
            ok &= part [to] != part [v];
        }
    }
}
puts (ok? "YES": "NO");
```

Finding the greatest weight vertex-weighted matchings

Given bipartite graph G. For each vertex of the first part of its specified weight. Required to find a matching of maximal weight, ie with the largest sum of weights of saturated vertices.

Below we describe and prove algorithm based on [Kuhn algorithm](#) That will find the optimal solution.

Algorithm

The algorithm is extremely simple. **Sort the** top of the first part in descending order (more precisely, non-increasing) weights, and apply to the resulting graph [Kuhn algorithm](#).

It is alleged that obtained with the maximum (in terms of number of ribs) and the matching is optimal in terms of sums of weights saturated vertices (despite the fact that after sorting, we do not actually use these weights).

Thus, the implementation will be something like this:

```
int n;
vector <vector <int>> g (n);
vector used (n);
vector <int> order (n); // Vertex list, sorted by weight
Reading ... ...

for (int i = 0; i <n; + +
    i) {int v = order
        [i];
        used.assign (n, false);
        try_kuhn (v);
    }
```

Function `try_kuhn ()` is taken without any changes of the algorithm Kuhn.

Proof

Recall the basic provisions **matroid theory**.

Matroid M - is an ordered pair (S, I) , where S - a set, I - a non-empty family of subsets of S, which satisfy the following conditions:

1. The set S is finite.
2. Family I is hereditary, ie if one set belongs to I, then all its subsets also belong to I.
3. M has the structure of an exchange, i.e. if $A \in I$, and $B \in I$, and $|A| < |B|$, there exists an element $x \in A - B$, that $A \cup x \in I$.

Elements of the family I called independent subsets.

Called the weighted matroid if for every element $x \in S$ defined on some weight. Weight subset is the sum of the weights of its elements.

Finally, the most important theorem in the theory of weighted matroid: to get the best response, ie independent subset with the largest weight, you need to act greedily, starting with the empty subset, we add (unless, of course, the current item can be added without violating the independence) all the elements one by one in order of decreasing (or rather, non-increasing) their weights:

```
sort the set S in non-increasing weight; ans =
[];
foreach (x in S)
    if (ans ∪ x ∈ I)
        ans = ans ∪ x;
```

It is alleged that at the end of this process we obtain a subset with the largest weight. Now

proveThat our task - neither more nor less than the weighted **matroid**.

Let S - set of all vertices of the first part. To keep our task in a bipartite graph with respect to matroid vertices of the first part, we assign each matchings is a subset S,

which is the set of saturated vertices of the first part. You can also define and inverse correspondence (From the set of vertices saturated - in matching), which, although not unique, but we will be quite happy.

Then I define the family as a family of subsets of S, for which there is at least one corresponding matching.

Next, each element of S, i.e. for each vertex of the first part, determined by the condition of some weight. And the weight of a subset, as we required in terms of the theory of matroids is defined as the sum of weights of the elements in it.

Then the problem of finding the maximum weight matchings now be reformulated as a problem of finding the maximum weight independent subset.

It remains to verify that the following three conditions above imposed on matroid. Firstly, it is obvious that S is finite. Secondly, it is obvious that the removal edges of the removal is equivalent to matching vertices of a plurality of vertices saturated, and therefore the property of inheritance is performed. Thirdly, as the algorithm is correct Kuhn, if the current matching is not possible, then there always exists a vertex, which can be satiate, without removing from the set of saturated vertices other peaks.

Thus we have shown that our task is a weighted matroid with respect to the set of saturated vertices of the first part, but because it is applicable to the greedy algorithm.

It remains to show that **Kuhn this algorithm is a greedy algorithm**.

However, it is quite obvious. Kuhn algorithm at each step of trying to saturate the current node - or just spending edge in the top of the second beat of unsaturated or finding a lengthening chain and matching alternating along it. And in fact, in any other case already saturated vertices not cease to be unsaturated and unsaturated vertices in the previous steps of the first part and not saturated at this step. Thus, the algorithm is greedy Kuhn algorithm for constructing an optimal subset of the matroid independent, which completes our proof.

Edmonds algorithm for finding the greatest matchings in arbitrary graphs

Dan unweighted undirected graph with n vertices. You want to find in it a maximum matching, etc. this is the largest (in power), the set M its edges, that no two edges of the selected not incident each other (ie, have no common vertices).

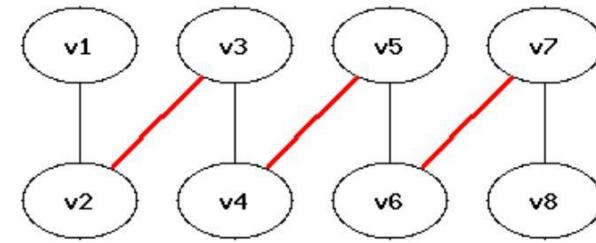
Unlike the case of a bipartite graph (see [Kuhn algorithm](#)), In the column G may be present cycles of odd length, which greatly complicates the search for ways of increasing.

We first give a theorem of Berge, from which it follows that, as in the case of bipartite graphs, maximum matching can be found using the increasing ways.

Increase path. Berge's theorem

Let recorded a matching M . Then a simple chain (v_1, v_2, \dots, v_k) called alternating chain if it turns edges belong - do not belong matchings

M . Alternating chain called magnifying if its first and last vertices belong matching. In other words, a simple chain P a magnifying if and only if when the top $v_1 \notin M$, $\text{Rib}(v_2, v_3) \in M$, $\text{Rib}(v_4, v_5) \in M, \dots \text{Rib}(v_{k-2}, v_{k-1}) \in M$, and the top $v_k \notin M$.



Berge's theorem (Claude Berge, 1957 g). Matching M is the largest, and then only if it does not exist for increasing the chain.

Proof of necessity. Suppose that for matchings there is an increasing chain . We show how to go to the matchings more power. Perform alternating matchings along this circuit. i.e. include in the matching edges $(v_1, v_2), (v_3, v_4), \dots, (v_{k-1}, v_k)$ from matching edges $(v_2, v_3), (v_4, v_5), \dots, (v_{k-2}, v_{k-1})$. The result is likely to be obtained correct matching, whose power will be one higher than the matching M (Since we added $k/2$ ribs and removed $k/2 - 1$ edge).

The proof of sufficiency. Suppose that for matchings there magnifying chain, prove that it is the greatest. Let M - A maximum matching. Consider symmetric difference $\overline{G} = M \oplus \overline{M}$ (ie, the set of edges belonging to either M or \overline{M}). But not both simultaneously). We show that \overline{G} contains the same number of ribs as M and \overline{M} (Because we excluded G). Only their common edge, then it will follow and $|M| = |\overline{M}|$. Note that \overline{G} consists of simple paths and cycles (since otherwise one would be the top two edges incident immediately any matchings which is impossible). Further, the cycles may not have an odd length (for the same reason). Chain neither may have an odd length (otherwise she was for increasing the chain). That contradicts or for . That is contrary to its maximum). Finally, circuits, and even cycles even length M ribs alternately included M and \overline{M} , Which means that \overline{G} occurs the same number of edges from M and \overline{M} . As mentioned above, it follows that $|M| = |\overline{M}|$ ie M is a maximum matching.

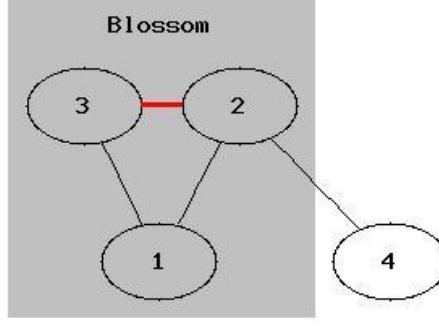
Berge's theorem provides the basis for the algorithm Edmonds - find increasing chains and striping along them until the chains are increasing.

Edmonds algorithm. Compression flowers

The main problem is how to find the path increases. If there are cycles in the graph of odd length, then just run a detour to the depth / width is impossible.

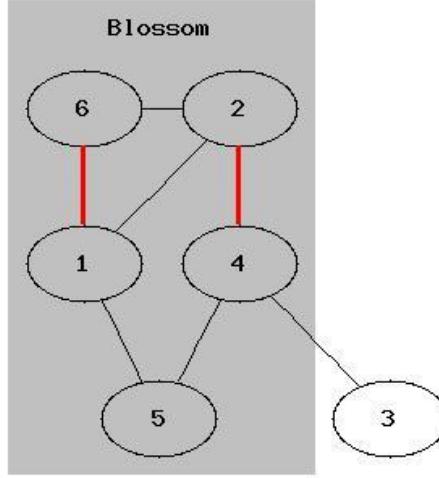
You can give a simple counterexample when run from one of the vertices algorithm does not handle the special

odd length cycles (actually, [Kuhn algorithm](#)) Finds increasing path, although it should. It is a cycle of length 3 with hanging on the edge of it, ie Count 1-2, 2-3, 3-1, 2-4, and 2-3 edge taken in matching. Then when you start from the top 1 if bypass goes first in the top two, he "rested" in the top 3, instead of finding a magnifying circuit 1-3-2-4. However, for this example, when you run out of the top 4 algorithm Kuhn still find this increasing chain.

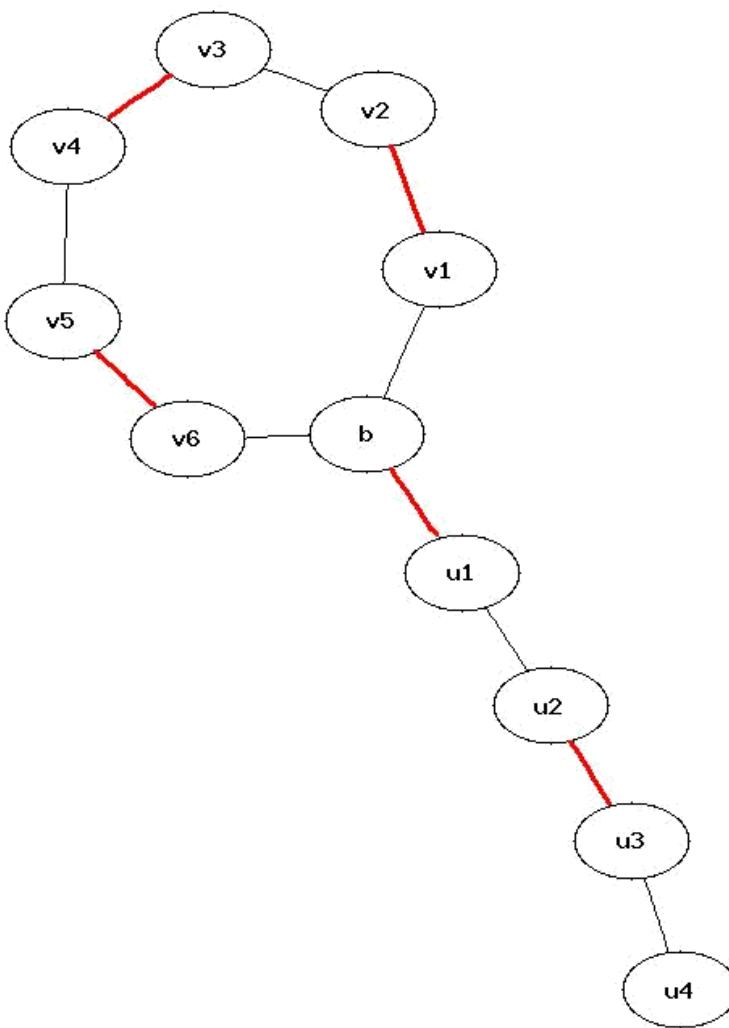


Nevertheless, it is possible to construct a graph in which at a certain order in the adjacency lists algorithm Kuhn deadlocked.

first 4, and then to 1, and when you start it from the top three from the top two will go first to one, and only then to 6).



As we have seen in this example, the problem is that when released into the cycle of odd length bypassing can go on a cycle in the wrong direction. In fact, we are only interested in "saturated" cycles, ie in which there is saturated edges, where the cycle length is $2k+1$. In this cycle, there is exactly one vertex not saturated edges of this cycle, we call it **base** (Base). To the top of the base is suitable alternate path even (possibly zero) length, starting at the free (ie not owned matchings) top, and this way is called **stem** (Stem). Finally, the subgraph formed by the "saturated" odd cycle, called **flower** (Blossom).



Idea of the algorithm Edmonds (Jack Edmonds, 1965) - in **compression flowers** (Blossom shrinking). Compression of the flower - the contraction of all the odd cycle in a pseudo-vertex (respectively, all the edges incident to the vertices of the cycle are incident pseudo-top). Edmonds algorithm searches the graph all the flowers, compresses them, and then the graph is not "bad" cycles of odd length, and such a graph (called "surface" (surface)

graph) can already look magnifying simple bypass circuit in depth / width. After finding increasing the chain in the surface graph must "develop" the flowers, thereby restoring the magnifying circuit in the original graph.

However obvious that compression can not be broken flower graph structure, namely that if the space existed chain increases, it also exists in the graph \overline{G} . Obtained after the compression of a flower, and vice versa.

Edmonds theorem. Column \overline{G} magnifying circuit exists if and only if increases in the circuit exists.

\overline{G} was obtained from the graph of a single flower compression (denoted

Proof. Thus, let graph \overline{G} by G cycle of a flower, and B through the proper compressed vertex), we prove the theorem. First we note it suffices to consider the case when the base of the flower is a free vertex (not belonging to the matchings). Indeed, otherwise the flower in the database ends alternating path of even length, starting at the top free. Procheredovav matching along this path,

Power matching will not change, and the base of the flower will be a free vertex. So, in the proof we can assume that the base of the flower is a free vertex.

Proof of necessity. Let path passes through, then obviously it will increase in the graph. Let can without loss of generality assume that the path , Plus some trail running along the tops will be in the column by increasing the , If he does is increasing in the graph not extends through . Then is some way Not passing over the tops and possibly other vertices. But then the way , As required.

The proof of sufficiency. Let path does not pass through, the if the path path case will not be considered. We consider separately the case where begins with compressed flower, ie has the form . Then in flower there is a corresponding vertex, which is connected (unsaturated) with an edge . It remains only to note that

$$\overline{P} \quad \overline{B} \quad \overline{P} \quad \overline{P} \quad G \quad \overline{G}$$

$$\overline{P} \quad \overline{B} \quad (\overline{B}, c, \dots)$$

From the base of the flower is always there even length alternating path to the top. All things considered above, we find that the path $P = (b, \dots, v, c, \dots)$ is by increasing the v in the graph.

Suppose now that the path \overline{P} passes through a pseudo-top \overline{B} . But do not begin and end there. Then two edges which pass through \overline{B} , even if it (a, \overline{B}) and (\overline{B}, c) . One of them is necessarily belong matchings M . However, since base of the flower is not saturated, and all other vertices of the flower B saturated edges cycle, we arrive at a contradiction. Thus, this case is simply impossible. Thus, we examined all cases, and in each of them showed Theorem Edmonds.

General diagram Edmonds takes the following form:

```

void edmonds() {
    for (int i = 0; i <n; + + I)
        if (top i not matchings) {
            int last_v = find_augment_path (i);
            if (last_v! = -1)
                perform striping along the path from i to last_v;
        }
}

int find_augment_path (int root) {
    wide detour:
        int v =
        tekuschaya_vershina;
        iterate over all edges
        of v
            if found a cycle of odd length, compress it if
            came to the free vertex return
            if you come to the non-free vertex, then add
                it to the queue in the adjacent
                matchings
    return -1;
}

```

Effective implementation of

Immediately evaluate the asymptotic behavior. Altogether there n iterations, each of which crawls in width for

Moreover, the compression operation can be flowers - they can be . Thus, if

we learn how to squeeze a flower , The total amount to the asymptotic behavior of behind the algorithm $O(n)$ $O(n \cdot (n + n^2)) = O(n^3)$.

The main difficulty of compression operations are flowers. If you execute them directly combining adjacency lists and removing one of the extra vertices of the graph, then the asymptotic behavior of compression will be a single flower $O(m)$,

In addition, difficulties arise when "unfolding" of flowers.

Instead, we shall, for each vertex of the graph to maintain a pointer to the base of the flower, to which it belongs (or yourself, if node does not belong to any of the flower). We need to solve two problems: the compression

flower for when it is detected, as well as convenient storage of all information for later increasing the alternation along the way.

So, one iteration of the algorithm is a circumvention Edmonds width executed from a given free vertices root . Will gradually build a tree traversal in width, and the path to it any vertices will be alternating path starting with a free vertex root . For ease of programming will be put in place only those peaks, the distance to which the tree is even ways (we call such vertices even - that is the root of the tree, and the second ends of edges in matchings). The tree itself will be stored in an array of ancestors, in which for each odd vertex (ie before $p[]$) which the distance in the tree odd ways, ie it first ends of edges in matchings) will store ancestor - even vertices. Thus, the recovery path of the tree, we need to use the arrays alternately $p[]$

and $\text{match}[]$ Wherein $\text{match}[]$ - For each vertex adjacent to it contains a matching problem, or -1 if one is not.

Now it becomes clear how to detect cycles of odd length. If we are out of the current top v while crawling across arrive at a vertex u , Is the root root or belonging matchings and tree paths (ie which is not equal to -1), we found a flower. Indeed, under these conditions $\text{match}[]$

and the top v , And the top u are even vertices. Distance from them to their lowest common ancestor has one parity, so we found a cycle of odd length.

Learn to **compress cycle**. So, we found an odd cycle when considering edges where u and v - even vertices. Find their least common ancestor, and he will be the base of the flower. Easy to see that the base is too even vertex (as odd vertices in the tree paths have only one son). However, it should be noted that - it may pseudovertex, so we actually find the base of the flower,

is the lowest common ancestor of vertices v and u . Implement immediately to find the lowest common ancestor (we are quite satisfied asymptotics $O(n)$)

```
int lca (int a, int b) {  
    bool used[MAXN] = { 0 };
```

```

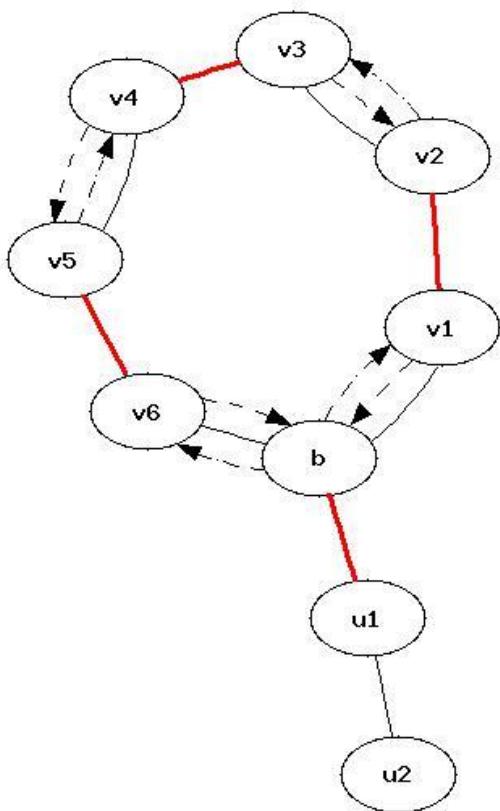
    // From the top of a climb to the root of marking all even
    vertices
    for (,;) {
        a = base[a];
        used[a] = true;
        if (match[a] == -1) break; // reached the root
        a = p[match[a]];
    }
    // Climb from the top of b, until we find a marked vertex
    for (,;) {
        b = base[b];
        if (used[b]) return b;
        b = p[match[b]];
    }
}

```

Now we need to identify the cycle itself - to walk from the vertices yet simply mark in a special array (call it current flower. After that we will need to reproduce the width of the bypass pseudo-tops - it is enough to put in place a preorder traversal of all vertices lying on a cycle of a flower. Thus we avoid explicit join adjacency lists.

However, there is still one problem: correct recovery at the end of bypass paths wide. For him, we maintain an array of ancestors. But after compression flowers only problem arises: traversal width continued right of all vertices of the cycle, including the odd, and the array of ancestors we intended to restore the ways in even the top. Furthermore, when compressed in the graph, there increases the chain through the flower, it will generally take place on this cycle in such a direction that the tree will be represented paths is downwards. However, all these problems are solved so gracefully maneuver: the compression cycle by placing all of its ancestors for even vertices (except base) to these "ancestors" indicated

an adjacent vertex in a loop. Vertices u and v if they are not also base pointers ancestors send each other. As a result, if the restoration of enhancing the way we come into flower in the odd cycle vertex path ancestor will be restored correctly, and lead to the base of the flower (from which he has will continue to recover normal).



So we are ready to implement compression flower:

```

int v, u; // edge (V, u), when considering which was discovered flower
int b = lca (v, u);
memset (blossom, 0, sizeof blossom);
mark_path (v, b, u);
mark_path (u, b, v);

```

where the function `mark_path()` passes on its way from the top to the base of the flower, shall enter into the array `blossom[]` them `true` and affix ancestors for even vertices. Parameter `children` - For son the summit (with this option we we close the cycle in the ancestors).

```
void mark_path (int v, int b, int children) {
    while (base[v] != B) {
        blossom[base[v]] = Blossom[base[match[v]]] = true;
        p[v] = Children;
        children = match[v];
        v = p[match[v]];
    }
}
```

Finally, we realize the basic function - `find_path (int root)` That will look for the path increases from the free vertex `root` and return to the last vertex of the path, or `-1` If increasing path not found.

Initially, we perform initialization:

```
int find_path (int root) {
    memset (used, 0, sizeof used);
    memset (p, -1, sizeof p);
    for (int i = 0; i < n; +
        + I) base[i] =
    I;
```

Next is a wide detour. Considering the next edge (v, to) We have a few options:

- Edge nonexistent. By this we mean that v and to belong to a single compressed pseudo-vertex ($\text{base}[v] == \text{base}[to]$). So in the current column of the edge surface is not. Also in this case, there is another case when an edge (v, to) already belongs to the current matchings; because we assume that the vertex v is an even vertex, then pass along this edge is in the tree paths climb to the top of ancestor v That is unacceptable.

```
if (base[v] == Base[to] || Match[v] == To) continue;
```

- Edge closes a cycle of odd length, ie detected flower. As mentioned above, an odd cycle length is detected under the conditions:

```
if (to == root || match[to], = -1 && P[match[to]], = -1)
```

In this case it is necessary to compress the flower. It has already been examined in detail this process, we present here its implementation:

```
int curbase = lca (v, to);
memset (blossom, 0, sizeof blossom);
mark_path (v, curbase, to);
mark_path (to, curbase, v);
for (int i = 0; i < n; + + I)
    if (blossom[base[i]]) {
        base[i] = Curbase;
        if (! Used[i]) {
            used[i] = true;
            q[qt + +] = I;
        }
    }
}
```

- Otherwise - it is "normal" edge, we proceed as in a normal search in width. The only subtlety - while making sure the top of this we have not yet visited, we must look not to the array `used`, and the array `p` - He filled for visited odd vertices. If we are in the top to still did not come, and it was unsaturated, we found a magnifying chain terminating at the top to . Return it.

```
if (p[to] == -1) {
    p[to] = v;
    if (match[to] == -1)
        return to;
```

```
to = match[to];  
used[to] = true;
```

```
q[qt + +] = To;
```

```
}
```

Thus, the complete implementation of the function `find_path()`:

```
int find_path (int root) {
    memset (used, 0, sizeof used);
    memset (p, -1, sizeof p);
    for (int i =0; i <n; +
        + I) base[i] =
    I;
    used[root] =
    true;int qh =0, Qt
    =0; q[qt + +] =
Root; while (qh
<qt) {
    int v = q[qh + +];
    for (size_t i =0; i <g[v]. Size(); + +
        I) { int to = g[v] [i];
        if (base[v] == Base[to] || Match[v] == To) continue;
        if (to == root || match[to] , = -1 && P[match[to]] !
        =
    -1) {
            int curbase = lca (v, to);
            memset (blossom, 0, sizeof blossom);
            mark_path (v, curbase, to);
            mark_path (to, curbase, v);
            for (int i =0; i <n; + + I)
                if (blossom[base[i]]) {
                    base[i] = Curbase;
                    if (! Used[i]) {
                        used[i] = true;
                        q[qt + +] = I;
                    }
                }
            }
        else if (p[to] == -1) {
            p[to] = v;
            if (match[to] == -1)
                return to;
            to = match[to];
            used[to] = true;
            q[qt + +] = To;
        }
    }
}
return -1;
}
```

Finally, we determine all the global arrays, and implementation of the main program for finding the greatest matchings:

```
const int MAXN = ...; / / maximum possible number of vertices in the input
graph

int n;
vector <int> G[MAXN];
int match[MAXN], P[MAXN], Base[MAXN], Q[MAXN];
bool used[MAXN], Blossom[MAXN];
...

int main() {
    Graph reading ... ...
    memset (match, -1, sizeof match);
    for (int i =0; i <n; + + I)
        if (match[i] == -1) {
```

```
int v = find_path (i);
while (v! = -1) {
    int pv = p[v], Ppv = match[pv];
    match[v] = Pv, match[pv] = V;
```

v = ppv;

1

}

Optimization: preliminary construction matchings

As in the case [Kuhn algorithm](#) Before performing Edmonds algorithm can any simple algorithm to construct a preliminary matching. For example, such a greedy algorithm:

```
for (int i =0; i <n; + + i)
    if (match[i] == -1)
        for (size_t j =0; j <g[i]. Size(); +
            + J) if (match[g[i][j]] == -1) {
            match[g[i][j]] = I; match[i] = G[i]
            [j];
            break;
        }
```

This optimization significantly (several times) will speed up the algorithm on random graphs.

The case of a bipartite graph

In bipartite graphs no cycles of odd length, and therefore the code that performs compression flowers, never executed. Mentally removing all parts of the code that handle compression of flowers, we obtain [Algorithm](#)

Kuhn substantially pure form. Thus, bipartite graphs on Edmonds algorithm degenerates into Kuhn algorithm and works for $O(nm)$.

Further optimization

In all the above operations with flowers thinly veiled operations with disjoint sets that can be performed much more efficiently (see [System of disjoint sets](#)). If we rewrite the algorithm

using that structure, then the asymptotic behavior of the algorithm decreases to $O(nm)$. Thus, for arbitrary graphs, we got the same asymptotic estimate that in the case of bipartite graphs (Kuhn algorithm), but significantly more complex algorithm.

Covering ways directed acyclic graph

G

Given a directed acyclic graph. Cover it requires the least number of ways, ie to find the smallest power over the tops of a set of disjoint simple paths, such that each vertex belongs to a path.

Reduction to bipartite graph

Let a graph with n vertices. We construct the corresponding bipartite graph in the standard way, ie: in each part of the graph will n vertices, and denote them respectively. Then, for each edge (i, j) original graph G draw the corresponding edge (a_i, b_j) .

Each edge corresponds to one edge
path , Then it is assigned a set of

Ribs G .

More easy to understand is, if we add a "backward" edges, ie form the graph
 $(a_{v_1}, b_{v_2}), (a_{v_2}, b_{v_3}), \dots, (a_{v_{k-1}}, b_{v_k})$ adding edges of the form . Then the path

will correspond to the path $(b_i, a_i), i = 1 \dots N$.

Conversely, consider any path $Q = (a_{v_1}, b_{v_2}, a_{v_2}, b_{v_3}, \dots, a_{v_k}, b_{v_k})$ in the first part and the second ending

lobe. Obviously, again will have the form . And it can be put
in line in the graph path . However, there is one subtlety: could coincide with

, So the path turned to cycle. However, by hypothesis graph acyclic, so it is generally
impossible (the only place where it is used acyclic graph); however, on the cyclic

graphs general procedure described here can not be generalized).

So, every simple path in the graph, starting at the first part and the second ending, you can

assign a simple path in the graph . And vice versa. But note that such a path in the

it matching in the graph . Therefore, any path from can associate

matching in a graph . And vice versa. Moreover, non-intersecting paths in match

disjoint matchings in .

The last step. Note that there are more paths in this set, the smaller these paths comprise fins. Namely, when it is p disjoint paths that cover all n the vertices of the graph, they together comprise

$r = n - p$ Ribs. Thus, to minimize the number of paths, we must **maximize the number of Ribs**

therein.

So, we have reduced the problem to finding the maximum matching in a bipartite graph H . After finding this matching (see [Kuhn algorithm](#)) We need to convert it into a set of paths in (this is done trivial algorithm ambiguity does not arise here). Some tops may remain unsaturated a matching, then the answer must be to add a zero-length path from each of these vertices.

The weighted case

The weighted case is not much different from the unweighted, just in the graph H appear on the edges of weight, and is required to find the minimum weight matching. Restoring response similar unweighted case, we obtain a covering graph fewest ways and at equality - the lowest cost.

Matrix Thatta

Tutte matrix - this elegant approach to the problem of **matchings** in an arbitrary (not necessarily bipartite) graph. However, in the simplest form the algorithm does not produce themselves ribs included in matching, but only the size of the maximum matching in a graph.

Below, we first consider the result obtained Tutt (Tutte) to verify the existence of a perfect matching (ie matching containing the edges, and all because of a saturating $n/2$ vertices). After that, we consider the result obtained later Lovasz (Lovasz), which already allows you to find the size of maximum matching, and not only limited to the case of perfect matchings. Then give the result of Rabin (Rabin) and Vazirani (Vazirani), who reported the reconstruction algorithm of the matchings (as a set of its constituent edges).

Definition

Let a graph G with n vertices (n - Even).

Then **Tutte matrix** (Tutte) is the following matrix $n \times n$:

$$\begin{pmatrix} 0 & x_{12} & x_{13} & \dots & x_{1(n-1)} & x_{1n} \\ -x_{12} & 0 & x_{23} & \dots & x_{2(n-1)} & x_{2n} \\ -x_{13} & -x_{23} & 0 & \dots & x_{3(n-1)} & x_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -x_{1(n-1)} & -x_{2(n-1)} & -x_{3(n-1)} & \dots & 0 & x_{(n-1)n} \\ -x_{1n} & -x_{2n} & -x_{3n} & \dots & -x_{(n-1)n} & 0 \end{pmatrix}$$

where x_{ij} is either independent variable corresponding to the edge between vertices i and j or identically zero, if the edges between vertices not.

Thus, in case of a complete graph with n vertices Tutte matrix contains $n(n-1)/2$ independent variables, if any edges in the graph are missing, then the corresponding elements of the matrix are converted to zeros. Generally, the number of variables in the matrix Tutte coincides with the number of edges.

Tutte matrix antisymmetric (skew).

Tutte's theorem

Consider the determinant $\det(A)$ Tutte matrix. It is, generally speaking, a polynomial in the variables x_{ij} .

Tutte's theorem reads as follows: in the graph there is a perfect matching if and only if when polynomial $\det(A)$ is not identically zero (ie has at least one term with a nonzero

ratio). Recall that the matching is called perfect if it saturates all the vertices, ie its power is $n/2$.

Canadian mathematician William Thomas Tutte (William Thomas Tutte) first pointed out the close relationship between matchings in graphs and determinants of matrices (1947). A simpler form of this connection later discovered Edmonds (Edmonds) in the case of bipartite graphs (1967). Randomized algorithms for finding maximum matching value themselves and edges of this matching were proposed later, respectively,

Lovasz (Lovasz) (1979), and Rabin (Rabin) and Vazirani (Vazirani) (in 1984).

Enforceability: The randomized algorithm

Directly apply Theorem Tutte even in the problem of testing the existence of a perfect matching is impractical. The reason for this is that when calculating the determinant character (ie in the form of polynomials over variables) interim results are polynomials containing x_{ij}

$O(n^2)$ variables. Therefore, the calculation of the determinant of Thatta symbolically require an inordinate amount of time.

Hungarian mathematician Laszlo Lovasz (Laszlo Lovasz) was the first to indicate the possibility of applying the **randomized** algorithm to simplify calculations.

The idea is very simple: replace all variables x_{ij} random numbers:

Then, if the polynomial $\det(A)$ was identically zero, after this replacement, and it will remain zero; if he was different from zero, then the replacement of such a random numerical probability that it will appeal to zero sufficiently small.

It is clear that such a test (random permutation values and calculation of the determinant $\det(A)$) If errs, it is only in one direction, may report the absence of a perfect matching, when in fact it exists.

We can repeat this test several times, substituting the values of variables as a new random number, and every restart we get more confident that the test gave the correct answer.

In practice, in most cases a single test to determine whether there is a perfect matching in a graph or not; Several such tests already given a very high probability.

To assess **Probability of Error** You can use Lemma Schwarz-Sippel (Schwartz-Zippel), which states that the probability of the vanishing of a nonzero polynomial of degree n by substituting the values of variables as random numbers, each of which can take s options values - this probability satisfies:

For example, when using standard random number function `C++` `rand()` we find that this probability at $n = 300$ is about one percent.

Asymptotics solution turns out to be $O(n^3)$ (Using, for example, [Gauss](#)) Multiplied the number of iterations of the test. It is worth noting that the asymptotic solution is far behind the decision [Edmonds compression algorithm flowers](#) However, in some cases, more preferably from ease of implementation.

Reestablish itself as a perfect matching set of ribs is more challenging. The easiest, albeit slow, recovery will be one of this matching one edge: iterate through the first edge response, choose it so that there remains a graph perfect matching, etc.

Proof of Tutte

To understand well the proof of this theorem, we first consider a simple result - Edmonds obtained for the case of bipartite graphs.

Edmonds theorem

Consider a bipartite graph in which for each lobe n vertices. Form the matrix $Bn \times n$ in which, by analogy with the matrix Thatta is a separate independent variable, if an edge is present in the graph, b_{ij} and identically zero otherwise.

(i, j)

This matrix is similar to the matrix of Thatta, but Edmonds matrix has half the difference, and each edge here has only one cell of the matrix.

We prove the following **Theorem**: Determinant $\det(B)$ is nonzero if and only if there is a bipartite graph, a perfect matching.

Proof. We expand the determinant according to its definition, as the sum over all permutations:

$$\det(B) = \sum_{\pi \in S_n} \text{sgn}(\pi) \cdot b_{1,\pi_1} \cdot b_{2,\pi_2} \cdot \dots \cdot b_{n,\pi_n}.$$

Note that since all non-zero elements of the matrix B - Various independent variables, in this sum all the nonzero terms are different, but because no reductions in the summation occurs. It remains to note that any non-zero term in this sum is disjoint set on the tops of the ribs, ie a perfect matching. Conversely, any perfect matching corresponds to a non-zero term in this sum. Coupled with the above, this proves the theorem.

Properties of antisymmetric matrices

To prove the theorem of Tutte must use several well-known facts of linear algebra on the properties of antisymmetric matrices.

Firstly (This fact is not useful to us, but it is interesting in itself) if an antisymmetric matrix has odd size, its determinant is always zero (Theorem Jacobi (Jacobi)). It's enough

notice that the antisymmetric matrix satisfies $A^T = -A$ And now we get the chain of equalities:

$$\det(A) = \det(A^T) = \det(-A) = (-1)^n \det(A),$$

which implies that for odd n determinant must be zero.

Secondly, it turns out, that in the case of antisymmetric matrices of even size of their determinant is always can be written as the square of a polynomial in the variables of the matrix elements (called the Pfaffian polynomial (pfaffian), and the result is due to Muir (Muir)):

$$\det(A) = \text{Pf}^2(A).$$

Thirdly, this Pfaffian is not arbitrary polynomial a sum of the form:

Thus, each term in the Pfaffian - a work of such
collectively constitute a partition of at n $n/2$ elements of their indices
their ratio, but his view we are not interested. $n/2$ pairs. Before each term has

Proof of Tutte

Using the second and third property from the previous paragraph, we see that the determinant of Thatta is the square of the $\det(A)$ sum of terms of the kind that each term

- The product of the matrix elements whose indices are not repeated and cover all the numbers from 1 to n . So again, as in the proof of Theorem Edmonds, every non-zero term in this sum corresponds to perfect matching in a graph, and vice versa.

Lovasz theorem: generalization to find the maximum size of matchings

Formulation

Rank Tutte matrix coincides with the double value **maximum matching** in this graph.

Application

To apply this theorem in practice, you can use the same technique of randomization, that in the above algorithm for the Tutte matrix, namely substitute random values of variables, and to find the rank of the resulting numerical matrix. Rank of the matrix, again, searched for using modified Gauss, see [here](#).

However, it should be noted that the algorithm given in the previous lemma Schwarz-Sippel inapplicable explicitly and intuitively it seems that the error probability becomes higher here. However, allegedly (see work Lovasz (Lovasz)), and that here the probability of error (ie, that the rank of the resulting matrix will be less than twice the size of the maximum matching) does not exceed (where s as above, denotes $\frac{n}{s}$ size of the set from which the random number).

Proof

The proof will follow from one **propertiesKnown** from linear algebra. Suppose we are given antisymmetric matrix A $n \times n$ And let set and - any two subsets S T of $\{1, \dots, n\}$ And the dimensions of these sets are identical. We denote A_{ST} matrix derived from A Only rows with indices of S and columns with indices T . Then we have:

$$\det(A_{SS}) \cdot \det(A_{TT}) = \det(A_{ST}) \cdot \det(A_{TS}).$$

We show how this property allows you to set **conformity** between the rank of the matrix Thatta A and the magnitude of the maximum matching.

On the one hand, consider the graph in a maximal matching, and denote the set of vertices saturable them through. Then the determinant is nonzero (Tutte's theorem) $\det(A_{UU})$

Investigator, the rank of Thatta at least not less than twice the size of the maximum matching.

Conversely, let the rank of equal r . This means that we found such a submatrix A_{TT} , where $|T| = r$. Whose determinant is nonzero. But according to the above property, this means that one of the matrices A_{SS} A_{TT} has nonzero determinant that Tutte's theorem means that subgraph induced by the set of vertices S T or, there is a perfect matching (and the value of $r/2$). Consequently, the rank of the matrix can not be greater than the maximum matchings that

completes the proof.

Rabin-Vazirani algorithm for finding the maximum matching

This algorithm is a further generalization of the previous two theorems, and allows, in contrast, the issue is not only the magnitude of maximum matching, but they themselves edges coming into it.

Statement of the theorem

Let the graph, there is a perfect matching. Then its Tutte matrix is nonsingular, ie $\det(A) \neq 0$. Generate on it as described above, the random numerical matrix B . Then, with a high probability, $(B^{-1})_{ji} \neq 0$ if and only if the edge (i, j) included in any perfect matching.

(Here, by B^{-1} denotes the matrix inverse. It is assumed that the matrix determinant is different from zero, so the inverse matrix exists.)

Application

This theorem can be used to restore the edges themselves maximal matching. First have to select the subgraph, which contains the desired maximal matching (this can be done in parallel with the rank of the search algorithm).

After that the problem reduces to finding a perfect matching on this numerical matrix obtained from the matrix Tutte. Here we apply Theorem Rabin-Vazirani - find the inverse matrix (which can be

make the modified algorithm for Gaussian $O(n^3)$, We find in it any nonzero element is removed from graph, and repeat the process. Asymptotics of such solutions is not the fastest - $O(n^4)$, But instead simplicity obtain solutions (compared, for example, Edmonds compression algorithm flowers).

Proof of Theorem

Recall the well-known formula for the elements of the inverse matrix B^{-1} :

$$(B^{-1})_{ji} = \frac{\text{adj}(B)_{i,j}}{\det(B)},$$

where indicated by the cofactor, ie this number is multiplied by the determinant of a matrix obtained from B Disposal i -Th row and j -Th column.

Hence we immediately obtain that the element $(B^{-1})_{ji}$ differs from zero if and only if the matrix B strike- i -Th row and-th column has a nonzero determinant that by applying Tutte's theorem, means a high probability that the graph without vertices i and j there is still a perfect matching.

Literature

- William Thomas Tutte. **The Factorization of Linear Graphs** [1946]
- Laszlo Lovasz. **On Determinants, Matchings and Random Algorithms** [1979]
- Laszlo Lovasz, M.D. Plummer. **Matching Theory** [1986]
- Michael Oser Rabin, Vijay V. Vazirani. **Maximum matchings in general graphs through randomization** [1989]
- Allen B. Tucker. **Computer Science Handbook** [2004]
- Rajeev Motwani, Prabhakar Raghavan. **Randomized Algorithms** [1995]
- A.C. Aitken. **Determinants and matrices** [1944]

Edge connectivity. Properties and being

Definition

Suppose we are given an undirected graph with G n peaks and m ribs.

Branch Connectivity column λ is the smallest number of edges that must be removed to cease to be a connected graph.

For example, for a disconnected graph edge connectivity is equal to zero. For a connected graph with a single bridge edge connectivity is equal to unity.

Say that the set of edges s shares vertices s and t , if you remove these edges from the graph vertices s and t are in different connected components.

Clearly, an edge connectivity of the graph is the minimum of the smallest number of edges that separate the two peaks s and t , taken among all possible pairs (s, t) .

Properties

Whitney ratio

Value for Whitney (Whitney) (1932 g) between Branch Connectivity λ , vertex connectivity κ and the smallest vertex degree of δ :

$$\kappa \leq \lambda \leq \delta.$$

We prove this assertion.

We first prove the first inequality. Consider a set of λ edges, making the graph disconnected. If we take from each of these edges on one end (either of the two) and remove it from the graph, it thereby using remote vertices (as $\leq \lambda$ one and the same vertex could meet twice) we will graph disconnected.

Thus, $\kappa \leq \lambda$.

To prove the second inequality. Consider a vertex of minimal degree, then we can remove all δ allied ribs and thereby separate it from the rest of the top graph. Consequently, $\lambda \leq \delta$.

Interestingly, the Whitney inequality **can not be improved**: i.e. for all triples of numbers satisfying this inequality, there exists at least one corresponding graph. See Problem "The construction of a graph with these quantities vertex and edge connectivity and the lowest vertex degree".

Ford-Fulkerson theorem

Ford-Fulkerson theorem (1956 g)

For any two vertices of the largest number of edge-disjoint circuits connecting them, equal to the minimum number of edges separating these peaks.

Finding the edge connectivity

A simple algorithm based on the search of the maximum flow

This method is based on the theorem of Ford-Fulkerson.

We have to sort through all pairs of vertices, and between each pair to find the greatest number of disjoint

along the edges of paths. This value can be found using the maximum flow algorithm: we do s source, t

- Drain and bandwidth of each rib put equal to 1. Thus, pseudo-

algorithm is as follows:

```
int ans = INF;
```

```

for (int s =0; s <n; + + s)
    for (int t = s +1; t <n; + + t) {
        int flow = ... the maximum value of the flux s in t ...
        ans = min (ans, flow);
    }
}

```

Asymptotic behavior of the algorithm using \ edmonds_karp {Edmonds-Karp algorithm for finding the maximum flow} is obtained, however it should be noted that hidden in the asymptotic constant is very small, since it is practically impossible to create a graph algorithm to find the maximum flow slowly worked simultaneously for all the source and drain.

Especially fast this algorithm will work on random graphs.

A special algorithm

Using streaming terminology, this problem - it is the task of finding **global minimal cut**.

To solve it developed special algorithms. At this site contains one of which - [algorithm Curtains-Wagner](#), running time $O(n^3)$ or $O(nm)$.

Literature

- Hassler Whitney. **Congruent Graphs and the Connectivity of Graphs** [1932]
- Frank Harari. **Graph Theory** [2003]

Edge connectivity. Properties and being

Definition

Suppose we are given an undirected graph G with n vertices and m edges.

Vertex connectivity graph is the smallest number of vertices, you want to delete, to graph ceased to be connected.

For example, for a disconnected graph vertex connectivity is zero. For a connected graph with a single point of articulation vertex connectivity equal to one. For a complete graph vertex connectivity considered equal $n - 1$ (As which pair of vertices we may choose, even the removal of all remaining vertices will not make them disconnected). For all graphs, but complete, vertex connectivity does not exceed $n - 2$. Because you can find a pair of vertices between which there is no edge, and remove all other $n - 2$ top.

Say that the set of vertices **shares** vertices s and, if the removal of these vertices of the graph vertices u and v are in different connected components.

It is clear that the vertex connectivity of the graph is equal to the minimum of the smallest number of vertices separating two vertices s and t , Taken among all possible pairs (s, t) .

Properties

Whitney ratio

Value for Whitney (Whitney) (1932) between Branch Connectivity λ , The vertex connectivity κ and the smallest vertex degree of δ :

$$\kappa \leq \lambda \leq \delta.$$

We prove this assertion.

We first prove the first inequality. Consider this set of λ edges, making the graph disconnected. If we take from each of these edges on one end (either of the two) and remove it from the graph, it thereby using remote vertices (as one and the same vertex could meet twice) we will graph disconnected.

Thus, $\kappa \leq \lambda$.

To prove the second inequality. Consider a vertex of minimal degree, then we can remove all δ allied ribs and thereby separate it from the rest of the top graph. Consequently, $\lambda \leq \delta$.

Interestingly, the Whitney inequality **can not be improved**: ie for all triples of numbers satisfying this inequality, there exists at least one corresponding graph. See Problem "The construction of a graph with these quantities vertex and edge connectivity and the lowest vertex degree".

Finding the vertex connectivity

s and t s and t .

i i_1

For this i_2 (i_1, i_2).

(u, v) (u_2, v_1) and (v_2, u_1) .

s t . s and t .

$$O(nm^2)$$

$$O(n^3m^2)$$

Graphing with the stated values of the vertex and edge connectivity and the lowest degree of the vertices

$$\kappa, \lambda, \delta$$

Whitney ratio

Value for Whitney (Whitney) (1932 г) between Branch Connectivity λ , vertex connectivity κ and the smallest vertex degree of δ :

$$\kappa \leq \lambda \leq \delta.$$

We prove this assertion.

We first prove the first inequality. Consider this set of λ edges, making the graph disconnected. If we take from each of these edges on one end (either of the two) and remove it from the graph, it thereby using remote vertices (as $\leq \lambda$ one and the same vertex could meet twice) we will graph disconnected.

Thus, $\kappa \leq \lambda$.

To prove the second inequality Consider a vertex of minimal degree, then we can remove all δ allied ribs and thereby separate it from the rest of the top graph. Consequently, $\lambda \leq \delta$.

Interestingly, the Whitney inequality **can not be improved**: i.e. for all triples of numbers satisfying this inequality, there exists at least one corresponding graph. This we prove constructive, showing how to construct the corresponding graphs.

Decision

Verify whether the data number κ, λ and δ Whitney ratio. If not, then there is no answer.

Otherwise, we construct the graph itself. It will consist of $2(\delta + 1)$ vertices, wherein the ~~first + 1~~ vertices form polnosvyaznom subgraph and second $\delta + 1$ vertices also form polnosvyaznom subgraph. Except that connect the two parts ribs so that these ribs of the first portion are adjacent the tops, and the other part - ~~the~~ λ vertices.

Easy to see that the resulting graph will have the necessary characteristics.

Inverse problem SSSP (inverse-SSSP - inverse problem of the shortest paths from one vertex)

There is a weighted undirected multigraph G of N vertices and M edges. An array P [1 .. N] and contains some initial vertex S. I want to change the edge weights so that all $P[I]$ was equal to the length of the shortest path from S to I, and the sum of all changes (sum of absolute changes of edge weights) would lower. If this is not possible, then the algorithm must give "No solution". Do negative edge weight is prohibited.

Description of the solution

We will solve this problem in linear time, just exhausting all the edges (ie, in one pass).

Let the current step we consider the edge from vertex A to vertex B length R. We assume that the vertex A has all conditions are met (ie, the distance from S to A really equals $P[A]$), and will check the fulfillment of the conditions for the vertex B. We have several options for the situation:

- 1. $P[A] + R < P[B]$

This means that we found a way shorter than it should be. Since $P[A]$ and $P[B]$, we can not change, then we must extend the current edge (regardless of the other edges), namely to perform:

$$R += P[B] - P[A] - R.$$

Furthermore, this means that we have found the way to the top of the B S, whose length is equal to the desired value $P[B]$, in the following steps so we do not have to be shortened any ribs (see option 2).

- 2. $P[A] + R >= P[B]$

This means that we found a way longer than required. Since these paths may be several we must choose among all such paths (edges) that which requires the least changes. Again, if we have extended some edge goes to the top of B (option 1), then we actually built it the shortest way to the top of B, and therefore shorten no edge will not have to. Thus, for each vertex we should keep the edge that are going to shorten, ie edge with the smallest weight changes.

Thus, just sorting through all the edges and having considered the situation for each edge ($O(1)$), we solve the inverse problem in linear time SSSP.

If at some point we are trying to change has altered edge, then obviously you can not do, and should issue a "No solution".

Furthermore, some peaks can not be achieved and the required estimate of the shortest path, then the answer will be too "No solution". In all other cases (except, of course, is clearly incorrect values in the array P, ie, $P[S] = 0$ or negative values), the answer will be.

Implementation

The program displays "No solution", if there is no solution, otherwise displays the first line of the minimum amount of changes weights of the edges, and in the next M lines - new edge weights.

```
const int INF = 1000 * 1000 *
1000; int n, m;
vector<int> p (n);

bool ok = true;
vector<int> cost (m), cost_ch (m), decrease (n, INF), decrease_id (n, -1); decrease [0] = 0;
for (int i = 0; i <m; + + i) {
    int a, b, c; // Current edge (a, b) the
    price of c cost [i] = c;
    for (int j = 0; j <= 1; + + j) {
        int diff = p [b] - p [a] -
c; if (diff > 0) {
            ok &= cost_ch [i] == 0 || cost_ch [i] ==
diff; cost_ch [i] = diff;
            decrease [b] = 0;
        }
        else
            if (-diff <= c && -diff < decrease [b])
                {decrease [b] = -diff;
                decrease_id [b] = i;
            }
    }
}
```

```
        swap (a, b);
    }

for (int i = 0; i <n; + + i) {
    ok &= decrease [i]! =
INF;      int r_id =
decrease_id [i]; if (r_id!
= -1) {
    ok &= cost_ch [r_id] == 0 || cost_ch [r_id] ===decrease
[i]; cost_ch [r_id] ==-decrease [i];
}
}

if (! ok)
    cout << "No solution";
else {
    long long sum = 0;
    for (int i = 0; i <m; + + i) sum + = abs (cost_ch
[i]); cout << sum << '\n';
    for (int i = 0; i <m; + + i)
        printf ("% d", cost [i] + cost_ch [i]);
}
```

The inverse problem of MST (inverse-MST - inverse problem of the minimum spanning tree) in O (N M2)

Given a weighted undirected graph G with N vertices and M edges (without loops and multiple edges). It is known that the graph is connected. Also listed some skeleton T of this graph (ie, selected N-1 edges that form a tree with N vertices). Want to change the weights of the edges so that the specified frame T is a minimal skeleton of the graph (more precisely, one of the minimum spanning tree), and make it so that the total change of all was the smallest scales.

Decision

We reduce task inverse-MST to the problem min-cost-flow, more precisely, to the problem of dual min-cost-flow

(In the sense of the duality of linear programming problems); then solve the latter problem.

So, let G be a graph with N vertices, M edges. Weight of each edge is denoted by C_i . We assume

without loss of generality that the edges with numbers from 1 to N-1 are edges T.

1. Necessary and sufficient condition MST

Let there be given a skeleton S (not necessarily minimal).

We first introduce the notation. Consider some edge j , does not belong to S. Obviously, the graph S there is a unique path connecting the ends of the ribs, ie the only path connecting the ends of the ribs and j consisting only of edges belonging to S. We denote $P[j]$ set of edges that form a path for the j -th rib.

To some skeleton S is minimal, necessary and sufficientTo:

$C_i \leq C_j$ for all $j \notin S$ and each $i \in P[j]$

It can be noted that as in our problem T backbone belong ribs 1 .. N-1, we can write this condition as follows:

$C_i \leq C_j$ for all $j = N..M$ and each $i \in P$

[j] (where i all lie in the range 1 .. N-1)

2. Paths Count

The notion of graph paths directly related to the previous theorem. Let there

be given a skeleton S (not necessarily minimal).

Then Count the ways H graph G is the following graph:

- It contains M vertices, each vertex in H uniquely corresponds to an edge in G.
- Bipartite graph H. The first are his share of node i, which correspond to the edges in G, belonging to the skeleton S. Accordingly, in the second part are the vertices j, which correspond to the edges not belonging to S.
- Edge is drawn from vertex i to vertex j if and only if i belongs to $P[j]$.
In other words, for each vertex j of the second part, it includes edges of all vertices of the first part corresponding to a plurality of ribs $P[j]$.

In case our problem we can slightly simplify the description of the graph ways:

an edge (i, j) exists in the H, if the $i \in P[j], j = N..M, i = 1..N-1$

3. Mathematical formulation of

Formally task inverse-MST written in this way:

```
find an array A [1 .. M] such that  
Ci + Ai <= Cj + Aj for all j = N..M and each i ∈ P[j] (i 1 .. N-1)  
and minimize the sum of | A1 | + | A2 | + ... + | Am |
```

here under the desired array A, we mean the values that you want to add to the weights of edges (ie, deciding task inverse-MST, we replace the weight of each edge i C_i the amount $C_i + A_i$).

Obviously, it makes no sense to increase the weight of the edges belonging to T, ie,

$A_i \leq 0, i = 1 \dots N-1$

it makes no sense to shorten the edges not belonging to T:

$A_i >= 0, i = N \dots M$

(Because otherwise we will only worsen the answer)

Then we can slightly **simplify** formulation of the problem by removing modules of the sum of:

```
find an array A [1 .. M] such that  
Ci + Ai <= Cj + Aj for all j = N.. M and each i ∈ P [j] (i 1 .. N-1)  
Ai <= 0, i = 1 .. N-  
1, Ai >= 0, i = N..  
M,  
and minimizing the sum of An + ... + Am - (A1 + ... + An-1)
```

Finally, just change "minimize" to "maximize", and in the amount of change all signs to the contrary:

```
find an array A [1 .. M] such that  
Ci + Ai <= Cj + Aj for all j = N.. M and each i ∈ P [j] (i 1 .. N-1)  
Ai <= 0, i = 1 .. N-  
1, Ai >= 0, i = N..  
M,  
maximize the amount of A1 + ... + An-1 - (An + ... + Am)
```

4. Reduction of the problem to the inverse-MST problem, the dual assignment problem

Formulation of the problem inverse-MST, which we have just given, is a formulation of the problem **linear programming** unknown $A_1 \dots A_m$.

We apply the classic method - consider **dual** her task.

By definition, to obtain a dual problem, you need to compare each inequality dual variable X_{ij} , interchanged objective function (which should be minimal)

and the coefficients on the right side, change the signs " \leq " on " \geq ", and vice versa, to minimize change maximization.

Thus, **dual to inverse-MST** task:

```
find Xij for each (i, j) ∈ H, such that:  
all Xij >= 0,  
for each i = 1 .. N-      for all j: (i, j) ∈ H <=  
1           Σ Xij 1,  
for each j = N.. M Σ Xij for all i: (i, j) ∈ H <= 1,  
and minimize Σ Xij (Cj - Ci) for all (i, j) ∈ H
```

The latter problem is **task assignment**: We need a graph H ways to select multiple edges so that none of the edges does not intersect with the other at the top, and the sum of edge weights (weight of the edge (i, j) is defined as $C_j - C_i$) should be minimal.

Thus, **The dual problem inverse-MST equivalent to that of appointments**. If we learn how to solve the dual problem of appointments, we will automatically decide task inverse-MST.

5. Decision dual assignment problem

Please take a moment to the special case of the assignment problem, which we got. First

this unbalanced assignment problem, because one lobe is $N-1$ vertices, and the other - M vertices, that is, in general, the number of vertices in the second part is more than an order of magnitude. To solve this dual problem appointments have specialized algorithm that decides it for $O(N^3)$, but here, this algorithm will not be considered. Secondly, such a task assignment can be called task assignment with weighted vertices: edge weights are set equal to 0, the weight of each vertex of the first part is set equal C_i , from the second part - equal to C_j , and the resulting solution of the problem will be the same.

We will solve the problem of the dual problem of appointments using **modified algorithm min-cost-flow**, who will find the minimum cost flow and simultaneously solution of the dual problem.

Flatten task assignment to the problem min-cost-flow very easily, but for the sake of completeness, we describe this process.

Add the graph source s and sink t . From s to each vertex of the first part will hold an edge with capacity = 1 and the value = 0. Every vertex of the second part will hold an edge to t with capacity = 1 and the value = 0. Capacity of all edges between the first and second installments as set equal 1.

Finally, the modified algorithm min-cost-flow (described below) served must **add an edge from s to t** a bandwidth = $N + 1$ and cost = 0.

6. Modified algorithm min-cost-flow solutions for the assignment problem

Here we consider **successive shortest-path algorithm with potentials**, which resembles the usual algorithm min-cost-flow, but also uses concept **potentials** That the end of the algorithm will be **decision dual problem**.

We introduce the notation. For each edge (i, j) denote U_{ij} its capacity through C_{ij} - its value through F_{ij} - flow along this edge.

We also introduce the concept of potential. Each vertex has its own potential P_{li} . The residual value of the ribs CPI_{ij} defined as:

$$CPI_{ij} = C_{ij} - P_{li} + P_{lj}$$

At any time of the algorithm **potentials such**That the following conditions are satisfied:

```
if  $F_{ij} = 0$ , then  $CPI_{ij} >= 0$ 
if  $F_{ij} = U_{ij}$ , then  $CPI_{ij} <= 0$ 
otherwise  $CPI_{ij} = 0$ 
```

The algorithm starts with zero flow, and we need to find some initial potential values that satisfy the above conditions. It is easily verified that this method is one of the possible solutions:

```
P_{lj} = 0 for  $j = N .. M$ 
P_{li} = min  $C_{ij}$ , where  $(i, j) \in H$ 
P_{ls} = min  $P_{li}$ , where  $i = 1 .. N-1$ 
P_{lt} = 0
```

Actually, the algorithm min-cost-flow consists of several iterations. **At each iteration**, we find the shortest path from s to t in the residual network, and as the weights of the edges using the residual value of CPI . Then we increase the flow along the path found by one and update the potentials as follows:

$$P_{li}^- = D_i$$

where D_i - found the shortest distance from s to i (again, in the residual network with edge weights CPI).

Sooner or later, we will find the path from s to t , which consists of a single edge (s, t) . Then after this iteration, we should **complete** the algorithm: indeed, if we do not stop the algorithm, it will already be on the way to a non-negative value, and add them to the answer is not necessary.

By the end of the algorithm we obtain a solution of the assignment (in the form of flow F_{ij}) and the solution of the dual assignment problem (in the array P_{li}).

(With Pli will have to conduct a slight modification: the total values Pli take PIs, because its values are meaningful only when PIs = 0)

6. Subtotal

So, we decided to dual task assignment, and therefore the task of inverse-MST. We estimate the **asymptotics** the resulting algorithm.

First we'll need to construct a graph paths. To do this just for each edge $j \notin T$ preorder traversal to find a way skeleton $T P[j]$. Then we construct a graph paths for $O(M) * O(N) = O(NM)$.

Then we find the initial values of the potentials for the $O(N) * O(M) = O(NM)$.

Then we iterate min-cost-flow, all iterations is no more than N (as N goes from the source of the ribs, each with a bandwidth = 1), at each iteration we are looking for ways to graph the shortest path from source to all other vertices. Since the vertices in the graph paths equals $M + 2$, and the number of edges - $O(NM)$, then

if you implement search shortest paths simplest version of Dijkstra's algorithm, each iteration of the min-cost-flow will perform in $O(M^2)$, and the whole algorithm min-cost-flow finishes in $O(NM^2)$.

The resulting asymptotic behavior of the algorithm is **$O(NM^2)$** .

Implementation

We sell all the above described algorithm. The only change - instead [Dijkstra's algorithm](#) applicable [Leviticus algorithm](#), which in many tests should run slightly faster.

```
const int INF = 1000 * 1000 * 1000;

struct rib {
    int v, c, id;
};

struct rib2 {
    int a, b, c;
};

int main () {

    int n, m;
    cin >> n >> m;
    vector<vector<rib>> g (n); // Graph format in adjacency lists
    vector<rib2> ribs (m); // All the ribs in a single list
    Graph reading ... ...

    int nn = m + 2, s = nn-2, t = nn-1;
    vector<vector<int>> f (nn, vector<int> (nn));
    vector<vector<int>> u (nn, vector<int> (nn));
    vector<vector<int>> c (nn, vector<int> (nn));
    for (int i = n-1; i < m; ++ i) {
        vector<int> q
        (n); int h = 0, t
        = 0;
        rib2 & cur = ribs [i]; q [t
        + +] = cur.a; vector<int>
        rib_id (n, -1); rib_id
        [cur.a] = -2;
        while (h < t) {
            int v = q [h + +];
            for (size_t j = 0; j < g[v].size();
                ++j) if (g[v][j].id == n-1)
                break;
            else if (rib_id [g [v] [j].v] == -1)
                {rib_id [g [v] [j].v] = g [v]
                [j].id; q [t + +] = g [v] [j].v;
                }
        }
        for (int v = cur.b, pv; v! = cur.a; v =
            pv) {int r = rib_id [v];
            pv = v! = ribs [r].a? ribs [r].a: ribs
            [r].b; u [r] [i] = n;
            c [r] [i] = ribs [i].c - ribs [r].c;
```

```

        c [i] [r] =-c [r] [i];
    }

u [s] [t] = n +1;
for (int i = 0; i <n-1; +
    + i) u [s] [i] =
    1;
for (int i = n-1; i <m; +
    + i) u [i] [t] =
    1;

vector <int> pi
(nn); pi [s] = INF;
for (int i = 0; i <n-1; + +
    i) {pi [i] = INF;
    for (int j = n-1; j <m; +
        + j) if (u [i]
        [j])
        pi [i] = min (pi [i], ribs [j]. c-ribs
        [i]. c); pi [s] = min (pi [s], pi [i]);
}

for (; ; ) {
    vector <int> id (nn);
    deque <int> q;
    q.push_back (s); vector
    <int> d (nn, INF); d [s]
    = 0;
    vector <int> p (nn, -
    1); while (! q.empty
    ()) {
        int v = q.front (); q.pop_front
        (); id [v] = 2;
        for (int i = 0; i <nn; + + i)
            if (f [v] [i] <u [v] [i]) {
                int new_d = d [v] + c [v] [i] - pi [v] +
pi [i];
                if (new_d <d [i]) {d
                    [i] = new_d; if
                    (id [i] == 0)
                    q.push_back (i);
                    else if (id [i] == 2)
                    q.push_front (i);
                    id [i] = 1;
                    p [i] = v;
                }
            }
        }
        for (int i = 0; i <nn; +
            + i) pi [i] -=
            d [i];
        for (int v = t; v! = s; v = p
            [v]) {int pv = p [v]; +
            + F [pv] [v], - f [v]
            [pv];
        }
        if (p [t] == s) break;
    }

    for (int i = 0; i <m; +
        + i) pi [i] -=
        pi [s];
    for (int i = 0; i <n-1; +
        + i) if (f [s]
        [i])

```

```
    ribs [i]. c += pi  
[i]; for (int i = n-1; i <m; ++ i)  
    if (f [i] [t])  
        ribs [i]. c += pi [i];
```

Output graph

}

Paint edges of the tree

This is a fairly frequent problem. Given tree G. request comes in two forms: the first kind - paint some edge, a second look - request amount colored edges between two vertices.

Here will be described fairly simple solution (using [tree segments](#)) That will respond to requests for O ($\log N$), with preprocessing (pretreatment of wood) for O (M).

Decision

First, we have to implement [LCA](#) That each request is the second type (i, j) is reduced to two requests (a, b), where a - ancestor b.

We now describe **preprocessing** actually our problem. Run dfs root of the tree, this dfs make some list visit vertices (each vertex is added to the list when it comes to search, and each time after dfs son returns from the current vertex) - incidentally, This list is used by the algorithm LCA. This list will be present each edge (in the sense that if i and j - the ends of the ribs, the list always find a place where i and j are contiguous to each other), and contain exactly two times: in the forward direction (from i to j, where i vertex closer to the top than the vertex j) and reverse (from j to i).

Construct two tree lengths (For amounts with a single modification) this list: T1 and T2. Tree T1 will consider each edge in the forward direction, and the tree T2 - on the contrary, only in reverse.

Let entered the regular **inquiry** species (i, j), where i - ancestor j, and requires to determine how much colored edges in the path between i and j. Find i and j in the list traversal depth (we definitely need a position where they first met), let it be some positions p and q (this can be done in O (1), if we calculate these positions in advance during preprocessing). Then **the answer is the sum of T1 [p .. q-1] - amount T2 [p .. q-1]**.

Why? Consider the segment [p; q] list traversal in depth. It comprises edges us desired path from i to j, but also contains a plurality of ribs which lie on the paths of the other i. However, between the right ribs us and the rest of the ribs there is one big difference: the desired edges are contained in the list only once, and in the forward direction, and all other edges will meet twice, both literally and reverse direction. Consequently, the difference T1 [p .. q-1] - T2 [p .. q-1] will give us the answer (minus one needed, because otherwise we will take another extra edge from vertex j somewhere up or down). Request the amount of tree segments runs in O ($\log N$).

The answer to **inquiry** type 1 (about painting a rib) is even easier - we just need to update the T1 and T2, namely to perform a single modification of the element that corresponds to our edge (the edge to find the list of crawl, again, can be O (1) if you perform this search in preprocessing). Individual modification in the tree segments runs in O ($\log N$).

Implementation

Here will be given full implementation solutions, including LCA:

```
const int INF = 1000 * 1000 * 1000;

typedef vector<vector<int>> graph;

vector<int> dfs_list;
vector<int> ribs_list;
vector<int> h;

void dfs (int v, const graph & g, const graph & rib_ids, int cur_h = 1)
{
    h[v] = cur_h;
    dfs_list.push_back (v);
    for (size_t i = 0; i < g[v].size
        (); ++ i) if (h[g[v][i]]
        == -1)
    {
        ribs_list.push_back (rib_ids[v]
        [i]); dfs (g[v][i], g, rib_ids,
        cur_h +1); ribs_list.push_back
        (rib_ids[v][i]);
        dfs_list.push_back (v);
    }
}
```

```
vector <int> lca_tree;
```

```

vector <int> first;

void lca_tree_build (int i, int l, int r)
{
    if (l == r)
        lca_tree [i] = dfs_list [l];
    else
    {
        int m = (l + r) >> 1;
        lca_tree_build (i + i, l, m);
        lca_tree_build (i + i +1, m +1,
                        r);
        int lt = lca_tree [i + i], rt = lca_tree [i +
                        i +1]; lca_tree [i] = h [lt] <h [rt]? lt: rt;
    }
}

void lca_prepare (int n)
{
    lca_tree.assign (dfs_list.size () * 8, -1);
    lca_tree_build (1, 0, (int) dfs_list.size () -
                    1);

    first.assign (n, -1);
    for (int i = 0; i <(int) dfs_list.size (); ++ i)
    {
        int v = dfs_list [i];
        if (first [v] == -1) first [v] = i;
    }
}

int lca_tree_query (int i, int tl, int tr, int l, int r)
{
    if (tl == l && tr == r)
        return lca_tree
            [i];
    int m = (tl + tr) >> 1;
    if (r <= m)
        return lca_tree_query (i + i, tl, m, l,
                               r); if (l > m)
        return lca_tree_query (i + i +1, m +1, tr, l,
                               r); int lt = lca_tree_query (i + i, tl, m, l, m);
    int rt = lca_tree_query (i + i +1, m +1, tr, m
                           +1, r); return h [lt] <h [rt]? lt: rt;
}

int lca (int a, int b)
{
    if (first [a] > first [b]) swap (a, b);
    return lca_tree_query (1, 0, (int) dfs_list.size () -1, first
                           [a], first [b]);
}

vector <int> first1,
first2; vector <char>
rib_used; vector <int>
tree1, tree2;

void query_prepare (int n)
{
    first1.resize (n-1 -1);
    first2.resize (n-1 -1);
    for (int i = 0; i <(int) ribs_list.size (); ++ i)
    {
        int j = ribs_list [i];
        if (first1 [j] == -1)
            first1 [j] = i;
        else
            first2 [j] = i;
    }
}

```

```
rib_used.resize (n-1);
tree1.resize (ribs_list.size () *
8); tree2.resize (ribs_list.size ()
* 8);
```

```

}

void sum_tree_update (vector <int> & tree, int i, int l, int r, int j, int
delta)
{
    tree [i] += delta; if (l <r)
    {
        int m = (l + r) >> 1;
        if (j <= m)
            sum_tree_update (tree, i + i, l, m, j, delta);
        else
            sum_tree_update (tree, i + i +1, m +1, r, j, delta);
    }
}

int sum_tree_query (const vector <int> & tree, int i, int tl, int tr, int
l, int r)
{
    if (l> r || tl> tr) return 0;
    if (tl == l && tr == r)
        return tree
    [i]; int m = (tl + tr)
>> 1; if (r <= m)
    return sum_tree_query (tree, i + i, tl, m, l,
r); if (l> m)
    return sum_tree_query (tree, i + i +1, m +1, tr, l,
r); return sum_tree_query (tree, i + i, tl, m, l, m)
    + Sum_tree_query (tree, i + i +1, m +1, tr, m +1, r);
}

int query (int v1, int v2)
{
    return sum_tree_query (tree1, 1, 0, (int) ribs_list.size () -1,
first [v1], first [v2] -1)
    - Sum_tree_query (tree2, 1, 0, (int) ribs_list.size ()
-1, first [v1], first [v2] -1);
}

int main ()
{
    // Read the
    graph int n;
    scanf ("% d", & n);
    graph g (n), rib_ids (n);
    for (int i = 0; i <n-1; +
+ i)
    {
        int v1, v2;
        scanf ("% d% d", & v1, &
v2); - V1, - v2; g [v1].
        push_back (v2); g [v2].
        push_back (v1); rib_ids
        [v1]. push_back (i);
        rib_ids [v2]. push_back
        (i);
    }

    h.assign (n, -1);
    dfs (0, g, rib_ids);
    lca_prepare (n);
    query_prepare (n);

    for (; ;)
    {
        if () {
            // inquiry about painting the edge labeled x;
            //      if start = true, then the edge is painted,
Paint otherwise removed
            rib_used [x] = start;
            sum_tree_update (tree1, 1, 0, (int) ribs_list.size
() -1, first1 [x], start? 1: 1);
    }
}

```

```
    sum_tree_update (tree2, 1, 0, (int) ribs_list.size  
() -1, first2 [x], start? 1: 1);
```

```
        }
    else {
        // request count of colored edges in the path between v1
        // and v2 int l = lca (v1, v2);
        int result = query (l, v1) + query (l, v2);
        // result - prompted
    }
}
```

Task 2-SAT

Task 2-SAT (2-satisfiability) - it is the task of distribution of values Boolean variables so that they satisfy all constraints.

2-SAT problem can be represented as a conjunctive normal form, where each expression in brackets is exactly two variable; this form is called a 2-CNF (2-conjunctive normal form). For example:

```
(A | | c) && (a | | ! D) && (b | | ! D) && (b | | ! E) && (c | | d)
```

Apps

Algorithm for solving a 2-SAT can be applied in all applications where there is a set of variables, each of which can take two possible values, and there is a connection between these values:

- **Location text labels on the map or chart.**

Means finding such an arrangement of marks for which no two disjoint.

It should be noted that in the general case where each label can occupy many different positions, we obtain the problem of general satisfiability, which is NP-complete. However, if we restrict ourselves to only two possible positions, the resulting problem is the task 2-SAT.

- **Location edges when drawing graph.**

Similarly to the previous paragraph, if we restrict ourselves to only two possible ways to hold an edge, then we will come to 2-SAT.

- **Scheduling Games.**

This refers to a system where each team must play with each other once, and want to distribute the game to the type of home-visiting, with some constraints.

- etc.

Algorithm

We first present the problem to another form - the so-called implicative form. Note that the expression of the form $a \mid b$ is equivalent! $A \Rightarrow b$ or! $B \Rightarrow a$. This can be interpreted as follows: if there is an expression $a \mid b$, and we need to make it appeal to true, if $a = \text{false}$, you must $b = \text{true}$, and vice versa, if $b = \text{false}$, it is necessary $a = \text{true}$.

We now construct the so-called **Count implications**: For each variable in the graph will be two vertices, which we denote by x_i and! X_i . Edges in the graph correspond to implicative relations.

For example, to form 2-CNF:

```
(A | | b) && (b | | ! C)
```

Count implications will contain the following edges (oriented)

```
! A =>  
b! B =>  
a! B =>!  
C c => b
```

Pay attention to the implications of such a property of the graph, that if there is an edge $a \Rightarrow b$, then there is an edge! $B \Rightarrow! A$.

Now, note that if for some variable x is performed, that of x is achievable! X , and from! X achievable x , then the problem has no solution. Indeed, whatever the value for the variable x , we would have chosen, we always arrive at a contradiction - that must be chosen and its inverse value. It turns out that this condition is not only sufficient but also necessary (proof of this fact is the algorithm described below). Reformulate this criterion in terms of graph theory. Recall that if one of other peaks attainable, and that the vertex is reachable from the first, these two peaks are in the same strongly connected component. Then we can formulate **criterion for the existence of solutions** follows:

To this problem 2-SAT has a solution! It is necessary and sufficient that for any variable x and vertex $x! X$ were in different components of the strong connectivity Count implications.

This criterion can be checked in time $O(N + M)$ using [search algorithm for strongly connected component](#).

Now build your own **algorithm** finding a solution to the problem 2-SAT under the assumption that a solution exists. Note that, despite the fact that a solution exists, some of the variables can be performed in that

x achievable! x, or (but not both) of! x achievable x. In this case, selecting one of the values of the variable x will lead to a contradiction, while another choice - will not. Learn to choose between two values that which does not lead to inconsistencies. Immediately, we note that by selecting a value, we have to run from him crawling in depth / width and mark all the values that follow from it, ie achievable

implications in a graph. Accordingly, for vertices are labeled no choice between x and! X do not need to their value has already been chosen and recorded. Described hereafter rule applies only to untagged still tops.

Allegedly following. Let comp [v] denotes the number of strongly connected components, which belongs to the vertex v, and the rooms are arranged in order of topological sort of strongly connected components in the graph components (ie earlier in order topological sorting correspond to large numbers if there is a path from v to w, then the comp [v] <= comp [w]). Then, if the comp [x] <comp [! X], then select the value! X, otherwise, ie

, if comp [x]> comp [! x], then we choose x.

We prove, that with this choice of values, we do not arrive at a contradiction. Let For definiteness, selected vertex x (the case where the peak! x, proved symmetrically).

First, we prove that x is not achievable! X. Indeed, since the number of strongly connected components comp [x] more room components comp [! X], it means that the connected component containing x, is located to the left of the connected components containing! X, and from the first can not be achieved last .

Secondly, we prove that no vertex y, accessible from x, is not "bad", ie incorrectly, that of y achievable! y. We prove this by contradiction. Let x be achieved from y, and y is achievable from! Y. Since x reachable from y, then, by implication graph property from! Y is achievable! X. But by assumption of y achievable! Y. Then we see that from achievable! x! x, which contradicts the assumption that we wanted to prove.

Thus we have constructed an algorithm that finds the desired values of the variables under the assumption that for any variable x and vertex x! X are in different strongly connected components. Above showed the correctness of this algorithm. Consequently, we have simultaneously proved the existence of the above criteria solutions.

Now we can collect **the whole algorithm** together:

- Construct a graph of implications.
- Find in this graph strongly connected components during the O (N + M), let comp [v] - is the number of strongly connected components, which belongs to the vertex v.
- Verify that for each variable x, and the vertex x! X lie in different components, ie comp [x] ≠ comp [! x]. If this condition is not satisfied, then return "solution does not exist."
- If comp [x]> comp [! X], then select the variable x is true, otherwise - false.

Implementation

Below is the implementation of the solution of the problem for 2-SAT is the implications of this graph g and its inverse graph gt (ie, in which the direction of each edge reversed).

The program displays the numbers of the selected vertices or phrase "NO SOLUTION", if there is no solution.

```
int n;
vector <vector <int>> g, gt;
vector <bool> used; vector
<int> order, comp;

void dfs1 (int v) {used
    [v] = true;
    for (size_t i = 0; i <g [v]. size ();
         + + i) {int to = g [v] [i];
        if (! used [to])
            dfs1 (to);
    }
    order.push_back (v);
}

void dfs2 (int v, int cl)
{comp [v] = cl;
 for (size_t i = 0; i <gt [v]. size ();
      + + i) {int to = gt [v] [i];
        if (comp [to] == -1)
            dfs2 (to, cl);
    }
}

int main () {
    Reading ... n, the graph g, construction graph gt ...
    used.assign (n, false);
}
```

```
for (int i = 0; i <n; +
    + i) if (!
        used [i])
            dfs1 (i);

comp.assign (n, -1);
for (int i = 0, j = 0; i <n; +
    + i) {int v = order
[ni-1]; if (comp [v]
== -1)
    dfs2 (v, j + +);
}

for (int i = 0; i <n; + + i)
    if (comp [i] == comp [i ^ 1])
        {puts ("NO
SOLUTION"); return 0;
}
for (int i = 0; i <n; + + i) {
    int ans = comp [i]> comp [i ^ 1]? i: i ^
1; printf ("% d", ans);
}

}
```

Heavy-light decomposition

Heavy-light decomposition - it is quite a common technique which can effectively solve many problems reduce to **requests on the tree**.

The simplest **example** problems of this kind - is the next task. Given a tree, each node is assigned a number.

Receives requests form, where a and b - the numbers of vertices of the tree, and requires determine the maximum number of nodes in the path between a and b .

Description of the algorithm

So, suppose we are given a tree G with n vertices, suspended for a root.

The essence of this decomposition is to **split tree in several ways** so that for every vertex v turned out that if we rise from v to the top, then the path will change the maximum $\log n$ tract. In addition, all paths should not overlap with each other in the ribs.

It is clear that if we learn to look for any such decomposition of wood, it will minimize any kind of inquiry "to learn something on the way out a in b ". Multiple queries like "find something on the interval $[l; r]$ On the way."

Construction of heavy-light decomposition

Calculate for each vertex of the size of its subtree $s(v)$ (ie the number of vertices in the vertex subtree v including the very top).

Next, consider all the edges leading to the sons of a vertex. We call an edge (v, c) **heavy**, if it leads to the top, such that:

All other edges are said **Light**. Obviously, one of the vertices v could come down heavy at most one edge (because otherwise at the top v it would be the size of two sons that given $s(v)/2$ the summit v gives the size of $2 \cdot s(v)/2 + 1 > s(v)$ ie a contradiction).

Now we construct itself **decomposition** tree into disjoint paths. Consider all the vertices of which does not go down any hard edges, and we will go from each of them up until you get to the root of the tree or not to pass a light edge. As a result, we get a few ways - show that this is the required way of heavy-light decomposition.

Proof of the correctness of the algorithm

First, note that the resulting algorithm will be the way **disjoint**. In fact, if any two paths have a common edge, it would mean that some of the peaks comes down two heavy ribs, which can not be.

Secondly, we show that down from the root to any vertex, we **we change the path not more ways**. In fact, pass down the easy edge reduces the size of the current subtree more than $\log n$ doubled

Thus, we could not pass the lighter edges. However, to move from one track to another, we can only through the lung edge (because each way but ending at the root, contains a slight edge at the end, and get right in the middle of the way, we can not).

Therefore, the path from the root to any vertex we can not change over routes, that QED.

Use in solving problems

When solving problems is sometimes more convenient to consider heavy-light as a set **vertex-disjoint** ways (Instead of edges disjoint). It's enough of each

ways to eliminate the last edge, if it is a light edge - then no properties are not violated, but

Now each vertex belongs to exactly one path.

Below we look at some common tasks that can be solved with the help of heavy-light decomposition.

We should also pay attention to the task **sum of the numbers on the way** As an example of a problem that can be solved by a simple technique.

The maximum number of the path between two vertices

Given a tree, each node is assigned a number. Receives requests form, where a and b - the numbers of vertices of the tree, and you want to know the maximum number of nodes on the path between a and b .

Construct advance heavy-light decomposition. Above each construct was obtained by [segment tree for maximum](#), that allow you to search the top with a maximum number attributed to this segment of the path for. Although the number of paths in the heavy-light decomposition can reach $n - 1$, The total size of all the paths is the value $O(n)$. And therefore the total amount of trees will also be linear segments.

Now, in order to respond to the request

(a, b) find the lowest common ancestor these

vertices (e.g., [binary method of lifting](#)). Now the problem is reduced to two requests: (a, l) and (b, l) . Each of which we can answer this way: we find ways in which lies lower vertex, make a request to this path, go to the top-end of the path, again define the way in which we were and make a request to him, and so on, until we reach the path comprising l .

Gently should be the case when, for example, a and b were in the same way - then you get the maximum to this path should not do on the suffix, and the domestic subsegments.

Thus, in the process of answering one subquery we pass along paths, each of them making a request of the maximum on the suffix or prefix / subsegments (request for prefix / subsegments could be a one time only).

So we got the solution for $O(\log^2 n)$ one request.

If additionally predposchitat highs in each path at all suffixes, you get a solution for $O(n \log n)$. Because Request no maximum on the suffix happens only once, when we get to the top l .

Sum of the numbers in the path between two vertices

Given a tree, each node is assigned a number. Receives requests form, where a and b - the numbers of vertices of the tree, and you want to know the sum of the path between vertices a and b . The variant of this problem when there are further questions, change the number attributed to a given vertex.

Although this problem can be solved with the help of heavy-light decomposition, building on each path segment tree for the sum (or simply predposchitav partial sums, if the problem no change requests), this problem can be solved **simpler techniques**.

If no modification requests, the amount to learn the path between two vertices can be in parallel with the search for the two vertices in the LCA [binary algorithm lifting](#) - It is enough during preprocessing for

LCA counting not only 2^k 's Ancestors each vertex, but also the sum of the path to that ancestor.

There is also a fundamentally different approach to this problem - consider Euler traversal of the tree, and build a segment tree above him. This algorithm is discussed in [article similar to the decision problem](#). (And if requests no modification - it is enough to do predposchetom partial sums, without a tree segments.) Both of these methods provide

relatively simple solutions with the asymptotic $O(\log n)$ one request.

Repainting edges of the path between two vertices

Given a tree, each edge originally painted white. Receives requests form (a, b, c) Where $-b$ vertex numbers - numbers, which means that all edges in the path of a need to be repainted in color c . Required after repaintings all say how much in the end it turned edges of each color.

Solution - just make [tree to painting on a segment](#) over the set of paths heavy-light decomposition.

Each request repainting the way (a, b) turn into two subqueries and where (a, l) The lowest common ancestor of vertices a and b (Found, for example, [binary algorithm lifting](#)), And each of those subrequests - in $O(\log n)$ queries to the trees above the track segments.

Altogether it is a solution with the asymptotic $O(\log^2 n)$ one request.

Problem in online judges

List of tasks that can be solved using heavy-light decomposition:

- TIMUS # 1553 "Caves and Tunnels" [Difficulty: Medium]
- IPSC 2009 L "Let there be rainbows!" [Difficulty: Medium]
- SPOJ # 2798 "Query on a tree again!" [Difficulty: Medium]
- Codeforces Beta Round # 88 E "The tree or tree" [Difficulty: high]

Length of association on the line segments in O (N log N)

Given N line segments on the line, ie each segment is defined by a pair of coordinates (X1, X2).

Consider the union of these segments and find its length.

The algorithm was proposed by Klee (Klee) in 1977. The algorithm works in O (N log N). It has been proved that this algorithm is faster (asymptotically).

Description

Put all the coordinates of the ends of segments in the array X and sort it by value coordinates.

Additional condition for sorting - with equal coordinates must first go left ends. In addition, for each element of the array will be stored, it refers to the left or to the right end of the segment. Now go through the entire array, with counter C overlapping segments. If C is non-zero, then the result is added to the difference $X_i - X_{i-1}$. If the current element refers to the left end, then increment the counter C, otherwise reduce it.

Implementation

```
unsigned segments_union_measure (const vector <pair <int,int>> & a)
{
    unsigned n = a.size ();
    vector <pair <int,bool>> x (n *
2); for (unsigned i = 0; i <n; i +
++)
{
    x [i * 2] = make_pair (a [i]. first,
false); x [i * 2 +1] = make_pair (a [i].
second, true);
}
sort (x.begin (), x.end ());

unsigned result = 0;
unsigned c = 0;
for (unsigned i = 0; i <n * 2; i + +)
{
    if (c && i)
        result += unsigned (x [i]. first - x [i-1].
first); if (x [i]. second)
        ++ C;
    else
        - C;
}
return result;
}
```

The sign area of a triangle and the predicate "Clockwise"

Definition

Suppose we are given three points p_1, p_2, p_3 . Find the value of **sign area** Triangle $p_1 p_2 p_3$ ie the area of this triangle, taken with the plus or minus sign depending on the type of pivot points formed by p_1, p_2, p_3 : Counterclockwise or her accordingly.

It is clear that if we learn how to calculate such a landmark ("targeted") area, and we can find the usual area of any triangle, and also be able to check, clockwise or counterclockwise directed any triple of points.

Calculation

Use the notion of **Bias** (Pseudoscalar) product of vectors. It just is twice the area of the triangle sign:

$$a \wedge b = |a||b| \sin \angle(a, b) = 2S,$$

where the angle $\angle(a, b)$ taken oriented, i.e. is the angle of rotation between the vectors counterclockwise. (Module skew product of two vectors is equal to the modulus **Vector** their work.)

Skew product is calculated as the value of the determinant of the coordinates of points:

$$2S = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}.$$

Expanding the determinant, we can obtain the following formula:

$$2S = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2).$$

You can group the third term with the first two, got rid of one multiplication:

$$2S = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

This formula is convenient to record and store in a matrix form as the following determinant:

$$2S = \begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix}.$$

Implementation

Function that computes the area of the triangle twice landmark:

```
int triangle_area_2 (int x1, int y1, int x2, int y2, int x3, int y3) {
    return (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
}
```

Function that returns the usual area of the triangle:

```
double triangle_area (int x1, int y1, int x2, int y2, int x3, int y3) {
    return abs (triangle_area_2 (x1, y1, x2, y2, x3, y3)) / 2.0;
}
```

Function that checks whether the specified forms triple points clockwise rotation:

```
bool clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
    return triangle_area_2 (x1, y1, x2, y2, x3, y3) < 0;
}
```

Function that checks whether the specified forms triple points counterclockwise rotation:

```
bool counter_clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
    return triangle_area_2 (x1, y1, x2, y2, x3, y3) > 0;
}
```

Checking on the intersection of two segments

Given two segments AB and CD (They may degenerate to a point). Need to check whether or not they intersect. If you want to find itself further point (s) of intersection, then see [relevant article](#).

First way: the oriented area of the triangle

We use [Oriented area of the triangle and the predicate 'Clockwise'](#). Indeed, to

segment an s overlap, it is necessary and sufficient to point and on different sides

direct AB and CD similarly, the point A B and C D - On different sides of the line AB . You can check this calculating the area of the corresponding oriented triangles and comparing their signs.

The only thing you should pay attention to - the boundary cases where some points fall on the line itself. Thus there is only a special case where the above checks give nothing - where both segments are **on the same line**. This case should be considered separately. It suffices to verify that the projections of these two segments on the axis X and Y intersect (often this test is called "test of the bounding box").

In general, this method - simpler than the one that will be given below (which produces the intersection of two lines), and has fewer special cases, but its main drawback - the fact that he does not find itself crossing point.

Implementation:

```
struct pt {
    int x, y;
};

inline int area (pt a, pt b, pt c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

inline bool intersect_1 (int a, int b, int c, int d) {
    if (a > b) swap (a, b);
    if (c > d) swap (c, d);
    return max(a, c) <= Min(b,
        d);
}

bool intersect (pt a, pt b, pt c, pt d) {
    return intersect_1 (a.x, b.x, c.x, d.x)
        && intersect_1 (a.y, b.y, c.y, d.y)
        && area(a, b, c) * Area(a, b, d) <= 0
        && area(c, d, a) * Area(c, d, b) <= 0;
}
```

In order to optimize the test of bounding box is moved to the beginning, to calculate areas - as it is more "light" verification.

Needless to say, this code applies to the case of real coordinates, just all comparisons with zero should be made by epsilon (and avoid multiplying two real-values $\text{area}()$, multiplying instead of signs).

The second method is the intersection of two lines

Instead of crossing segments perform [intersection of two lines](#). As a result, if the lines are not parallel, obtain some point, you have to check it belongs to both segments; it suffices to verify that this point belongs to both segments in the projection on the axis X and axle Y .

If parallel lines were, then, if they do not match, then the segments do not exactly overlap. If direct match, then the segments lie on one line, and to check their intersection is sufficient to verify that intersect their projection on the axis X and Y .

There still remains a special case, when one or both of the segment **degenerate** in point: in this case, talking about direct incorrectly, and this method may not be applicable (this case should be handled separately).

Implementation (Excluding the case of degenerate intervals):

```
struct pt {
    int x, y;
};

const double EPS = 1E-9;

inline int det (int a, int b, int c, int d) {
    return a * d - b * c;
}

inline bool between (int a, int b, double c) {
    return min(a, b) <= c + EPS && c <= max(a, b) + EPS;
}

inline bool intersect_1 (int a, int b, int c, int d) {
    if (a> b) swap (a, b);
    if (c> d) swap (c, d);
    return max(a, c) <= Min(b,
        d);
}

bool intersect (pt a, pt b, pt c, pt d) {
    int A1 = ay-by, B1 = bx-ax, C1 = -A1 * ax - B1 * ay; int
    A2 = cy-dy, B2 = dx-cx, C2 = -A2 * cx - B2 * cy; int zn =
    det (A1, B1, A2, B2);
    if (zn!= 0) {
        double x = - det (C1, B1, C2, B2) * 1. / Zn;
        double y = - det (A1, C1, A2, C2) * 1. / Zn;
        return between (a.x, b.x, x) && Between (a.y, b.y, y)
            && Between (c.x, d.x, x) && Between (c.y, d.y, y);
    }
    else
        return det (A1, C1, A2, C2) == 0 && Det (B1, C1, B2, C2) == 0
            && intersect_1 (a.x, b.x, c.x, d.x)
            && intersect_1 (a.y, b.y, c.y, d.y);
}
```

Here first factor is calculated Zn - The denominator in Cramer. If $Zn = 0$, The coefficients A and B directly proportional, and the lines are parallel or coincide. In this case, you should check whether or not they are the same, which is necessary to check that the coefficients are directly proportional to the C same factor, which is sufficient to compute the determinant of the following two if they are both zero, then the lines are the same:

If the $zn \neq 0$, The lines intersect, and Cramer's rule we find the point of intersection (x, y) and check it belongs to both segments.

It should be noted that if the starting point coordinates have been real-valued, it is necessary to normalize the direct (i.e., bring them to a state that the sum of the squares of the coefficients a and equal to one), otherwise error when compared to the parallel lines and a match may be too large.

Finding the equation for a straight line segment

Task - given the coordinates of the end of the segment to build a line through it.

We believe that the segment is non-degenerate, ie has a length greater than zero (otherwise, of course, passes through infinitely many different lines).

Two-dimensional case

Let a segment PQ we know the coordinates of its ends P_x, P_y, Q_x, Q_y .

Required to construct **equation of a line in the plane** Passing through this segment, i.e. find the coefficients A, B, C straight-line equation:

$$Ax + By + C = 0.$$

Note that the desired triples (A, B, C) Passing through a given period, **infinitely many**:

You can multiply all three coefficients on an arbitrary non-zero number and get the same line. Therefore, our task - to find one of these triples.

It is easy to see (by substituting these expressions and the coordinates of points P and Q in the equation of the line), which fits the following set of coefficients:

$$\begin{aligned} A &= P_y - Q_y, \\ B &= Q_x - P_x, \\ C &= -AP_x - BP_y. \end{aligned}$$

Integer case

An important advantage of this method of constructing direct is that if all the coordinates are integers, then the obtained coefficients will also **integer**. In some cases this allows geometric operations, all without resorting to real numbers.

However, there is a slight drawback: for the same line can be obtained different coefficients triples. To avoid this, but not away from the integer coefficients, we can apply the following method,

often called **rationing**. Find **greatest common divisor** Numbers $|A| |B| |C|$, Divide by him all three factors, and make the sign of normalization: if $A < 0$ or $A = 0, B < 0$ Then multiply all three coefficients on -1 . As a result, we arrive at the conclusion that for the same lines will receive the same triple coefficients that make it easy to check the lines on equality.

Real-valued case

When working with real numbers should always be aware of the errors.

Coefficients are obtained and we have about the original coordinates, the coefficient - already about a square of them. It may already be sufficiently large numbers, and, for example, **intersection of the lines** they will become more more, which can lead to large errors even when rounding the original coordinates, the order of 10^3 .

Therefore, when working with real numbers, it is desirable to produce a so-called **normalization** line: namely, do the coefficients such that $A^2 + B^2 = 1$. For this we need to calculate the number Z :

$$Z = \sqrt{A^2 + B^2},$$

and share all three coefficients A, B, C it.

Thus, the order of the coefficients A and B will not depend on the order of input coordinates, and the ratio would be the same C order as the input coordinates. In practice, this leads to a considerable improvement precision.

Finally, we mention the **comparison** direct - because after a normalization to the same line can be obtained only two triples format: up to multiplication by -1 . Accordingly, if we perform additional normalization with sign (if $A < -\varepsilon$ or $|A| < \varepsilon, B < -\varepsilon$) Then multiply on -1). The resulting coefficients are unique.

Three-dimensional and multi-dimensional case

Already in the three-dimensional case **There is no simple equation** Describing the line (it can be defined as the intersection of two planes, ie, a system of two equations, but it is inconvenient way).

Consequently, three-dimensional and multidimensional cases, we must use **parametric direct way to specify**, ie a dot \mathbf{p} and the vector \mathbf{v} :

$$\mathbf{p} + t\mathbf{v}, \quad t \in \mathbb{R}.$$

Ie straight - it all points that can be obtained from the point \mathbf{p} adding vector \mathbf{v} with an arbitrary coefficient.

Construction line in parametric form by coordinates endpoints - trivial, we simply take one end of the segment as a point \mathbf{p} And the vector from the first to the second end - for the vector \mathbf{v} .

Intersection point

Suppose we are given two lines defined by its coefficients A_1, B_1, C_1 and A_2, B_2, C_2 . Need to find their point of intersection, or to find out what the lines are parallel.

Decision

If two lines are not parallel, then they intersect. To find the point of intersection of the two is enough to make direct system of equations and solve it:

$$\begin{cases} A_1x + B_1y + C_1 = 0, \\ A_2x + B_2y + C_2 = 0. \end{cases}$$

Using the formula Cramer, immediately find the solution of the system, which is the required **intersection point**:

$$x = -\frac{\begin{vmatrix} C_1 & B_1 \\ C_2 & B_2 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{C_1B_2 - C_2B_1}{A_1B_2 - A_2B_1},$$
$$y = -\frac{\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{A_1C_2 - A_2C_1}{A_1B_2 - A_2B_1}.$$

If the denominator is zero, that

$$\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} = A_1B_2 - A_2B_1 = 0$$

the system has no solutions (direct **parallel** and do not coincide), or has infinitely many (direct **coincide**). If you need to distinguish between these two cases, it is necessary to verify that the coefficients are directly proportional with the same coefficient of proportionality, the coefficients A and B . Which is enough to calculate the determinant of the two if they are both zero, then the lines are the same:

$$\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}$$

Implementation

```
struct pt {
    double x, y;
};

struct line {
    double a, b, c;
};

const double EPS = 1e-9;

double det (double a, double b, double c, double d) {
    return a * d - b * c;
}

bool intersect (line m, line n, pt & res) {
    double zn = det (m.a, m.b, n.a, n.b);
    if (abs (zn) < EPS)
        return false;
    res.x = - det (m.c, m.b, n.c, n.b) / zn;
    res.y = - det (m.a, m.c, n.a, n.c) / zn;
}
```

```
    res.y = - det (m.a, m.c, n.a, n.c) / Zn;
    return true;
}

bool parallel (line m, line n) {
    return abs (det (m.a, m.b, n.a, n.b)) <EPS;
}

bool equivalent (line m, line n) {
    return abs (det (m.a, m.b, n.a, n.b)) <EPS
        && abs (det (m.a, m.c, n.a, n.c)) <EPS
        && abs (det (m.b, m.c, n.b, n.c)) <EPS;
}
```

Intersection of two segments

Given two segments, and (they may degenerate to a point). Required to find their intersection: it may be empty (if the segments do not intersect) can be a single point, and can be a whole segment (if the segments overlap).

Algorithm

Segments will work with both direct: construct two segments of direct equation, checking whether parallel lines. If the lines are not parallel, it's simple: find their intersection point and check that it belongs to both segments (it is sufficient to verify that the point belongs to each segment in the projection on the axis X and axle Y separately). As a result, in this case the answer is either "empty" or single point found.

A more complicated case - if the lines were parallel (same applies here the case where one or both segments degenerated into points).

In this case, we must verify that both segments lie on a line (or, in the case when they both degenerate to the point - that this point is the same). If it is not, then the answer is - "empty". If so, then the answer

- An intersection of two segments lying on the same line that is implemented is quite simple - it is necessary to take the maximum of the left end and right ends of the minimum.

At the beginning of the algorithm write the so-called "check for bounding box" - first, it is necessary for the case when the two segments are on a line, and secondly, it is a lightweight verification, allows the algorithm to work faster on average random tests.

Implementation

We present here a complete implementation, including all auxiliary functions for working with points and lines.

The main feature here is the `intersect` Which intersects two segments transmitted to it, and if they intersect at least one point, then returns `true`. And arguments `left` and `right` returns the start and end response interval (in particular, when the answer - the only point that is returned beginning and end are the same).

```
const double EPS = 1E-9;

struct pt {
    double x, y;

    bool operator < (const pt & p) const {
        return x < p.x-EPS || abs(x-p.x) < EPS && y < p.y - EPS;
    }
};

struct line {
    double a, b, c;

    line() {}
    line (pt p, pt q) {
        a = p.y - q.y;
        b = q.x - p.x;
        c = - a * p.x - b * p.y;
        norm();
    }

    void norm() {
        double z = sqrt (a * a + b *
        b); if (abs(z) > EPS)
            a /= z, b /= z, c /= z;
    }

    double dist (pt p) const {
        return a * p.x + b * p.y + c;
    }
};
```

```

# Define det (a, b, c, d) (a * d - b * c)

inline bool betw (double l, double r, double x) {
    return min(l, r) <= x + EPS && x <= max(l, r) + EPS;
}

inline bool intersect_1d (double a, double b, double c, double d) {
    if (a > b) swap (a, b);
    if (c > d) swap (c, d);
    return max (a, c) <= Min (b, d) + EPS;
}

bool intersect (pt a, pt b, pt c, pt d, pt & left, pt & right) {
    if (! intersect_1d (a.x, b.x, c.x, d.x) || ! intersect_1d (a.y, b.y,
c.y, d.y))
        return false;
    line m (a, b);
    line n (c, d);
    double zn = det (m.a, m.b, n.a, n.b);
    if (abs (zn) <EPS) {
        if (abs (m.dist (c)) > EPS || abs (n.dist (a)) >
            EPS) return false;
        if (b < a) swap (a, b);
        if (d < c) swap (c, d);
        left = max (a, c);
        right = min (b,
d);return true;
    }
    else {
        left.x = right.x = - det (m.c, m.b, n.c, n.b) / zn;
        left.y = right.y = - det (m.a, m.c, n.a, n.c) / zn;
        return betw (a.x, b.x, left.x)
            && betw (a.y, b.y, left.y)
            && betw (c.x, d.x, left.x)
            && betw (c.y, d.y, left.y);
    }
}

```

Finding a simple square polygon

Let a simple polygon (ie without self-intersections, but not necessarily convex) sets the coordinates of its vertices in order to bypass or counterclockwise. Need to find its area.

Method 1

This is easily done if iterate over all edges and fold area of the trapezoid bounded each edge.
Area must be taken so familiar with how it will turn out (thanks to sign all the "extra" area will be reduced).

Ie formula is:

$$S = (X_2 - X_1) * (Y_1 + Y_2) / 2$$

Code:

```
double sq (const vector <point> & fig)
{
    double res = 0;
    for (unsigned i = 0; i < fig.size (); i++)
    {
        point
            p1 = i? fig [i-1]: fig.back (),
            p2 = fig [i];
        res += (p1.x - p2.x) * (p1.y + p2.y);
    }
    return fabs (res) / 2;
}
```

Method 2

You can do otherwise. We choose an arbitrary point O, iterate over all the edges, adding to the response-oriented area of the triangle formed by the edge and the point O (see [Oriented area of a triangle](#)). Again, thanks mark, any excess space will be reduced, and will only answer.

This method is good because it is easier to generalize to more complex cases (eg, when some of the parties - not straight and circular arc).

Pick's theorem. Finding the square lattice polygon

Polygon without self-called lattice if all its vertices are the points with integer coordinates (Cartesian).

Pick's theorem

Formula

Suppose we are given some lattice polygon with nonzero area.

Denote its area through; number of points with integer coordinates lying strictly inside the polygon - through I ; number of points with integer coordinates lying on the sides of the polygon - through B .

Then the relation called **by Lance**:

$$S = I + \frac{B}{2} - 1.$$

In particular, if the values of I and B for a polygon, it is possible to calculate the area per

$O(1)$ Even without knowing the coordinates of its vertices.

This ratio is discovered and proved the Austrian mathematician Georg Alexander Pick (Georg Alexander Pick) in 1899

Proof

The proof is done in several stages: from the simplest shapes to arbitrary polygons:

- The unit square. In fact, for him $S = 1, I = 0, B = 4$ And the formula is true.
- Arbitrary nondegenerate rectangle with sides parallel to the coordinate axes. For proof of the formula denoted by a and the lengths of the sides of the rectangle. Then we find:
$$S = ab, I = (a-1)(b-1), B = 2(a+b)$$
. Direct substitution we see that the formula True peak.
- Angled triangle with legs parallel to the axes. To prove this, we note that any such triangle can be obtained by cutting off some of its diagonal rectangle. Denoting C the number of integer points lying on the diagonal, we can show that the formula holds for Pick such a triangle, regardless of the C .
- Arbitrary triangle. Note that any such triangle can be converted into a rectangle by attaching to the sides of right triangles with legs parallel to the axes coordinate (this will need no more than 3 such triangles). From here you can get a correct formula for Pick any triangle.
- Arbitrary polygon. To prove this triangulate it, ie divided into triangles with vertices at integer points. For one triangle formula Peak we have already proved. Next, we can prove that in addition to any arbitrary polygon triangle formula Peak retains its validity. Hence, by induction, it follows that it is true for any polygon.

Generalization to higher dimensions

Unfortunately, this is so simple and beautiful formula Peak bad generalized to higher dimensions.

Demonstrated that Reeve (Reeve), proposing in 1957 to consider the tetrahedron (now called **tetrahedron Riva**) with the following vertices:

$$\begin{aligned}A &= (0, 0, 0), \\B &= (1, 0, 0), \\C &= (0, 1, 0), \\D &= (1, 1, k),\end{aligned}$$

where - any natural number. Then this tetrahedron $ABCD$ in any k does not contain any points inside with integer coordinates, and on its border - lie just four points A, B, C, D and no other. Thus, the volume and surface area of the tetrahedron may be different, while the number of points within and on the boundary - unchanged; therefore, the formula does not allow generalizations Pick even three-dimensional case.

However, a similar generalization to higher-dimensional space is still available, - it **Ehrhart polynomials** (Ehrhart Polynomial), but they are very complex and depend not only on the number of points inside and on the border of the shape.

Segments covering problem points

Given N line segments on the line. Required to cover their fewest points, ie find the smallest set of points such that each segment belongs to at least one point.

Also consider more complicated version of this problem - when further specified "prohibited" a set of segments, ie no point of the answer should not belong to any segment prohibited.

It should also be noted that this problem can be viewed as a problem in scheduling theory - is required to cover a given set of events-segments fewest points.

The following will describe a greedy algorithm that solves both problems for $O(N \log N)$.

The first task

Note first that we can consider only those solutions in which each point is located at the right end of a segment. Indeed, it is easy to see that any solution if it does not satisfy this property can lead to it, moving it to the right point as much as possible.

Let us now try to build a solution that satisfies the specified property. Take the right end-point segments, sort them, and move them from left to right. If the current point is already covered by the right end of the segment, we skip it.

Suppose now that the current point is the right end of the current segment, which has not been covered before.

Then we need to add in response to the current location, and mark all the cuts, which belongs to this point, as covered. Indeed, if we missed the current point and would not add it to the answer, as it is the right end of the current segment, we will not be able to cover the current segment.

However, the naive implementation of this method will work for $O(N^2)$. We describe the **effective implementation** this method.

Take all the points, the endpoints (both left and right) and sort them. Moreover, for each point stick together with her room segment, as well as the way in which it is the end of it (left or right). Moreover, sort the points so that if there are several points in a single coordinate, the first would go

the left end, and only then - right. Head of the stack, which will be stored in rooms segments considered in the moment; initially the stack is empty. We move through the points in sorted order. If the current point - the left end, then just add the number of its segments in the stack. If it is the right end, then check that it has not covered this segment (you can just make an array of Boolean variables). If it has already been covered, then do nothing and go to the next point (looking ahead, we argue that in this case the current stack segment already there). If he had not been covered, we add the current point in response, and now we want to note for all current segments that they are covered. Since the stack is just numbers stored uncovered more segments, we will get out of the stack on one segment and note that it is already covered, until the stack is not completely empty. At the end of the algorithm, all segments will be covered, and, moreover, the fewest points (again, it is important to demand that the first coordinates are equal, the left ends, and only then to the right).

Thus, the entire algorithm runs in $O(N)$, not counting the sort of points, and the total complexity of the algorithm is exactly equal $O(N \log N)$.

The second task

Here are emerging prohibited segments, so, first, the decision may not exist at all, and secondly, we can not say that the answer can be only of the right ends of the segments. However, the algorithm described above may be appropriately modified.

Again, take all the points-ends of the segments (as target segments and the forbidden), sort them, keeping together with each point and the type of cut, the end of which it is. Again, sort the segments so that coordinates with equal left ends before going right, and if all types are equal, the left ends banned should go before the end of the target left and right ends of banned - after the target (the prohibited segments accounted for as long as possible with equal coordinates). Head counter prohibited segments that will be equal to the number of forbidden lines, covering the current point. Head queue (queue), which will store the number of the current target segments. Let's take the points

b sorted order. If the current point - the left end of the target segment, then just add the number of its segments b turn. If the current point - the right end of the target segment, if the counter is prohibited segments is zero, then we proceed similarly to the previous problem - put an end to the current location, and pushes all the segments of the line, noting that they are covered. If the counter is prohibited intervals greater than zero, at the current point, we can not shoot, but because we need to find the last point, free from prohibited segments; for this it is necessary to maintain an appropriate pointer `last_free`, which will be updated when entering prohibited segments. Then we shoot `last_free-EPS` (because it can not directly shoot - this

point disallowed segment), and push the segments of the queue until the point `last_free-EPS` belongs to them. Namely, if the current point - the left end of the forbidden interval, we increment the counter, and if before

counter is zero, then assign last_free current coordinate. If the current point - the right end prohibited segment, simply decrease the counter.

Centers of gravity of polygons and polyhedra

Center of gravity (Or the center of mass) of a body is the point having the property if hung body beyond this point, it will maintain its position.

The following are the two-and three-dimensional problems associated with finding different mass centers - mainly in terms of computational geometry.

As discussed below solutions are two main **fact**. First - that the center of mass of the system of material points equal to the average of their coordinates taken with coefficients proportional to their masses. The second fact - that if we know the centers of mass of two disjoint pieces, the center of mass of their

Unions will lie on the segment joining the two centers, and he will divide it in the same regard as the mass of the second figure refers to the weight of the first.

Two-dimensional case: polygons

In fact, speaking of the center of mass of the two-dimensional figures can bear in mind one of the following three **problems**:

- Center of mass of the system of points - ie all mass is concentrated only in the vertices of the polygon.
- Center of mass frame - ie weight polygon centered on its perimeter.
- Center of mass of the solid figures - ie weight polygon distributed throughout its area.

Each of these tasks is an independent decision, and will be considered separately below.

Center of mass of the system of points

This is the simplest of the three tasks, and its solution - known physical formula center of mass system of material points:

$$\vec{r}_c = \frac{\sum_i \vec{r}_i m_i}{\sum_i m_i},$$

where m_i - Mass points \vec{r}_i , their radius vectors (defining their position relative to the origin), and \vec{r}_c - The desired radius vector of the center of mass.

In particular, if all the points have the same mass, the coordinates of the center of mass has **average** coordinates of the points. For **Triangle** This point is called **centroid** and coincides with the point of intersection of the medians:

$$\vec{r}_c = \frac{\vec{r}_1 + \vec{r}_2 + \vec{r}_3}{3}.$$

For **evidence** these formulas is enough to recall that the equilibrium is reached at a point \vec{r}_c in which the sum of all the forces is zero. In this case, it becomes the condition that the sum of the radius vectors of all points around \vec{r}_c , Multiplying by the mass corresponding point is equal to zero:

$$\sum_i (\vec{r}_i - \vec{r}_c) m_i = \vec{0},$$

and expressing here \vec{r}_c , We obtain the required formula.

Center of mass frame

We assume for simplicity that the frame is homogeneous, ie its density is everywhere the same.

But then each side of the polygon can be replaced by a single point - the middle of this segment (as the center of mass of a homogeneous segment is the midpoint of the interval), with a mass equal to the length of this segment.

Now we have the problem of a system of material points, and applying it to the solution of the preceding paragraph, we find:

$$\vec{r}_c = \frac{\sum_i \vec{r}'_i l_i}{P},$$

where \vec{r}'_i - Point-to-mid i -Second side of a polygon, l_i - Length i -Second hand P - Perimeter, ie the sum of the lengths of the sides.

For **Triangle** can show the following statement: this point is **point intersection of the bisectors** triangle formed by midpoints of the sides of the original triangle. (To show this, we need to use the above formula, and then notice that the bisectors of a triangle divide the parties in the same proportions as the centers of mass of these parties).

Center of mass of a solid figure

We believe that mass is distributed uniformly on the figure, ie density at each point of the figure is the same number.

Triangle case

It is argued that for a triangle answer is still the same **centroidle** point formed by the arithmetic mean of the vertex coordinates:

$$\vec{r}_c = \frac{\vec{r}_1 + \vec{r}_2 + \vec{r}_3}{3}.$$

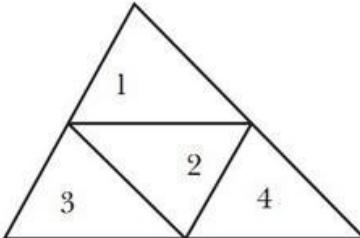
Case triangle proof

We present here an elementary proof that does not use the theory of integrals.

Like the first purely geometric proof led Archimedes, but it was very complicated, with a large number of geometric constructions. The proof given here is taken from article Apostol, Mnatsakanian "Finding Centroids the Easy Way".

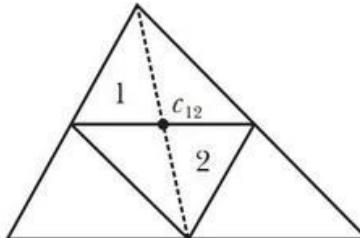
The proof boils down to showing that the center of mass of the triangle lies on one of the medians; repeating this process twice more, we thus show that the center of mass lies at the intersection of the medians, which is the centroid.

We divide this triangle T four by connecting the midpoints of sides, as shown:

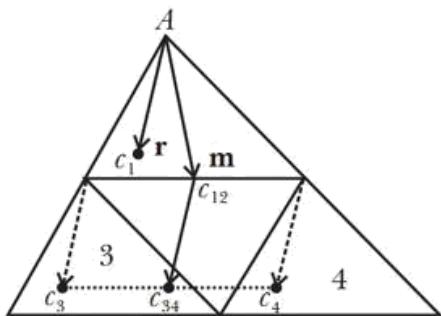


Four of the resulting triangle similar triangle T coefficient $1/2$.

Triangles number 1 and number 2 together form a parallelogram whose center of mass c_{12} lies at the intersection of its diagonals (as this figure is symmetric with respect to both diagonals, and hence, its center of mass shall lie on each of the two diagonals). Point c_{12} is in the middle common side triangles number 1 and number 2, and lies on the median of the triangle T :



Suppose now that the vector - vector from the vertex to the center of mass of the triangle number 1, and let the vector \vec{m} - Vector drawn from A to point c_{12} (Which, recall, is the midpoint of the side on which it is):



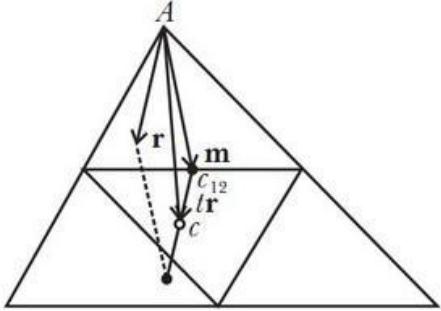
Our goal - to show that the vector \vec{r} and \vec{m} collinear.

We denote c_3 and c_4 points that are the centers of mass of the triangles number 3 and number 4. Then, obviously, the center of mass of the two triangles together to point c_{34} , Which is the midpoint of c_3c_4 .

Furthermore, the vector from point c_{12} to point c_{34} coincides with vector \vec{r} .

Seeking the center of mass c Triangle T lies in the middle of the segment connecting points c_{12} and c_{34}

(Since we broke triangle T into two parts of equal areas: № 1 - № 2 and № 3 - № 4):



Thus, a vector from the vertex A to the centroid c equal. On the other hand, because the triangle similar to the triangle number 1 T coefficient $1/2$. Then the same vector is $2\vec{r}$. Hence we obtain the equation:

$$\vec{m} + \vec{r}/2 = 2\vec{r},$$

from which we find:

Thus, we have proved that the vector and \vec{r} \vec{m} collinear, which means that the required centroid lies on the median, coming from the top.

Moreover, in passing we have proved that the centroid divides each median in relation to $2 : 1$ Starting from the top.

The case of a polygon

We now turn to the general case - ie a case **mnoougoulnika**. For him, such arguments are no longer applicable, so we reduce the problem to a triangle: namely, we divide the polygon into triangles (ie triangulate it is), we find the center of mass of each triangle, and then find the center of mass of the resulting mass centers of the triangles.

The final formula is obtained as follows:

$$\vec{r}_c = \frac{\sum_i \vec{r}_i^o S_i}{S},$$

where \vec{r}_i^o - centroid i Th triangle in the polygon triangulation, S_i - Area i Th triangle triangulation S - Total area of the polygon.

Triangulation of a convex polygon - a trivial task: for this example, we can take the triangles (r_1, r_{i-1}, r_i) Wherein $i = 3 \dots n$.

Polygon case: an alternative method

On the other hand, the application of this formula is not very convenient for **nonconvex polygons**, produce them as triangulation - in itself a difficult task. But for such polygons can come up with a simpler approach. Namely, the analogy with the way you can search for an arbitrary polygon area: Select an arbitrary point Z And then summed iconic areas of the triangles formed by this point and polygon:

$S = |\sum_{i=1}^n S_{z,p_i,p_{i+1}}|$. A similar method can be applied to find the center of mass: only now we will summarize the centers of mass of the triangles (z, p_i, p_{i+1}) taken with coefficients proportional to their areas, ie final formula for the center of mass is:

where - an arbitrary point p_i - Points of a polygon, $\vec{r}_{z,p_i,p_{i+1}}$ - The centroid of the triangle (z, p_i, p_{i+1}) $S_{z,p_i,p_{i+1}}$ - The sign area of the triangle - the sign area of the entire polygon (ie is $S = \sum_{i=1}^n S_{z,p_i,p_{i+1}}$).

Three-dimensional case: polyhedra

Similarly, two-dimensional case, in 3D, you can talk directly about the four possible formulations of the problem:

- Center of mass of the system of points - the vertices of the polyhedron.
- Center of mass frame - edges of a polyhedron.
- Center of mass of the surface - that is, mass is distributed over the surface area of the polyhedron.
- The center of mass of solid polyhedron - ie mass is distributed throughout the polyhedron.

Center of mass of the system of points

As in the two-dimensional case, we can apply the physical formula and get the same result:

$$\vec{r}_c = \frac{\sum_i \vec{r}_i m_i}{\sum_i m_i},$$

which in case of equal mass is converted to the arithmetic average of coordinates of all points.

Center of mass frame of the polyhedron

Similarly, two-dimensional case, we simply replace each edge of the polyhedron material point, located in the middle of the edge, and with a mass equal to the length of the edge. After receiving the task of material points, we can easily find the solution as a weighted sum of the coordinates of these points.

Center of mass of the surface of the polyhedron

Each face of the polyhedron surface - dimensional figure, the center of mass which we are able to search. Finding these centers of mass and replacing every facet of its center of mass, we obtain a problem with the material points, which we can easily solve.

The center of mass of solid polyhedron

The case of the tetrahedron

As in the two-dimensional case, we solve first the simplest task - a tetrahedron.

$$\vec{r}_c = \frac{\vec{r}_1 + \vec{r}_2 + \vec{r}_3 + \vec{r}_4}{4}$$

The intersection of a circle and a straight line

Decision

with geometric hand (And, due to this we get a more accurate solution in terms of numerical stability).

We assume without loss of generality that the center of the circle is at the origin (if it is not, then move it back, correcting suitable constant C in the equation line). Ie have a circle with center (0,0) radius r and straight to the equation $Ax + By + C = 0$.

First, we find **closest to the center point** direct - point to some coordinates **(X0, y0)**. First, this point must be at a distance from the origin:

$$\begin{aligned} & | \quad C \quad | \\ \hline \hline & \sqrt{A^2 + B^2} \end{aligned}$$

Secondly, since the vector (A, B) perpendicular to the line, the coordinates of this point should be proportional to the coordinates of the vector. Given that the distance from the origin to the desired point, we know we just need to normalize the vector (A, B) to this length, and we get:

$$\begin{aligned} x_0 &= \frac{A \cdot C}{A^2 + B^2} \\ y_0 &= \frac{B \cdot C}{A^2 + B^2} \end{aligned}$$

(There are not obvious signs only 'minus', but these formulas are easily verified by substituting in the equation of the line - should get zero)

Knowing the closest to the center point of the circle, we can determine how many points will be the answer, and even give an answer, if these points 0 or 1.

Indeed, if the distance (x_0, y_0) from the origin (its already expressed by us - see above)

larger than the radius, then **answer - zero points**. If this distance is equal to the radius, then **the answer is one point** - (X_0, y_0) . But in the remaining case, there will be two points, and their coordinates we will find it.

So we know that the point (x_0, y_0) lies inside the circle. Desired point (ax, ay) and (bx, by) , in addition to what should belong to the line, must lie on the same distance d from the point (x_0, y_0) , this distance is easy to find:

$$d = \sqrt{r^2 - \frac{C^2}{A^2 + B^2}}$$

Note that the vector $(-B, A)$ is collinear with the line, but because the required points (ax, ay) and (bx, by) can be obtained by adding to the point (x_0, y_0) vector $(-B, A)$, normalized to the length of d (we get a desired point), and subtracting the same vector

(Obtain the second desired point).

The final decision is:

$$\begin{aligned} mult &= \sqrt{\frac{d^2}{A^2 + B^2}} \\ ax &= x_0 + B \cdot mult \end{aligned}$$

```
ay = y0 - A mult
bx = x0 - B mult
by = y0 + A mult
```

If we solved this problem in a purely algebraic, then most likely would have received a decision in another form, which gives more accuracy. Therefore, the "geometric" approach described herein, in addition to clarity, and even more precise.

Implementation

As indicated at the outset, it is assumed that a circle is the origin. Therefore, the input parameters - the radius of the circle and the coefficients A, B, C of the line equation.

```
double r, a, b, c; // Input

double x0 = -a * c / (a * a + b * b), y0 = -b * c / (a * a + b * b); if (c * c > r * r * (a * a + b * b) + EPS)
    puts ("no points");
else if (abs (c * c - r * r * (a * a + b * b)) < EPS) {puts ("1 point");
    cout << x0 << ' ' << y0 << '\n';
}
else {
    double d = r * r - c * c / (a * a + b * b); double mult = sqrt (d / (a * a + b * b)); double ax, ay, bx,
    by;
    ax = x0 + b * mult;
    bx = x0 - b * mult;
    ay = y0 - a * mult;
    by = y0 + a * mult;
    puts ("2 points");
    cout << ax << ' ' << ay << '\n' << bx << ' ' << by << '\n';
}
```

The intersection of two circles

Given two circles, each determined by the coordinates of its center and radius. You want to find all their points of intersection (either one or two, or none of the point or the circle are the same).

Decision

Reduce our problem to one [The intersection of a circle and a straight line](#).

We assume without loss of generality that the center of the first circle - at the origin (if it is not, then transfer the center at the origin, and the derivation of the response will be back to add the coordinates of the center). Then we have a system of two equations:

$$\begin{aligned}x^2 + y^2 &= r_1^2 \\(x - x_1)^2 + (y - y_1)^2 &= r_2^2\end{aligned}$$

Subtract the second equation of the first to get rid of the squares of the variables:

$$\begin{aligned}x^2 + y^2 &= r_1^2 \\x(-2x_1) + y(-2y_1) + (x^2 + y^2 + r_1^2 - r_2^2) &= 0\end{aligned}$$

Thus, we have reduced the problem of the intersection of two circles to the problem of the intersection of the first circle with the following line:

$$\begin{aligned}Ax + By + C &= 0, \\A &= -2x_1, \\B &= -2y_1, \\C &= x^2 + y^2 + r_1^2 - r_2^2.\end{aligned}$$

A solution of this problem is described in [corresponding article](#).

Only **degenerate case** which should be considered separately - when the centers of the circles coincide. Indeed, in this case, instead of a linear equation, we obtain an equation of the form $0 = C$, where C - a number, and this case will be handled correctly. Therefore, this case should be considered separately: if the radii of the circles are the same, then the answer is - infinity, otherwise - no points of intersection.

Construction of the convex hull bypass Graham

Given N points of the plane. Build their convex hull, ie the smallest convex polygon containing all the points.

We discuss the method **Graham** (Graham) (proposed in 1972) with improvements Andrew (Andrew) (1979).

With it you can build a convex hull during $O(N \log N)$ using only comparison operations of addition and multiplication. The algorithm is asymptotically optimal (it is proved that there is no algorithm with the best asymptotic behavior), although in some problems it is unacceptable (in the case of parallel processing or online-processing).

Description

Algorithm. Find the leftmost and rightmost points A and B (if several such points, then we take the very bottom of the left and right of the top-most). It is clear that A, B and certainly fall within a convex hull. Next, draw through them straight AB, dividing the set of all points on the upper and lower subsets S1 and S2 (the point lying on the line can be attributed to any set - they still will not be included in the shell). Points A and

B assigned to both sets. Now we construct the upper shell for S1 and for S2 - lower shell, and combine them to get an answer. To get, say, the top shell, you need to sort all the points on the abscissa, and then go through all the points, considering at each step except for the point of the previous two terms included in the shell. If the current triple of points does not form a right turn (which is easily verified by [Oriented area](#)) is the nearest neighbor to remove from the shell. Eventually remain only point included in the convex hull.

Thus, the algorithm is to sort all points along the abscissa and two (worst case) rounds all points, ie, the required asymptotic behavior of $O(N \log N)$ is reached.

Implementation

```
struct pt {
    double x, y;
};

bool cmp (pt a, pt b) {
    return a.x < b.x || a.x == b.x && a.y < b.y;
}

bool cw (pt a, pt b, pt c) {
    return ax * (by-cy) + bx * (cy-ay) + cx * (ay-by) < 0;
}

bool ccw (pt a, pt b, pt c) {
    return ax * (by-cy) + bx * (cy-ay) + cx * (ay-by) > 0;
}

void convex_hull (vector <pt> & a)
{
    if (a.size () == 1)
        return;
    sort (a.begin (), a.end (), &
    cmp); pt p1 = a [0], p2 = a.back
    (); vector <pt> up, down;
    up.push_back (p1);
    down.push_back (p1);
    for (size_t i = 1; i <a.size (); + + i) {
        if (i == a.size () -1 || cw (p1, a [i], p2)) {
            while (up.size ()>= 2 && ! cw (up [up.size () -2],
            up [up.
            size () -1], a [i]))
                up.pop_back
            (); up.push_back (a
            [i]);
        }
        if (i == a.size () -1 || ccw (p1, a [i], p2)) {
            while (down.size ()>= 2 && ! ccw (down [down.size
            () -2], down [down.size () -1], a [i]))
                down.pop_back
            (); down.push_back (a
            [i]);
        }
    }
}
```

```
 }  
a.clear ();
```

```
for (size_t i = 0; i < up.size ();  
     ++ i) a.push_back (up  
                      [i]);  
for (size_t i = down.size () -2; i > 0;  
     - i) a.push_back (down [i]);  
}
```

Finding area combining triangles.

Vertical decomposition method

Given N triangles. Required to find the area of their union.

Decision

Here we consider the method **vertical decomposition**. That problems in the geometry is often very important.

So, we have N triangles that can arbitrarily interfere with one another. Get rid of these intersections using vertical decomposition: find all points of intersection of all segments

(Forming a triangle), and sort the results according to their point of abscissa. Suppose we have some array B. We move through this array. At i-th step we consider the elements B [i] and B [i + 1]. We have a vertical strip between the lines $X = B [i]$ and $X = B [i + 1]$, where, according to the very construction of the array B, inside this band segments can not intersect with each other.

Therefore, within this band triangles cut to trapezoids, and the sides of these trapezoids inside the strip do not overlap at all. Will move from side to side

these trapezoids from the bottom up, and put the area of trapezoids, making sure that each piece was taken exactly once. In fact, this process is very similar to the processing of nested parentheses. Adding the area of trapezoids within each band, and adding the results for all the bands, and we will find the answer - the area of combining triangles.

Consider again the process of adding space trapezes already in terms of implementation. We iterate through all sides of the triangles, and if any party (not vertical, we do not need the vertical sides, and on the contrary, will greatly disturb) falls into this vertical strip (fully or partially), then we put it away in a vector , it's best to do it in this form: Y coordinate of the points of intersection with the side boundaries of the vertical scroll, and the number of the triangle. After this, we constructed a vector containing pieces

sides sort it meaningfully Y: first the left Y, then the right Y. As a result, the first vector element containing the underside of the lowest trapezoid. Now we just go to the resulting vector. Let i - the current item; this means that the i-th piece - is some downside trapezoid some block (which can contain multiple trapezoids), which area we want to add to the answer immediately. So we set a counter to 1 triangles, and climb up on the segments and increment the counter if we find any side of the triangle for the first time, and decrease the counter, if we find a triangle for the second time. If at any interval j counter becomes equal to zero, we

found the upper limit of the block - on this we stay, add the area of the trapezoid bounded by segments i and j, i and assign j +1, and repeat the whole process again.

So, thanks to vertical decomposition method we solved the problem of geometric primitives using only the intersection of two segments.

Implementation

```
struct segment {
    int x1, y1, x2, y2;
};

struct point {double
    x, y;
};

struct item {
    double y1, y2;
    int triangle_id;
};

struct line {
    int a, b, c;
};

const double EPS = 1E-7;

void intersect (segment s1, segment s2, vector<point> & res) {
    line l1 = {s1.y1-s1.y2, s1.x2-s1.x1, l1.a * s1.x1 + l1.b *
        s1.y1}, l2 = {s2.y1-s2.y2, s2. x2-s2.x1, l2.a * s2.x1 +
        l2.b * s2.y1};
    double det1 = l1.a * l2.b - l1.b * l2.a;
    if (abs (det1) <EPS) return;
```

```

point p = {(l1.c * 1.0 * l2.b - l1.b * 1.0 * l2.c) / det1,
            (l1.a * 1.0 * l2.c - l1.c * 1.0 * l2.a) / det1};
if (px> = s1.x1-EPS && px <= s1.x2 + EPS && px> = s2.x1-EPS && px
<= s2.x2 + EPS)
    res.push_back (p);
}

double segment_y (segment s, double x) {
    return s.y1 + (s.y2 - s.y1) * (x - s.x1) / (s.x2 - s.x1);
}

bool eq (double a, double b)
    {return abs (ab) <EPS;
}

vector <item> c;

bool cmp_y1_y2 (int i, int j)
    {const item & a = c
     [i]; const item & b = c
     [j];
    return a.y1 <b.y1-EPS || abs (a.y1-b.y1) <EPS && a.y2 <b.y2-EPS;
}

int main () {

    int n;
    cin >> n;
    vector <segment> a (n *
3); for (int i = 0; i <n;
+ + i) {
        int x1, y1, x2, y2, x3, y3;
        scanf ("% d% d% d% d% d", &x1, &y1, &x2, &
y2, &x3, &y3); segment s1 = {x1, y1, x2, y2};
        segment s2 = {x1, y1, x3,
y3}; segment s3 = {x2, y2,
x3, y3}; a [i * 3] = s1;
        a [i * 3 +1] =
s2; a [i * 3
+2] = s3;
    }

    for (size_t i = 0; i <a.size();
+ +i) if (a[i].x1> a [i].
x2)
        swap (a [i]. x1, a [i]. x2), swap (a [i]. y1, a [i].
y2);

    vector <point> b;
    b.reserve (n * n *
3);
    for (size_t i = 0; i <a.size (); + + i)
        for (size_t j = i +1; j <a.size ();
+ + j) intersect (a [i], a
[j], b);

    vector <double> xs (b.size ());
    for (size_t i = 0; i <b.size ();
+ + i)
        xs [i] = b [i]. x;
    sort (xs.begin (), xs.end ());
    xs.erase (unique (xs.begin (), xs.end (), & eq), xs.end ());

    double res = 0; vector
<char> used (n);
    vector <int> cc (n *
3); c.resize (n * 3);
    for (size_t i = 0; i +1 <xs.size (); + +
i) {double x1 = xs [i], x2 = xs
[i +1]; size_t czs = 0;

```

```
for (size_t j = 0; j <a.size ();
     ++ j) if (a [j]. x1! = a
[j]. x2)
        if (a [j]. x1 <= x1 + EPS && a [j]. x2> = x2-
EPS) {item it = {segment_y (a [j],
x1), segment_y (a [j], x2), (int) j / 3};
cc [csz] = (int)
csz; c [csz + +] =
it;
}
sort (cc.begin (), cc.begin () + csz, & cmp_y1_y2);
```

```
double add_res = 0;
for (size_t j = 0; j < csz;) {
    {item lower = c[cc[j + +]];
    used[lower.triangle_id] = true; int cnt = 1;
    while (cnt && j < csz) {
        char & cur = used[c[cc[j + +]].triangle_id];
        cur = !cur;
        if (cur) ++cnt; else -cnt;
    }
    item upper = c[cc[j-1]];
    add_res += upper.y1 - lower.y1 + upper.y2 - lower.y2;
}
res += add_res * (x2 - x1) / 2;
}

cout.precision(8);
cout << fixed << res;
}
```

Check points on the convex polygon belonging

Given a convex polygon with N vertices, the coordinates of all vertices are integers (although it does not change the essence of the decision); vertices are given in order of the counter-clockwise (otherwise you just need to sort them). Receives requests - point, and is required to identify each point, it lies inside this polygon or not (polygon boundaries included). For each request will respond in a mode on-line in O ($\log N$). Pretreatment of the polygon will be performed in O (N).

Algorithm

Will solve **binary search on the corner**.

One solution is as follows. Choose a point with the smallest coordinate X (if there are several, then choose the lowest, ie, with the lowest Y). Concerning this point, we denote it Zero, all other vertices of the polygon lie in the right half-plane. Furthermore, we note that all the vertices of the polygon is arranged in the corner relative to the point Zero (this follows from the fact that a convex polygon, and is already ordered counterclockwise), and all angles are in the range $(-\pi / 2; \pi / 2]$.

Let receives another request - a point P. Consider its polar angle with respect to the point of Zero. Binary search will find two such neighboring vertices L and R polygon that polar angle P lies between the polar angles L and R. Thus, we find that sector of the polygon, in which lies the point P, and we only need to check whether the point P lies in the triangle (Zero, L, R). This can be done, for example, using [Oriented area of the triangle](#) and [Predicate "Clockwise"](#). Just look at clockwise or counter tops is a triple (R, L, P).

Thus, we in O ($\log N$) sector we find a polygon, and then O (1) check the identity point of the triangle, and hence the required asymptotic behavior is achieved. Pretreatment

Only the polygon is to predposchitat polar angles at all points, although these computations can be carried on the same step of the binary search.

Notes on the implementation

To determine the polar angle, you can use the standard function atan2. Thus, we get a very short and simple solution, but instead may have problems with accuracy.

Given that initially all the coordinates are integers, it is possible to obtain a solution, generally does not use fractional arithmetic.

Note that the polar angle of point (X, Y) relative to the origin is determined uniquely fraction Y / X, provided that the point is in the right half. Moreover, if one point of the polar angle is less than the other, and then the fraction is less Y_1/X_1 Y_2/X_2 , and vice versa.

Thus, for comparison of the polar angles of the two points, it suffices to compare fractions and Y_1/X_1 Y_2/X_2 , it is already possible to perform integer arithmetic.

Implementation

This implementation assumes that there are no duplicate this polygon vertices and the area of the polygon is nonzero.

```
struct pt {
    int x, y;
};

struct ang {
    int a, b;
};

bool operator <(const ang & p, const ang & q)
{
    if (pb == 0 && qb == 0)
        return p.a < q.a;
    return p.a * 111 * q.b < p.b * 111 * q.a;
}

long long sq (pt & a, pt & b, pt & c) {
    return ax * 111 * (by-cy) + bx * 111 * (cy-ay) + cx * 111 * (ay-by);
}
```

```

int main () {

    int n;
    cin >> n;
    vector <pt> p
    (n); int zero_id
    = 0;
    for (int i = 0; i <n; + + i) {
        scanf ("% d% d", & p [i]. x, & p [i]. y);
    if (p [i]. x <p [zero_id]. x | | p [i]. x == p [zero_id]. x && p [i]. y
<p [zero_id]. Y)
            zero_id = i;
    }
    pt zero = p [zero_id];
    rotate (p.begin (), p.begin () + zero_id, p.end
    ());
    p.erase (p.begin ());
    - N;

    vector <ang> a (n);
    for (int i = 0; i <n; + + i) {
        a [i]. a = p [i]. y -
        zero.y; a [i]. b = p [i].
        x - zero.x; if (a [i]. a
        == 0)
            a [i]. b = a [i]. b <0? 1: 1;
    }

    for (; ; ) {
        pt q; / / Another request
        bool in = false;
        if (q.x >= zero.x)
            if (q.x == zero.x && q.y == zero.y)
                in = true;
            else {
                ang my = {q.y-zero.y, q.x-zero.x};
                if (my.a == 0)
                    my.b = my.b <0? 1: 1; vector <ang>
                    :: iterator it = upper_bound (a.
begin (), a.end (), my);
                if (it == a.end () && my.a == a [n-1]. a &&
                my.
b == a [n-1]. b)
                    it = a.end () -1;
                if (it! = a.end () && it! = a.begin ())
                    {int p1 = int (it - a.begin ());
                    if (sq (p [p1], p [p1-1], q) <=
                    0)
                        in = true;
                    }
                }
            puts (in? "INSIDE": "OUTSIDE");
    }

}

```

Finding the inscribed circle of a convex polygon using the ternary search

Given a convex polygon with N vertices. Required to find the coordinates of the center and radius of the largest inscribed circle.

It describes a simple method to solve this problem by using two ternary search working for $O(N \log_2 C)$, where C - coefficient and a coordinate value defined by the required accuracy (Cf. below).

Algorithm

We define the function **Radius (X, Y)** Returning radius of the inscribed circle in the given polygon with center at (X; Y). It is assumed that X and Y lie within (or edge) of the polygon. Obviously, this feature is easy to implement with the asymptotic **O (N)** - Just go through all the sides of the polygon, consider for each distance to the center (where the distance can be taken as from direct to the point, is not necessarily regarded as a segment), and return the minimum distance of the found - obviously it will be the largest radius.

So, we need to maximize this function. Note that, as a convex polygon, then this function is suitable for **Ternary search** in both arguments: for a fixed X_0 (of course, this, that the line $X = X_0$ crosses polygon) function **Radius (X0, Y)** as a function of one variable Y will first increase and then decrease (again, we consider only the Y, the point (X_0, Y) the polygon). Moreover, the function **max (at Y) {Radius (X, Y)}** as a function of one argument X will first increase and then decrease. These properties are clear from geometric considerations.

Thus, we need to do two ternary search: X and inside it to Y, maximizing the value of Radius. The only special moment - you need to choose the right border of ternary searches as a function evaluation **Radius** outside the polygon will be incorrect. To search for X no difficulty, simply choose the abscissa of the leftmost and rightmost point. To search for a Y are those segments of the polygon, which gets the current X, and find the ordinates of the points of these segments at the abscissa X (vertical segments is not considered).

It remains to estimate **asymptotics**. Let the maximum value that can take us - it is C_1 , and the required accuracy - about $10 \cdot C_2$, and let $C = C_1 + C_2$. Then the number of steps that will have to commit each ternary search, is the value of $O(\log C)$, and the final asymptotic behavior is obtained: $O(N \log_2 C)$.

Implementation

Constant steps determines the number of steps both ternary search.

In the implementation is worth noting that for each side immediately predposchityvayutsya coefficients in the equation line u immediately normalized (divided by $\sqrt{A^2 + B^2}$), to avoid unnecessary operations in the ternary search.

```
const double EPS = 1E-9;
int steps = 60;
struct pt {
    double x, y;
};

struct line {
    double a, b, c;
};

double dist (double x, double y, line & l)
    {return abs (x * la + y * lb + lc);
}

double radius (double x, double y, vector <line> & l)
    {int n = (int) l.size ();
    double res = INF;
    for (int i = 0; i <n; + + i)
        res = min (res, dist (x, y, l
[i])); return res;
```

```

}
double y_radius (double x, vector <pt> & a, vector <line> & l)
{
    int n = (int) a.size ();
    double ly = INF, ry = -INF;
    for (int i = 0; i <n; + + i)
    {
        int x1 = a [i]. x, x2 = a [(i + 1)% n]. x, y1 = a [i]. y, y2
        = a
        [(i + 1)% n]. y;
        if (x1 == x2) continue;
        if (x1 > x2) swap (x1, x2), swap (y1, y2); if
        (x1 <= x + EPS && x - EPS <= x2) {
            double y = y1 + (x - x1) * (y2 - y1) / (x2 - x1);
            ly = min (ly, y);
            ry = max (ry, y);
        }
    }
    for (int sy = 0; sy < steps; + + sy)
    {
        double diff = (ry - ly) /
        3;
        double y1 = ly + diff, y2 = ry - diff;
        double f1 = radius (x, y1, 1), f2 = radius (x, y2, 1);
        if (f1 < f2)
            ly = y1;
        else
            ry = y2;
    }
    return radius (x, ly, 1);
}

```

```

int main () {

    int n; vector
    <pt> a (n);
    Reading ... a ...

    vector <line> l (n);
    for (int i = 0; i <n; + + i) {
        l [i]. a = a [i]. y - a [(i
        + 1)% n]. y; l [i]. b = a [(i
        + 1)% n]. x - a [i]. x;
        double sq = sqrt (l [i]. a * l [i]. a + l [i]. b
        * l [i]. b); l [i]. a /= sq, l [i]. b /= sq;
        l [i]. c = - (l [i]. a * a [i]. x + l [i]. b * a [i]. y);
    }

    double lx = INF, rx = -INF;
    for (int i = 0; i <n; + + i)
    {
        lx = min (lx, a [i].
        x); rx = max (rx, a
        [i]. x);
    }

    for (int sx = 0; sx < stepsx; + + sx)
    {
        double diff = (rx - lx) /
        3;
        double x1 = lx + diff, x2 = rx - diff;
        double f1 = y_radius (x1, a, 1), f2 = y_radius (x2, a, 1);
        if (f1 < f2)
            lx = x1;
        else
            rx = x2;
    }
}
```

```
double ans = y_radius (lx, a, l);
printf ("% .7 lf", ans);
}
```

Finding the inscribed circle of a convex polygon method

"Compression of the parties" ("shrinking sides") for $O(n \log n)$

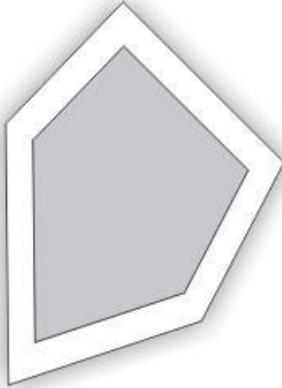
Given a convex polygon with n vertices. Required to find the inscribed circle of maximum radius, ie, to find the radius and center coordinates. (If at a given radius, multiple centers, it is sufficient to find any of them.)

Unlike described [here](#) Ternary search of the double asymptotics of the algorithm - $O(n \log n)$. does not depend on the location and limits of the required accuracy, and therefore this algorithm is suitable for considerably higher n and more restrictions on the value of the coordinates.

Thank you **Ivan Krasil'nikov (Mf)** for this beautiful description of the algorithm.

Algorithm

So, given a convex polygon. Begin simultaneously with the same speed **shift** all its sides are parallel to each other inside the polygon:



Suppose, for convenience, this movement occurs at a rate of 1 unit per second coordinate (ie the time in some sense equal to the distance: later point in time, each unit will overcome a distance equal to one).

During this movement of the polygon will gradually fade (contact point). Sooner or later the whole polygon shrinks to a point or a segment, and this time t will be **answer Problem** - desired radius (And the center of the desired circle will lie in this interval). In fact, if we squeezed polygon thickness t in all directions, and he appealed to the point / segment, then it means that there is a point distant from all sides of the polygon at a distance t . And for long distances - such a point does not exist.

So, we need to learn how to effectively simulate the process of compression. To learn this for each side **determine the time** through which it will shrink to a point.

For this, consider carefully the process of moving parties. Note that the vertices of the polygon always move along the bisectors of angles (this follows from the equality of the corresponding triangles). But then the question of time, through which the compressed side, reduced to the question of determining the height H triangle, which is known side length L and two adjacent thereto angle α and. Using, for example, the sine theorem, we obtain the formula:

$$H = L \cdot \frac{\sin \alpha \cdot \sin \beta}{\sin(\alpha + \beta)}.$$

Now we know for $O(1)$ determine the time in which the party will shrink to a point.

These times are listed for each side in some **structure for extracting data minimum**, eg red-black tree (**set** in the language of C ++).

Now, if we extract the side with **the least time** H , This side of the first compressed to the point - in time H . If a polygon has not shrunk to the point / segment, then this aspect should **remove** of the polygon, and continue the algorithm for the remaining sides. When you delete a part, we need to **join** each other of its left and right neighbors, **extending** them to their point of intersection. This will need to find this point of intersection, recalculate the length of two sides and their times of extinction.

When implemented for each side will have to keep the number of its right and left neighbor (and thus how to build a doubly linked list of sides of the polygon). This allows for the removal of the binding of the two sides and its neighbors for $O(1)$.

If you delete the part is that its side-neighbors **parallel**, This means that the polygon is then compression point degenerates in / cut, so we can immediately stop the algorithm, and to return as a response time of disappearance current side (so that problems do not arise parallel sides).

If such a situation does not arise parallel sides, the algorithm to finalize the moment in which a polygon will be only two parties - and then the answer to the problem will be the removal of the previous hand.

Obviously, the asymptotic behavior of this algorithm is because the algorithm consists of n steps, each of which is removed on one side (which produced several operations set during $O(n \log n)$).

Implementation

We present the implementation of the algorithm described above. This implementation returns only the desired radius of the circle; however, the addition of O center of the circle will not be hard.

This elegant algorithm that from computational geometry requires only finding the angle between the two sides, the intersection of two lines and two lines check for parallelism.

Note. It is assumed that the signal input to the polygon - **strictly convex** no three points lie on one line.

```

const double EPS = 1E-9;
const double PI = ...;

struct pt {
    double x, y;
    pt() {}
    pt (double x, double y) : X(x), Y(y) {}
    pt operator- (const pt & p) const {
        return pt (x-p.x, y-p.y);
    }
};

double dist (const pt & a, const pt & b) {
    return sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

double get_ang (const pt & a, const pt & b) {
    double ang = abs (atan2 (a.y, a.x) - atan2 (b.y, b.x));
    return min (ang, 2* PI-ang);
}

struct line {
    double a, b, c;
    line (const pt & p, const pt & q) {
        a = p.y - q.y;
        b = q.x - p.x;
        c = - a * p.x - b * p.y;
        double z = sqrt (a * a + b * b); a /= z, b /= z, c /= z;
    }
};

double det (double a, double b, double c, double d) {
    return a * d - b * c;
}

pt intersect (const line & n, const line & m) {
    double zn = det (n.a, n.b, m.a, m.b);
    return pt (
        - det (n.c, n.b, m.c, m.b) / zn,
        - det (n.a, n.c, m.a, m.c) / zn
    );
}

```

```

bool parallel (const line & n, const line & m) {
    return abs (det (n.a, n.b, m.a, m.b)) <EPS;
}

double get_h (const pt & p1, const pt & p2,
              const pt & l1, const pt & l2, const pt & r1, const pt & r2)
{
    pt q1 = intersect (line (p1, p2), Line (l1, l2));
    pt q2 = intersect (line (p1, p2), Line (r1,
        r2));double l = dist (q1, q2);
    double alpha = get_ang (l2 - l1, p2 - p1) / 2;
    double beta = get_ang (r2 - r1, p1 - p2) / 2;
    return l * sin(alpha) * sin(beta) / sin(alpha + beta);
}

struct cmp {
    bool operator() (const pair <double,int> & A, const pair
<double,int> & B) const {
        if (abs (a.first - b.first) > EPS)
            return a.first <b.first;
        return a.second <b.second;
    }
};

int main() {
    int n;
    vector <pt> p;
    Reading ... n and p ...

    vector <int> Next (n), Prev
(n);for (int i =0; i <n; + + I)
{
    next[i] = (i + 1) % N;
    prev[i] = (i - 1 + N) % N;
}

set <pair <double,int>, Cmp> q;
vector <double> H (n);
for (int i =0; i <n; + +
I) { h[i] = Get_h
(
    p[i], P[next[i]],
    P[i], P[prev[i]],
    p[next[i]], P[next[next[i]]]
);
    q.insert (make_pair (h[i], I));
}

double last_time;
while (q.size() > 2) {
    last_time = q.begin() ->
First; int i = q.begin() ->
Second; q.erase (q.begin());
    next[prev[i]] = Next[i];
    prev[next[i]] = Prev[i];
    int nxt = next[i], Nxtl = (nxt +1)% N, prv
        = prev[i], Prvl = (prv +1)% N;
    if (parallel (line (p[nxt], P[Nxtl]), Line (p[prv], P[Prvl])))
        break;
    q.erase (make_pair (h[nxt], Nxt));
    q.erase (make_pair (h[prv], Prv));
    h[nxt] = Get_h ( p[nxt],
                    P[Nxtl],
                    P[prv], P[prv],
                    p[next[nxt]]+1) % N
);
    h[prv] = Get_h ( p[prv],
                    P[Prvl],

```

```
p[ (prev[prv]+1) % N], P[prev[prv]],
```

```
    p[nxt], P[nxt1]
);
q.insert (make_pair (h[nxt], Nxt));
q.insert (make_pair (h[prv], Prv));
}

cout << Last_time << endl;
}
```

The main function here - it *get_h()* that on the side and its left and right neighbors calculates time of disappearance of the hand. Sought for this intersection point of this side with the neighbors, and then the above formula calculation is made of the desired time.

Voronoi diagram in 2D

Definition

Dana n points $P_i(x_i, y_i)$ plane. Consider the partition of the plane into n areas V_i (Called Voronoi polygons or Voronoi cells, sometimes - polygons proximity cells Dirichlet partition Thyssen), where - the set of all points in the plane that are closer to the point P_i than to any other point P_k :

$$V_i = \{(x, y) : \rho((x, y), P_i) = \min_{k=1\dots N} \rho((x, y), P_k)\}$$

Needless partition of the plane is called a Voronoi diagram given set of points P_k .

Here $\rho(p, q)$ - Given metric, usually the standard Euclidean metric: but below will be considered and the case (so $(x_p - x_q)^2 + (y_p - y_q)^2$) called the Manhattan metric. Here and below, unless otherwise specified, will be considered for the Euclidean metric

Voronoi cells are convex polygons, some are endless.

Points belonging to the definition of multiple cells at once Voronoi and usually relate directly to several cells (in the case of Euclidean metric set of points of measure zero, in the case of the Manhattan metric bit more complicated).

These polygons were first studied in depth by the Russian mathematician Voronoi (1868-1908 gg.).

Properties

- Voronoi diagram is a planar graph, so it has $O(n)$ vertices and edges.
- Fix any $i = 1 \dots n$. Then for each $j = 1 \dots n, j \neq i$ draw the line - perpendicular bisector of the segment (P_i, P_j) ; consider that the half-plane formed by the straight line in which point lies. Then the intersection of all half-planes for each j give Voronoi cell. P_i
- Each vertex of the Voronoi diagram is the center of a circle drawn through any three points of the set P . These circles are essentially used in many proofs related to Voronoi diagrams.
- Voronoi cell V_i is infinite if and only if point P_i lies on the boundary of the convex hull of the set P .
- Consider the graph dual to the Voronoi diagram, ie in this graph vertices are points and the edge conducted between points P_i and P_j have a common edge. Then, with the proviso no four points lie on a circle to the dual graph of the Voronoi diagram is the Delaunay triangulation (having many interesting properties).

Application

Voronoi diagram is a compact data structure that stores all the information needed to solve many problems of intimacy.

As discussed below, it is the time required to build most of the Voronoi diagram, in the asymptotics is not considered.

- Finding the nearest point for each.

Note the simple fact that if the nearest point is the point, this point is "their" in the cell edge V_i . It follows that to find for each point nearest to it is enough to see her ribs Voronoi cell. However, each edge belongs to exactly two cells, so it looked exactly twice,

and due to the linearity of the number of edges we obtain a solution for this problem $O(n)$.

- Finding the convex hull.

Recall that the vertex belongs to the convex hull if and only if its Voronoi cell is infinite. Then we find in the Voronoi diagram any infinite edge and begin to move in a fixed direction (eg, counterclockwise) on the cell containing this edge until we reach

until the next infinite edges. Then go through this edge in the adjacent cell and continue to crawl. In result, all viewed ribs (except infinite) will be sought by the parties of the convex hull.

Obviously, the run time - $O(n)$.

- Finding the Euclidean minimum spanning tree.

Find the minimum spanning tree with vertices at these points P . Connecting all these points. If we apply the standard methods of graph theory, then, because graph in this case has no edges, even the optimal algorithm will have no less asymptotics.

Consider the graph dual Voronoi diagram, ie Delaunay triangulation. It can be shown that the presence of Euclidean minimum spanning tree is equivalent to constructing the core Delaunay triangulation. Indeed, [Prim's algorithm](#) each time sought the shortest edge between two points mzhestvami ; if we fix one set point, the point closest to it has an edge in the Voronoi cell, so the Delaunay triangulation will present an edge to the nearest point, as required.

Triangulation is a planar graph, ie by a linear edges, so it is possible to apply [Kruskal's algorithm](#) and an algorithm with running time $O(n \log n)$.

- Finding the largest empty circle.

You want to find the largest radius circle that does not contain any of the points inside (center of the circle must lie inside the convex hull points). Note that because the largest radius of curvature function at this point is strictly monotonic within each Voronoi cell, it reaches its maximum

$$(x, y)$$

at one of the vertices of the Voronoi diagram, or at the intersection of ribs and a convex hull diagram (the number of such points is more than twice the number of edges of the diagram). Thus, it remains only to sort out these points and for each to find the nearest, ie solution for $O(n)$.

Simple algorithm for constructing the Voronoi diagram for $O(n^4)$

Voronoi diagram - quite well studied object, and for them to get a lot of different algorithms for optimal working asymptotic behavior, and some of these algorithms even work

$$O(n \log n)$$

average $O(n)$. However, these algorithms are very complex.

We consider here the simplest algorithm based on the above property that each cell

Voronoi is the intersection of half-planes. Fix. Spend between point i and each point P_j line - the perpendicular, then cross the pairs all received direct - get $O(n^2)$ points, and each belonging to check all the half-planes. As a result, we get all the vertices of the Voronoi cell (they would no longer be a n , So we can sort without impairing their asymptotic behavior in the polar angle), and only require the construction of the Voronoi diagram $O(n^4)$ action.

Special case of the metric

Consider the following metric:

$$\rho(p, q) = \max(|x_p - x_q|, |y_p - y_q|)$$

Consideration should start with simple case - a case of two points A and B .

$$A_y = B_y$$

If $A_x = B_x$ or, the Voronoi diagram for them to be respectively vertical or horizontal line.

Otherwise Voronoi diagram will look like "corner": Cut at an angle 45 degrees in the rectangle formed by the points A and B And the horizontal / vertical beams of the end thereof depending on whether the longer side of the rectangle of the vertical or horizontal.

Special case - when this rectangle has the same length and width, i.e. $|A_x - B_x| = |A_y - B_y|$
In this case, there will be two infinite regions ("corners" formed by two rays parallel to the axes), which by definition must belong to both of the cells. In this case, further comprises determining in the condition as it is understood these areas (sometimes artificially introduced a rule that every corner referred to his cell).

Thus, even for two points Voronoi diagram in this metric is a non-trivial object, and in the case of a larger number of points, these figures need to be able to quickly cross.

Finding all the faces, the outer edge of a planar graph

Dan planar stacked on a plane graph with n vertices. Want to find all its facets. Face is the part of the plane bounded by the edges of the graph.

One of the faces will be different from others in that it will have an infinite area, such a face is called the outer face.

Some problems need to find only the outer edge, the algorithm for finding which, as we shall see, in fact no different from the algorithm for all faces.

Euler's theorem

We present here a few Euler's theorem and its corollaries, of which it will follow that the number of edges and faces of a planar simple (without loops and multiple edges) of the graph are of the order $O(n)$.

Let a planar graph G is connected. We denote n the number of vertices in the graph, m - The number of edges - the number of faces. Then we have **Euler's theorem**:

$$f + n - m = 2$$

Prove this formula easily follows. In the case of wood ($m = n - 1$) Formula is easily verified. If the graph - not a tree, then remove any edge belonging to any cycle; wherein the amount not change. We will repeat this process until you come to a tree for which the identity is already installed. Thus, the theorem is proved.

$$\begin{aligned} f + n - m \\ f + n - m = 2 \end{aligned}$$

Consequence. For any planar graph let k - The number of connected components. Then we have:

$$f + n - m = 1 + k$$

Consequence. The number of edges m simple planar graph is a value $O(n)$.

Proof. Suppose that the graph is connected and (in the case $n \geq 3$) assertion is obtained automatically.) Then, on the one hand, each facet is bounded by at least three ribs. On the other hand, each edge limits the maximum two faces. Consequently,

$$3f \leq 2m$$

Whence, substituting this into the formula Euler, we obtain:

$$\text{Ie } m = O(n).$$

If the graph is not connected, then summing the resulting estimates of its connected components, we again obtain $m = O(n)$. As required.

Consequence. Number of faces f simple planar graph is a value $O(n)$. This follows from the preceding corollary and Communications $f = 2 - n + m$.

Bypassing all facets

Always assume that the graph, if it is not connected, laid on the plane so that no connected component is not contained within another (eg, a square with lying strictly inside the segment

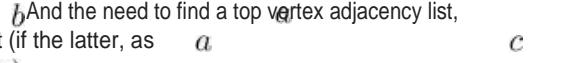
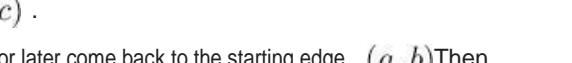
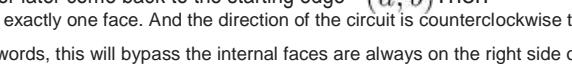
- Incorrect algorithm for our test).

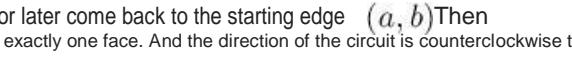
Naturally, it is considered that the graph correctly laid on the plane, i.e. no two vertices do not coincide, and the edges do not intersect in the "unauthorized" locations. If the input graph admits such intersecting ribs, the pre-need to get rid of them, entering into every point of intersection of an additional vertex (it should be noted that as a result of this process, instead of n points we can get the order of points). More information about this process, see the applicable section below.

Suppose that for each vertex of it all outgoing edges are ordered by the polar angle. If it is not, then blocked should be streamlined by performing sorting each adjacency list (because $m = O(n)$). It required $O(n \log n)$ operations).

Now we choose an arbitrary edge (a, b) and let the next rounds. Coming in at the top of some some edge, out of this summit we must on the next edge in the sort order.

v

For example, in the first step we are in the top  And the need to find a top vertex adjacency list, then we denote the next node in the adjacency list (if the latter, as  take the first vertex), and pass along the edge .

By repeating this process many times, we will sooner or later come back to the starting edge  Then to stop. Easy to see that with this we will bypass traversing exactly one face. And the direction of the circuit is counterclockwise to the outer edge, and clockwise - for internal faces. In other words, this will bypass the internal faces are always on the right side of the current edge.

So we learned to circumvent one face, starting from any edge on its boundary. You're starting to learn to choose the edges so that the resulting faces are not repeated. Note that each edge two different ways in which it can get: Each of them will receive their faces. On the other hand, it is clear that one oriented edge belongs to exactly one edge.

Thus, if we will mark all edges each detected faces from an array and do not run circumvention of already labeled edges, then we will bypass all the faces (including foreign), even just once.

Present directly **implementation** this bypass. We assume that the graph adjacency lists already ordered by the corner, and multiple edges and loops are absent.

A first embodiment of easy, the next node in the adjacency list he is looking for a simple search. Such an implementation is theoretically works for, although in practice many tests, it works very quickly

(With a hidden constant, significantly less than unity).

```

int n; // the number of vertices
vector<vector<int>> G; // graph

vector<vector<char>> Used
(n); for (int i = 0; i < n; ++ I)
    used[i]. Resize (g[i].
Size()); for (int i = 0; i < n; ++ I)
    for (size_t j = 0; j < g[i]. Size(); ++ J)
        if (! Used[i] [j]) {
            used[i] [j] =
true;
            int v = g[i] [j], Pv = i;
            vector<int> Facet;
            for (,;)
                facet.push_back (v);
            vector<int> :: Iterator it = find (g[v]. Begin(),
g[v]. End(), Pv);
                if (++ It == g[v]. End()) it = g[v].
Begin();
                if (used[v] [it-g[v]. Begin()])
break; used[v] [it-g[v]. Begin()] =
true;
                Pv = v, v = * it;
}
Conclusion ... facet - current faces ...
}

```

Another embodiment of a more streamlined - enjoys the fact that the top of the list in order of adjacency corner. If you implement a function **Cmp_ang** comparing two points in the polar angle with respect to the third point (eg by issuing it as a class, as in the example below), the search terms in the adjacency list, you can use binary search. As a result, we obtain a realization for $O(n \log n)$.

```

class Cmp_ang {
    int center;
public:
    Cmp_ang (int center) : Center(center)
    {}
    bool operator() (int a, int b) const {
        Should return ... trueIf a point is smaller than b
        polar angle relative to center ...
    }
};

int n; // the number of vertices
vector<vector<int>> G; // graph

vector<vector<char>> Used
(n); for (int i = 0; i < n; ++ I)

```

```
used[i].Resize(g[i].  
Size());for (int i =0; i <n; + + I)
```

```

        for (size_t j =0; j <g[i]. Size(); + + j)
            if (! Used[i] [j]) {
                used[i] [j] =
                    true;
                int v = g[i] [j], Pv = i;
                vector <int> Facet;
                for (,; ) {
                    facet.push_back (v); vector <int> :::
                    Iterator it = lower_bound (g
[v]. Begin(), G[v]. End(),
                                         pv, cmp_ang (v));
                    if (++ It == g[v]. End()) it = g[v].
Begin();
                    if (used[v] [it-g[v]. Begin()])
break; used[v] [it-g[v]. Begin()] =
true;
                    pv = v, v = * it;
                }
            Conclusion ... facet - current faces ...
        }
    }
}

```

And possible option based on container *map*. Because we only need to quickly learn the position numbers in the array. Of course, such an implementation would also work $O(n \log n)$.

It should be noted that the algorithm is not quite working correctly with **isolated** peaks - such peaks he just does not detect as individual faces, though, from a mathematical point of view, they must be a single connected component and faces.

In addition, special edge is **outside face**. As distinguished from its "normal" faces, described in the next section. It should be noted that if the graph is not connected, the external face will consist of several circuits, and each of these circuits will be found by the algorithm separately.

Isolation of the outer edge

The above code displays all the faces, making no distinction between the outer edge and inner edges. In practice, in contrast, requires only the found or external face or internal only. There are several methods of allocating the outer edge.

For example, it can be determined from the area - the outer face must have the largest area (should only take into account that the inner face can have the same area as the external). This method will not work if the planar graph G is not connected.

Another more reliable criterion - towards the bypass. As noted above, all the faces except the outer crawling in a clockwise direction. Outer edge, even if it consists of several circuits, cost algorithm counterclockwise. Determine the direction of the circuit by simply considering *landmark polygon area*. Area can be considered directly in the course of the inner loop. However, in

This method has its subtlety - processing faces zero area. For example, if the graph consists of a single edge, then the algorithm will only face whose area is zero. Apparently, if a face area is zero, then it is the outer edge.

In some cases it is also applicable criteria such as the number of vertices. For example, if the graph is a convex polygon with it held in non-intersecting diagonals, its outer face will contain all the vertices. But again you have to be careful with the case where the external and internal faces have the same number of vertices.

Finally there is a method for finding and following the outer edge: you can start from a specially such edge that was found in the outer face will result. For example, you can take the left-most vertex (if there are several, it will suit any) and select from it the edge, going first in the sort order. As a result of this tour will rib outer face. This method can be extended to the case of a disconnected graph: you need to find each connected component of the leftmost vertex and run circumvention of the first edge of it.

We present the implementation of a very simple method based on the sign area (itself bypassing I took for example, it does not $O(n^2)$ matter here). If the graph is not connected, the code "... the external face is ..." executed separately for each circuit constituting the outer face.

```

... Normal code to detect faces ...
Immediately after ... cycle, detecting
another side: ...

        // consider      the
        area double area =
        0;
        // add a dummy point for simplicity
counting area

```

```
facet.push_back (facet[0]);
```

```

        for (size_t k =0; k +1<Facet.size(); ++ K)
            area += (p[facet[k]]. First + p[facet[k]
+1]]. First)
                                * (p[facet[k]]. Second - p[facet[k
+1]]. Second);
                if (area <EPS)
                    The outer face is ... ...
            }

```

Construction of a planar graph

For the above algorithms is essential that the input graph is correctly packed planar graph. However, in practice, often fed to the input of the program set of segments, possibly overlapping each other in "unauthorized" places, and need to build on these segments planar graph.

Implement the construction of a planar graph can be as follows. Fix any input segment. Now this cross section with all other segments. Point of intersection points, as well as the ends of the interval set in the vector, and sort the standard way (ie first one coordinate, with equal - on the other). Then go through this vector and will add edges between adjacent points in this vector (of course, making sure that we have not added loops). After completing this process for all segments, ie for $O(n^2 \log n)$. We construct the corresponding planar graph (which will $O(n^2)$ points).

Implementation:

```

const double EPS = 1E-9;

struct point {
    double x, y;
    bool operator < (const point & p) const {
        return x < p.x - EPS || abs (x - p.x) <EPS && y <p.y - EPS;
    }
};

map <point,int>
Ids; vector <point>
p;
vector <vector <int>> G;

int get_point_id (point pt) {
    if (! Ids.count (pt))
    {
        ids[pt] = (int)p.size();
        p.push_back (pt);
        g.resize (g.size() + 1);
    }
    return ids[p];
}

void intersect (pair <point,point> a, pair <point,point> b, vector <point> &
res) {
    Standard procedure ... intersects two segments a and b,
and throws the result in res ...
    ... If the segments overlap, then throws those ends that fall
inside the first segment ...
}

int main() {
    // input int m;
    vector <pair <point,point>> a (m);
    Reading ... ...

    // Graphing
    for (int i =0; i <m; + +
        I) { vector
        <point> cur;
        for (int j =0; j <m; + + J)
            intersect (a[i], A[j], Cur);
        sort (cur.begin(), Cur.end());
        for (size_t j =0; j +1<Cur.size(); + + J) {

```

```
int x = get_id (cur[j]), Y = get_id (cur[j +1]);  
if (x! = y) {
```

```
        g[x]. Push_back
        (y); g[y].
Push_back (x);
    }
}

int n = (int) g.size();
// Sort by angle and removing multiple edges
for (int i =0; i <n; + + I) {
    sort (g[i]. Begin(), G[i]. End(), Cmp_ang (i));
    g[i]. Erase (unique (g[i]. Begin(), G[i]. End()), G[i]. End());
}
}
```

Finding the closest-pair

Statement of the Problem

Given n points p_i on the plane defined by its coordinates (x_i, y_i) . Required to find two of them are point, the distance between them is minimal:

Distance we take the usual Euclidean:

Trivial algorithm - through all the pairs and calculating the distance for each - for works $O(n^2)$. This algorithm has been proposed $\Omega(n \log n)$ preparations (Preparata) in 1975 Drug and Shamos also revealed that the decision tree model, this algorithm asymptotically optimal.

Algorithm

Construct an algorithm for the general scheme of algorithms "Divide-and-conquer": Algorithm executed in the form of a recursive function, which transmits a plurality of pixels; This recursive function is the set of splits in half, calling itself recursively on each half, and then performs any business combination answers. Merge operation is to detect when a single point

Optimal solutions fell into one half, and another point - to another (in this case, a recursive call of each of the halves individually detect the pair, of course, can not.) The main difficulty, as always, lies in the effective implementation of this stage of the association. If recursive function transmits a plurality of n points, then the stage of integration should work no more than, then the asymptotic behavior of the whole algorithm $O(n)$ will be given by:

$$T(n) = 2T(n/2) + O(n).$$

The solution of this equation is known to be $T(n) = O(n \log n)$.

So, proceed to the construction of the algorithm. To come in the future to effectively implement the steps of combining, split into two set of points according to their will x -Coordinates: in fact we spend some vertical line which divides the set of points into two subsets of approximately equal

sizes. Such a decomposition is convenient to make as follows: sort the points as a standard pair of numbers, ie:

$$p_i < p_j \iff (x_i < x_j) \vee ((x_i = x_j) \wedge (y_i < y_j)).$$

Then we take the average after sorting point $p_{m/2}$, p_m referred to the first half, and all points afterwards - in the second half of:

Now called recursively on each of the sets A_1 and A_2 answers h_1 and h_2 for each halves. Take the best of them: $h = \min(h_1, h_2)$.

Now we need to make the step of combining try to find such a pair of points, And the other - distance between which is less than, one point lies in the in . Obviously, for it is sufficient to consider only those points which are spaced from the vertical line section is not distance minimal, i.e. many considered at this stage points is:

For each point of the set must try to find the points that are closer to it than . Eg it suffices to consider only those points whose coordinates differ by no more than . Furthermore, no

$$B = \{p_i \mid |x_i - x_m| < r_f\}.$$

$$\begin{matrix} B \\ y \\ h \\ h \end{matrix}$$

make sense to consider those points which y Coordinate more y Coordinates of the current point. Thus,

for each point p_i define the set of points under consideration $C(p_i)$ follows:

If we sort the points of B on x -coordinate, then find $C(p_i)$ will be very easy: it is somewhat points straight to the point.

So, in the new notation **stage of integration** as follows: to construct a set

B , Sort it according to terms y -Coordinate, then for each point $p_i \in B$ consider all the points

$p_i \in C(p_i)$ And each pair (p_i, p_j) Calculate the distance and compare it with the current best distance.

At first glance, it is still sub-optimal algorithm: it seems that the size of the sets are of the order n ,

and the required asymptotic behavior does not work. However, surprisingly, we can prove that the size of each set

is the value that does not exceed a certain small constant irrespective

points themselves. The proof is given in the next section.

Finally, pay attention to the sorting algorithm described above which contains just two: first sorting

pairs $(,)$, and then sorting the elements of \dots on. In fact, both of these sorts

inside the recursive function can be eliminated (otherwise we would not have reached

evaluation xy for the steps of combining,

asymptotic behavior of the algorithm and the total

$O(n)$

would have been $O(n \log^2 n)$. From the first sort is easy to get rid - sufficiently in advance of the launch of recursion to perform this sort: in fact inside recursion elements themselves do not change, so there is no need to sort again. With the second sorting little harder to fulfill its pre-fail. But, remembering **mergesort** (Merge sort), which also works on the principle of divide-and-conquer, you can simply embed this sort in our recursion. Let recursion, taking a set of points (as we recall, ordered pairs (x, y))

Returns the same set, but already sorted coordinate y . To do this, simply merge (for) two results returned by recursive calls. Thereby get sorted by y set.

Rating asymptotics

To show that the above algorithm is indeed satisfied for prove the following fact:

It remains

So, let us consider some point. Recall that the set $|C(p_i)| = O(1)$

$O(n \log n)$

coordinate is no more

But not less

And, moreover, in the

points of the set

y_i

lie in a strip

y_i

in other words, that we consider the point

lie in the rectangle $C(p_i)$

A set of points, -

and the point

itself

And all

p_i

$C(p_i)$

Our task - to estimate the maximum number of points that can be in this rectangle

;

thus we estimate the maximum size of the set and

(It is one less, because in

This rectangle is also a point). At the same time we must not forget that in the general case can occur

and duplicate points.

$2h \times h$

Recall that obtained as the result of at least two recursive calls - from sets

an

contains a point on the left of the line section and partially

- The remaining points of the line section and point to

A_1 thereon,

A_2 the right

therefrom. For any pair of points A_1 As well as of A_2 The distance can not be less than - of

it meant incorrectness recursive function.

d

Wherein

To estimate the maximum number of points in the rectangle $2h \times h$ divide it into two squares $h \times h$ To

first put all square point $C(p_i) \cap A_1$ And the second - the rest, ie $C(p_i) \cap A_2$. Of

the above considerations, it follows that in each of these squares distance between any two points does not exceed

h . But then this means that in each square is not more than four points?

Indeed, suppose there is a square $h \times h$. And the distance between any two points does not exceed the same

. We prove that the square can not be more than 4 points. For example, this can be done as follows:

divide this square into 4 squares with sides $h/2$. Then in each of these small squares can not

be more than one point (because even diagonal equal $h/\sqrt{2}$ than h). Consequently, squared way around

can not be more than 4 points.

So, we have proved that in a rectangle $2h \times h$ can not be more

$4 \cdot 2 = 8$ points and, therefore,

size of the set $C(p_i)$ can not exceed

7

, As required.

Implementation

We introduce a data structure for storing point (its coordinates and a certain number) and comparison operators are needed for the two types of sorting:

```

struct pt {
    int x, y, id;
};

inline bool cmp_x (const pt & a, const pt & b) {
    return a.x < b.x || a.x == b.x && a.y < b.y;
}

inline bool cmp_y (const pt & a, const pt & b) {
    return a.y < b.y;
}

pt a[MAXN];

```

For easy implementation of recursion, we introduce an auxiliary function `upd_ans()` which computes distance between two points and check whether it is not better than the current response:

```

double mindist;
int ansa, ansb;

inline void upd_ans (const pt & a, const pt & b) {
    double dist = sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) + .0);
    if (dist < mindist)
        mindist = dist, ansa = a.id, ansb = b.id;
}

```

Finally, the implementation of most of the recursion. It is assumed that before calling the array is already sorted by x -Coordinate. Recursion is simply passed two pointers r which indicate that it should seek an answer to. If the distance between r and l too small, the recursion must stop and perform trivial algorithm to find the nearest pair and then sort on the subarray y -Coordinate.

To merge two sets of points obtained from the recursive call, one (ordered by-coordinate) we use the standard STL function `merge()` and create a secondary buffer $t[]$ (One to all recursive calls). (Use `inplace_merge()` inexpedient, since it generally does not work in linear time).

Finally, the set B stored in the same array t .

```

void rec (int l, int r) {
    if (r - l <= 3) {
        for (int i = l; i <= r; ++I)
            for (int j = i + 1; j <= r; +
                + J) upd_ans (a[i],
                    A[j]);
        sort (a + l, a + r + 1, &
            Cmp_y); return;
    }

    int m = (l + r) >> 1;
    int midx = a[m]. X;
    rec (l, m), Rec (m + 1,
        R); static pt t[MAXN];
    merge (a + l, a + m + 1, A + m + 1, A + r + 1,
        T, & cmp_y); copy (t, t + r - l + 1, A + l);

    int tsz = 0;
    for (int i = l; i <= r; ++I)
        if (abs (a[i]. X - midx) < Mindist) {
            for (int j = tsz - 1; j >= 0 && A[i]. Y - t[j]. Y < mindist;
            - J)
                upd_ans (a[i], T[j]);
            t[tsz + +] = A[i];
        }
}

```

By the way, if all the coordinates are integers, then for the duration of recursion can never move to fractional values, and stored in `mindist` squared minimum distance.

In the main program should call recursion as follows:

```
sort (a, a + n, &
cmp_x); mindist =
1E20;
rec (0, N-1);
```

Summary: search triangle with minimal perimeter

The algorithm described above is interesting generalized to this problem: a given set of points choose from three different points so that the sum of pairwise distances between them was minimal.

In fact, to solve this problem the algorithm remains the same: we share the field into two halves vertical line, recursively call the decision of both halves, choose at least \minper of found perimeters, building a strip thickness $\minper/2$ And it sort out all the triangles which can improve the response. (Note that the perimeter of the triangle \minper longest side $\leq \minper/2$.)

Problem in online judges

List of tasks that can be reduced to finding the next two points:

- [UVA 10245 "The Closest Pair Problem"](#) [Difficulty: Easy]
- [SPOJ # 8725 CLOPPAIR "Closest Point Pair"](#) [Difficulty: Easy]
- [CODEFORCES Team Olympiad Saratov - 2011 "Minimum Amount"](#) [Difficulty: Medium]
- [Google CodeJam 2009 Final "Min Perimeter"](#) [Difficulty: Medium]
- [SPOJ # 7029 CLOSEST "Closest Triple"](#) [Difficulty: Medium]

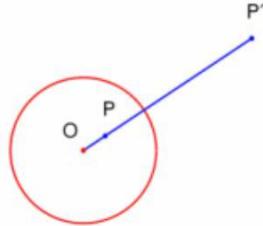
Geometric transformation inversion

Geometric transformation inversion (inversive geometry) - a special type of conversion points on the plane. The practical benefit of this transformation is that it often allows to reduce the solution of a geometrical problem **with circles** to solve the corresponding problem **with straight** which typically has a much simpler solution.

Apparently, the founder of this branch of mathematics was Ludwig Immanuel Magnus (Ludwig Immanuel Magnus), who in 1831 published an article on inverse transformation.

Definition

Fix a circle centered at the radius r . Then **inversion** point on this circle is called a point P' which lies on the line OP . And a distance condition is imposed:



If we assume that the center of the circle coincides with the origin, we can say that the point P' has the polar angle, and that, as the distance calculated by the above formula.

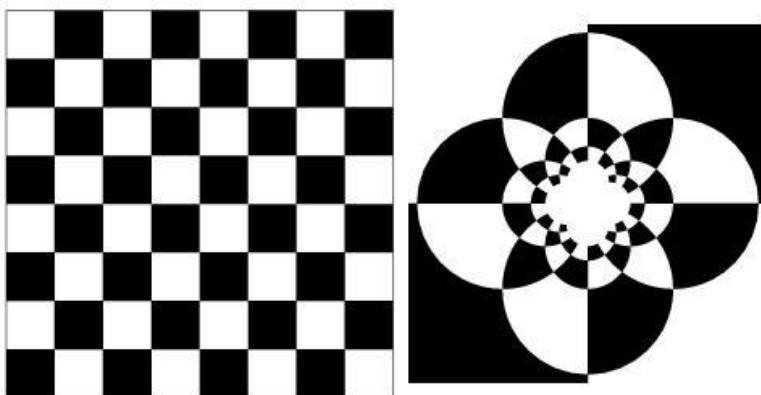
In terms of **complex numbers** inversion transformation is expressed quite simply, if we assume that the center O circle coincides with the origin:

$$z' = r^2 \cdot \frac{z}{|z|^2}.$$

With the mating member \bar{z} can be a simpler form:

$$z' = \frac{r^2}{\bar{z}}.$$

Application of inversion (at mid-board) to the image of a chessboard provides an interesting picture (right):



Properties

Obviously, any point lying **on the circumference**, with respect to which the inversion transformation, when displaying a same goes. Any point lying **inside** circumference passes into **external** region, and vice versa. It is believed that the center point of the circle becomes "infinite" ∞ And point "infinity" - on the contrary, at the center of a circle:

$$(O)' = \infty, \\ (\infty)' = O.$$

Obviously, the repeated use of the inversion transformation **draws** her first application - all points are returned:

$$(P')' \equiv P.$$

Generalized circle

Generalized circle - this is either a circle or a straight line (it is believed that this is also a circle, but having infinite radius).

A key property of the inversion transformation - that when applied generalized circle **always translate into a generalized circle** (Assuming that the transformation pointwise inversion is applied to all points in the figure).

Now we'll see what happens with the lines and circles in the conversion of inversion.

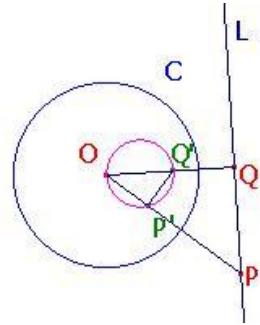
Inversion of the line passing through the point O

It is argued that any straight line passing through O After inversion transformation **unchanged**.

In fact, any point on this line, and in addition ∞ , By definition, also goes to the point of this line (and eventually get the point entirely fill the entire line as the inversion transformation is reversible). Points remain O and ∞ But the inversion they pass each other, so the proof is complete.

Inversion line not passing through the point O

It is argued that any such move straight **in the circle** Passing through O .



Consider any point P this line, and we also consider the point - the nearest O point of the line. It is clear that the segment OQ perpendicular to the line, but because the angle formed by them $\angle P'Q'O$ - Direct.

We now use **Lemma Equal angles** That we shall prove later, this lemma gives us the equality

$$\angle Q'QO = \angle Q'P'O.$$

Therefore, the angle $\angle Q'P'O$ also direct. As we take any point, it turns out that the point P' lies on the circle, built on OQ' the entire this circle as a whole, hence the assertion.

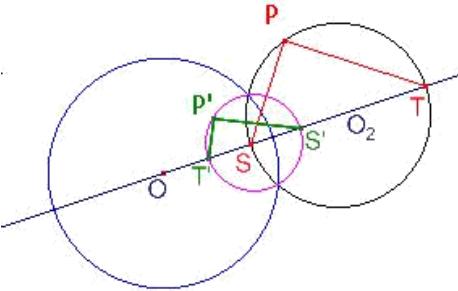
Inversion of a circle passing through the point O

Any such circle goes **the direct** Not passing through the point O .

In fact, it follows immediately from the previous paragraph, if we recall the reversibility of the inversion transformation.

Inversion circle not passing through the point O

Any such circle goes **in the circle** Still does not pass through point O .



In fact, we consider any such circle
circles straight; this line intersects the circle

with center O_2 . Connect centers and
 Z at two points O_2 and (obviously O_2 Diameter).

Now consider any point on the circle
by the corollary **lemma about equal angles** also should be direct and angle
that the
point P lies on the circle, built on the interval
imaginary P' eventually cover the circle.

Z . Angle $S T$ direct for any such point, but

Z , Which implies
 $S' T'$ as diameter. Again, it is easy to understand that all

: Otherwise the
 O point ∞ would be

Clearly, this new circle can not pass through
belong to the old circle.

Lemma about equal angles

This accessory is a property that was used in the above analysis results inversion transformation.

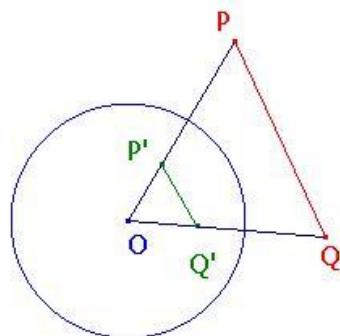
Formulation

Consider any two points P and Q and apply to them the inversion transformation, we obtain the points P' and Q' . Then the following angles are equal:

$$\angle PQO = \angle Q'P'O, \\ \angle QPO = \angle P'Q'O.$$

Proof

Prove that the triangle $\triangle PQO$ and $\triangle Q'P'O$ similar (the order of vertices is important!).



In fact, by definition, have an inversion transformation:

$$OP \cdot OP' = r^2, \\ OQ \cdot OQ' = r^2,$$

from which we obtain:

$$OP \cdot OP' = OQ \cdot OQ', \\ \frac{OP}{OQ} = \frac{OQ'}{OP'}.$$

Thus, the triangles have a common angle, and the two sides adjacent thereto are proportional, therefore, these triangles are similar, and therefore the same respective angles.

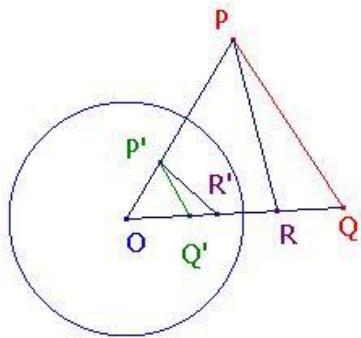
Consequence of Lemma

If there are any three points P, Q, R , The point R lies on the segment OQ . Then performed:

$$\angle QPR = \angle Q'P'R',$$

these angles being oriented in opposite directions (i.e., when viewed as the two angle-oriented, then

they are of opposite sign).

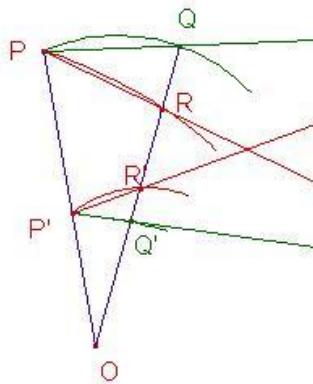


For proof, note that
can apply Lemma Equal angles:
- Is the difference of two
 $\angle QPR$ angles $\angle QPO$ and $\angle RPO$, Each of which

In the implementation of the last transition we changed the order of the points, which means that we have changed the orientation angle of the opposite.

Conformality

Inversion transformation is conformal, ie **preserves angles at points of intersection curves**. Thus, if the angles considered as oriented, the orientation angles in the application inversion is reversed.



For **evidence** this, consider two arbitrary curves intersect at P and having it tangents. Suppose on the first point of the curve will go Q , On the second - the point R (We we let them in the limit P).

Obviously, after the application of inversion curves will continue to intersect (unless, of course, they do not pass through the point O). But such a case we do not see), and their intersection point is P' .

Given that the point R lies on the line joining O and Q we find that we can apply the corollary to Lemma Equal angles from which we obtain:

$$\angle QPR = -\angle Q'P'R',$$

where under the sign "minus" we mean that the angles are oriented in different directions.

Letting terms and R to point P , In the limit we obtain that equality - an expression of the angle between the intersecting curves, as required.

Reflection property

If M Generalized circle, then when you convert it inversion **saved** if and only if M **orthogonal** circle with respect to which the inversion (and considered to be different).

The proof of this property is interesting because it **demonstrates** application of geometric inversion to avoid the circles and the task.

The first step **evidence** is an indication of the fact that intersection. In fact, the inversion transformation her appearance, and vice versa.

Time M after the conversion has not changed, it means that it contains both points

and have at least two points
 M displays in the interior of the circle
 C

from the interior and the exterior of the circle. This implies that the points of intersection of the two (the one she can not may - which means two touching circles, but in this case, obviously, be the condition can not exist; match circumferentially and are not, by definition.)

Denote one point of intersection through M , Another - through M' . Consider a circle with center point and perform the inversion transformation for her. Note that if and circle, and generalized circle C must pass intersecting lines. Given the conformity

inversion transformation, we find that M and M' coincide if and only if the angle between two these intersecting straight line (in fact, the first inversion transformation - respect, - Changes the direction of the angle between the circles on the opposite, so if the circle coincides with its inversion, the angles between intersecting lines on both sides must be the same and equal $\frac{180}{2} = 90$ degrees).

Practical application

Immediately it should be noted that when used in the calculations must take into account a large error introduced by transformation inversion may appear very small fractional number of orders, and is usually due to high error inversion method works well only with relatively small coordinates.

Constructing shapes after inversion

In computing software is often more convenient and reliable to use not ready-made formulas for the coordinates and radii of the resulting generalized circles, and to restore every time straight / circle at two points. If the recovery is sufficient to take direct any two points and calculate their images and combine line, then the circles are much more complicated.

If we want to find the circumference, resulting in a direct inversion, according to the above calculations, it is necessary to find the nearest to the center of the inversion point of the line, apply to it inversion (received

a certain point Q' , And then the desired circle will have a diameter OQ' .

Suppose now that we want to find the circumference, resulting in another circle of inversion. Generally speaking, the center of the new circle - not coincides with the center of the old circle. To determine the center of the new circle can take this concept: to navigate through the inversion center of the circle and the center of the old line, to see her point of intersection with the old circle - let it be and point. Segment forms a circle diameter of the old, and easy to understand that after the inversion of this segment will continue to be

form diameter. Consequently, the center of the new circle can be found as the arithmetic average points S' and T' .

Options circle after inversion

$$(x_0, y_0)$$

Required for a given circle (the known coordinates of its center and radius r_0) To determine in advance which it enters the circle after conversion inversion circle with center and radius r .

Ie we solve the problem described in the previous paragraph, but we want to obtain a closed

form solution. The answer appears in the form of formulas:

$$\begin{aligned} x' &= x_c + s(x_0 - x_c), \\ y' &= y_c + s(y_0 - y_c), \\ r' &= |s| \cdot r_0, \end{aligned}$$

where

$$s = \frac{r^2}{(x_0 - x_c)^2 + (y_0 - y_c)^2 - r_0^2}.$$

Mnemonically these formulas can remember this: the circle center moves "almost" as to transform inversion, only the denominator in addition $|z|^2 = (x_0 - x_c)^2 + (y_0 - y_c)^2$ have yet to subtract r_0^2 .

These formulas are derived exactly as described in the preceding paragraph algorithm: are expressions for the two diametrical points and then applied to them inversion, and then the arithmetic mean of their coordinates. Similarly, we can calculate the radius as half the length of the segment ST .

Application in evidence: the partition problem points circle

Given the various points on the plane, as well as an arbitrary point different from the rest. Prove that there is a circle passing through the point O Such that inside and outside the same number would lie

set points, i.e. by n pieces.

For **evidence** we perform the inversion transformation relative to the selected point (with any radius, for example, n). Then the desired circle will match a line not passing through the point O .

And on one side of the line is the half-plane corresponding to the inner circle, and on the other

Appropriate appearance. It is clear that there always exists a line which divides the set of n points into two halves by n points, and thus passes through point O . (For example, a straight line can be obtained, turning the whole picture by any such corner to nor in any of the considered coordinates, and then just taking a vertical line between n Th and $n+1$ Nd points). This direct

matches the search of a circle passing through the point O and hence the assertion.

Used to solve problems in computational geometry

A remarkable property of geometric inversion - that in many cases it can simplify geometric problem posed by replacing circles consideration only direct examination.

Ie if the problem is quite complicated operations with various circles, it makes sense to apply to the input data inversion transformation, try to solve the problem without resulting modified circles (or a smaller number of them), and then re-use inversion to obtain the solution of the original problem.

An example of this problem is described in the next section.

Steiner chain

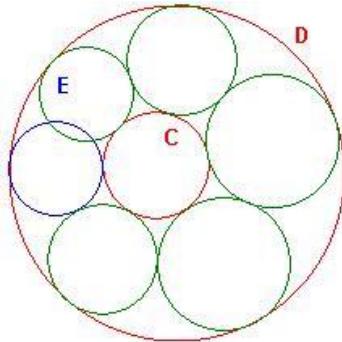
C D

E

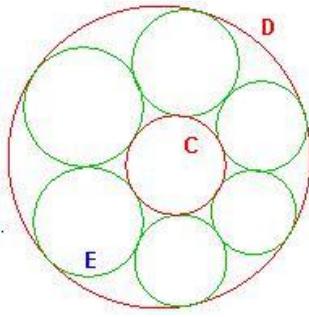
Given two circles and one is strictly inside the other. Then, the third circle is drawn regarding these two circles, whereupon an iterative process is started each time a new circle is drawn so that it concerned previous painted, and the first two.

Sooner or later, another draw a circle intersect with one of the previous set, or at least touch it.

Case of the intersection:



Touch case:



Accordingly, our task - to put as much as possible circles, so that the intersection (i.e., the first of the presented cases) were not. The first two circles (external and internal) are fixed, we can only vary the position of the first relating to the circle, then everything is uniquely placed on the circumference.

If you touch the receiving chain circles called **Steiner chain**.

With this so-called chain of linked **Steiner** (Steiner's porism): if there is at least one set of Steiner (ie, there is a corresponding provision relating to starting a circle, leading to a chain of Steiner), then for any other choice regarding starting circle will also receive a Steiner chain, and the number of circles in her will same.

From this statement it follows that in the solution to maximize the number of circles the answer does not depend on the position of the first set of the circle.

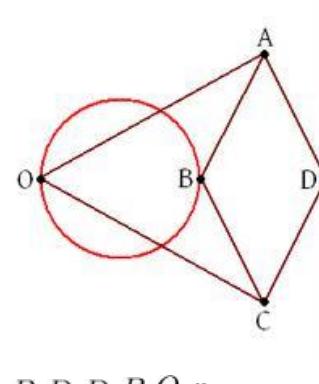
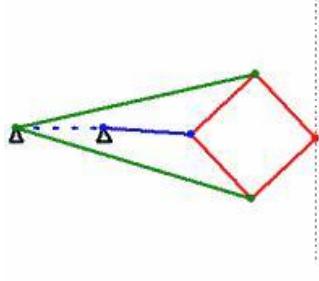
Proof and constructive algorithm for solving the following. Note that the problem has a very

simple solution in the case where centers of the outer and inner circles coincide. Clearly, in this case the number of circles set does not depend on the first set. In this case, all circles have the same radius, and their number and the coordinates of the centers can be calculated using simple formulas.

To go to this simple situation of any supplied to the input, apply the inversion transformation with respect to a circle. We need the center of the inner circle and moved coincided with the center of the outer, so look for a point with respect to which we take the inversion, it is necessary only on the line connecting the centers of the circles. Using the formula for the coordinates of the center of the circle after use inversion can equate the position of the center of inversion, and to solve this equation. Thus we have from any situation can go to a simple, symmetric case, and, having solved the problem for him, re-apply the inversion transformation and obtain the solution of the original problem.

Application technique: right-Lipkin Peaucellier

For a long time the problem of transforming the circular (rotational) motion into linear remained very challenging in engineering, managed to find at best approximate solutions. And only in 1864 an officer of the French army corps of engineers Charles Nicolas Peaucellier (Charles-Nicolas Peaucellier) and in 1868



$B D D B O r.$

an
d

, , , , . Point moves along a circle passing through the point.

a

n

d

lie on a line (this follows from

For evidence we note first that the points

$AB BC CD DA$ lie on a line (this follows from the equality of triangles). We denote the point of intersection of the segments OB and OD as P . We introduce the notation:

O AB BC CD DA P B D A C BD

We need to show that the quantity $OB \cdot OD = \text{const}$:

By the Pythagorean theorem, we obtain:

Take the difference between these two quantities:

Thus, we have proved that $OB \cdot OD = \text{const}$, which means that D - Inversion point B .

The search for common tangent to two circles

Given two circles. Required to find all of their common tangents, ie all such lines that relate to both circles simultaneously.

The described algorithm will work also in a case where one (or both) of the circle degenerate into points. Thus, this algorithm can also be used to find the tangent to a circle passing through a given point.

Number of common tangents

Just note that we do not consider **degenerate** cases: when the circle are the same (in this case, they are infinitely many common tangents), or one circle is inside the other (in this case, they have no common tangents, or if the circle concerned, there is one common tangent).

In most cases, have two circumferential **four** common tangents.

If the circle **concern**, They will have three obshih tangents, but it can be understood as a degenerate case: as if two tangents coincide.

Moreover, the algorithm described below will operate in the case where one or both of the radius of the circle are zero: in this case, respectively, one or two common tangent.

To summarize, we, except as described in the early cases, will always seek **four tangents**. In degenerate cases, some of them will be the same, but nevertheless, these cases will also fit into the overall picture.

Algorithm

For simplicity of the algorithm, we assume without loss of generality that the center of the first circle has coordinates $(0; 0)$ (If not, then this can be achieved by a simple shift of the whole picture, and after finding a solution

- The shift of their direct back.)

We denote r_1 and r_2 the radii of the first and second circles, and by v - Coordinates of the center of the second circle (point v is different from the origin, as we do not consider the case when the same circle, or one circle is inside the other).

To solve the problem come to clean it **algebraically**. We need to find all lines of the form, which lie at a distance r_1 from the origin, and the distance r_2 from point v .

In addition, we impose the condition of normalization direct sum of the squares of the coefficients a and should be equal to one (it must, otherwise the same straight line will correspond to an infinitely many views species $ax + by + c = 0$).

Total obtain a system of equations for the unknown a, b, c :

$$\begin{cases} a^2 + b^2 = 1, \\ |a \cdot 0 + b \cdot 0 + c| = r_1, \\ |a \cdot v_x + b \cdot v_y + c| = r_2. \end{cases}$$

To get rid of the modules, we note that all there are four ways to uncover modules in this system. All of these methods can be considered a general case, if we understand how disclosure of the module that the coefficient on the right side may be multiplied by -1 .

In other words, we come to such a system:

$$\begin{cases} a^2 + b^2 = 1, \\ c = \pm r_1, \\ a \cdot v_x + b \cdot v_y + c = \pm r_2. \end{cases}$$

Introducing the notation $d_1 = \pm r_1$ and $d_2 = \pm r_2$ We arrive at the fact that four times the system must address:

$$\begin{cases} a^2 + b^2 = 1, \\ c = d_1, \\ a \cdot v_x + b \cdot v_y + c = d_2. \end{cases}$$

The solution of this system is reduced to solving a quadratic equation. We omit all the tedious calculations, and immediately give a ready answer:

$$\begin{cases} a = \frac{(d_2-d_1)v_x \pm v_y \sqrt{v_x^2 + v_y^2 - (d_2-d_1)^2}}{v_x^2 + v_y^2}, \\ b = \frac{(d_2-d_1)v_y \mp v_x \sqrt{v_x^2 + v_y^2 - (d_2-d_1)^2}}{v_x^2 + v_y^2}, \\ c = d_1. \end{cases}$$

Overall we've got 8 solutions instead 4. But it is easy to understand, in what place there extraneous solutions: in fact, in the latter system is sufficient to take only one solution (eg, the first). In fact, the geometric meaning of what we take and clear: we actually sort out which side

$$\pm r_1 \quad \pm r_2$$

from each of the circles will direct. So two ways arising in the solution of the latter system, redundant: just select one of two solutions (only, of course, in all four cases, select one and the same family of solutions).

The last thing we have not yet considered - it **as a direct shift** in the case where the first circle does not initially was at the origin. However, here is simple: the linearity of the line equation that

ratio of c should take the value of $a \cdot x_0 + b \cdot y_0$ (Wherein x_0 and y_0 - Coordinates of the center of the original the first circle).

Implementation

We first describe all the necessary data structures and other auxiliary definitions:

```
struct pt {
    double x, y;

    pt operator- (pt p) {
        pt res = { x-p.x, y-p.y };
        return res;
    }
};

struct circle: pt {
    double r;
};

struct line {
    double a, b, c;
};

const double EPS = 1E-9;

double sqr (double a) {
    return a * a;
}
```

Then the decision itself can be written in such a way (where the main function to call - the second: and the first function - Auxiliary):

```
void tangents (pt c, double r1, double r2, vector <line> & ans)
{
    double r = r2 - r1;
    double z = sqr(c.x) + Sqr(c.y);
    double d = z - sqr(r);
    if (d <-EPS) return; d
    = sqrt (abs (d)); line
    l;
    l.a = (c.x * r + c.y * d) / z;
    l.b = (c.y * r - c.x * d) / z;
    l.c = r1;
    ans.push_back (l);
}

vector <line> tangents (circle a, circle b)
{
    vector <line> ans;
    for (int i = -1; i <=1; i +=2)
```

```
        for (int j = -1; j <=1; j +=2)
            tangents (b-a, a.r * i, b.r * j,
ans);for (size_t i =0; i <ans.size(); ++ I)
    ans[i]. C -= ans[i]. A * a.x + ans[i]. B *
a.y; return ans;
}
```

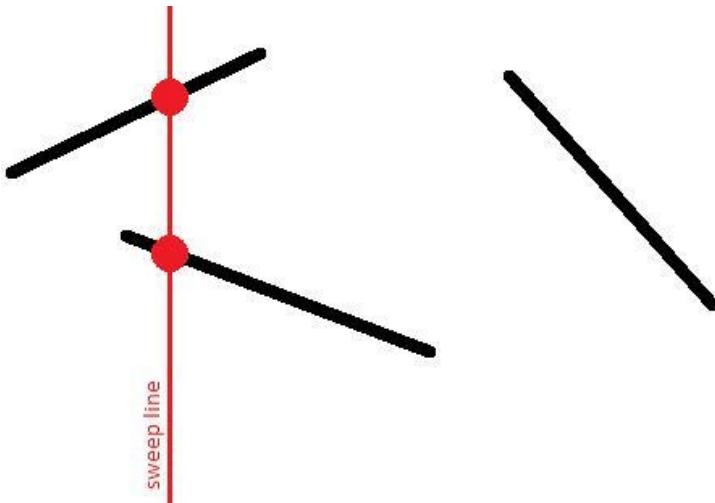
Find pairs of intersecting segments algorithm swept by lines for O (N log N)

Dana n segments in the plane. Need to check whether intersect with each other at least two of them. (If the answer is yes - then bring this pair of intersecting segments, among quite a few answers to choose any of them.)

Naive algorithm for solving - for bust $O(n^2)$ all pairs of segments check for each pair intersect or not. In this article we describe an algorithm with running time $O(n \log n)$ Which is based on principle **Scanning (Sweep out) direct** (In English: "sweep line").

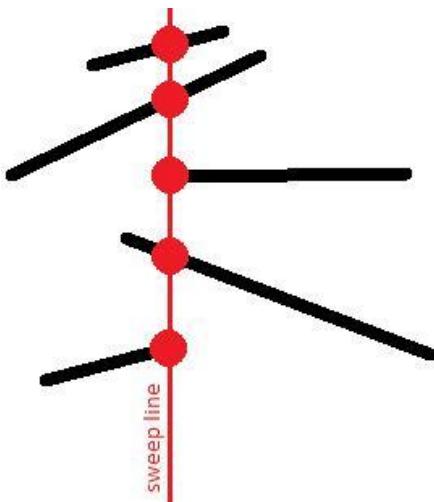
Algorithm

We draw a vertical line mentally $x = -\infty$ and start moving this straight right. In the course of its motion this line will meet with the segments, and at any given time each segment will intersect with our direct at one point (we'll consider that there is no vertical segments).

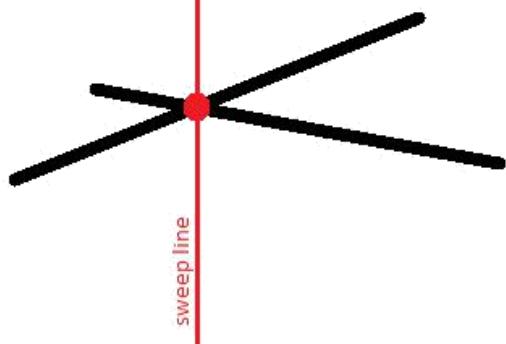


Thus, for each segment at some point of time it will appear on the scanning line, then will move straight ahead and this point, and finally, at some point disappear straight segment.

We are interested in **the relative order of the segments** vertically. Namely, we will keep a list of segments intersecting the scanning line at a time where cuts will be sorted by their y -Coordinate on-line scanning.



This order is interesting in that overlapping segments will have the same y -Coordinate of the at least one time:



Formulate key statements:

- To find intersecting pairs is sufficient to consider for each fixed position scanning line **Only adjacent segments.**
- It suffices to consider the scanning line is not valid in all possible positions $(-\infty \dots +\infty)$
and only in those positions, when new segments or disappear old. In other words, enough to confine only provisions equal abscissas-ends of the segments.
- When a new segment is sufficient **insert** it in the right place in the list obtained for a previous scanning line. Should be checked for intersection **only added to the segment its immediate neighbors in the list, the top and bottom.**
- With the disappearance of the segment enough **remove** from its current list. After this you should **verify at the intersection with the upper and lower neighbors** list.
- Other changes in the order of the segments in the list, other than as described, does not exist.
Other checks to make crossing is not necessary.

To understand the truth of these assertions are the following observations:

- Two disjoint segment never change their **relative order.**
Indeed, if one segment was initially higher than the other, and then became lower, between these two moments was the intersection of these two segments.
- Have matching **y**-Coordinates two disjoint segment also can not.
- From this it follows that at the moment when we find a segment in the queue position of this segment, and rearrange over this segment does not have the queue, his **order relative to other segments queue will not change.**
- Two intersecting segments at their point of intersection will be **neighbors** each other in turn.
- Therefore, to find pairs of intersecting segments enough to cross check all the only pair of segments that ever during the motion of the scanning line at least once **were neighbors of each other.**
Easy to see that it is enough just to check the added segment with its upper and lower neighbors, as well as removing the segment - its upper and lower neighbors (which, after removal become neighbors of each other).
- It should be noted that for a fixed position scanning line we **first** must produce **addition** all segments appearing here, and only **then - removal** all endangered segments here.
Thus, we will not miss intersection of the segments on top: ie such cases when the two segments have a common vertex.
- Note that **vertical segments** in fact does not affect the correctness of the algorithm.

These segments are allocated in order that they appear and disappear at the same time. However, due to the previous remark, we know that the first all segments will be added to the queue, and only then will be removed. Consequently, if the vertical line segment intersects with any other open at this time segment (including vertical), then this is detected.

At what place queued vertical segments? After the vertical segment has no one specific **y**-Coordinates, it extends to the whole segment **y**-Coordinate. But it is easy to understand that as **y**- Coordinates can take any coordinate of this segment.

Thus, the entire algorithm will make no more $2n$ tests on the intersection of a pair of segments, and make operations $O(n)$ with the queue lengths (by $O(1)$ operations in moments of appearance and disappearance of each segment).

Summary **asymptotics** the algorithm is thus $O(n \log n)$.

Implementation

Here are the full realization of the above algorithm:

```
const double EPS = 1E-9;

struct pt {
    double x, y;
};

struct seg {
    pt p, q;
    int id;
    double get_y (double x) const {
        if (abs (p.x - q.x) <EPS)   return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
    }
};

inline bool intersect1d (double l1, double r1, double l2, double r2) {
    if (l1> r1) swap (l1, r1);
    if (l2> r2) swap (l2, r2);
    return max (l1, l2) <= Min (r1, r2) + EPS;
}

inline int vec (const pt & a, const pt & b, const pt & c) {
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    return abs(s)<EPS? 0 : S>0 ? +1 : -1;
}

bool intersect (const seg & a, const seg & b) {
    return intersect1d (a.p.x, a.q.x, b.p.x, b.q.x)
        && intersect1d (a.p.y, a.q.y, b.p.y, b.q.y)
        && vec (a.p, a.q, b.p) * Vec (a.p, a.q, b.q) <= 0
        && vec (b.p, b.q, a.p) * Vec (b.p, b.q, a.q) <= 0;
}

bool operator < (const seg & a, const seg & b) {
    double x = max (min (a.p.x, a.q.x), Min (b.p.x, b.q.x));
    return a.get_y(x) <B.get_y(x) - EPS;
}

struct event {
    double x;
    int tp, id;
    event() {}
    event (double x, int tp, int id) :
        X(x), Tp(tp), Id(id)
    {}
    bool operator < (const event & e) const {
        if (abs (x - e.x) > EPS) return x <e.x;
        return tp> e.tp;
    }
};

set <seg> s;
vector <set <seg> :: iterator> where;

inline set <seg> :: iterator prev (set <seg> :: iterator
    it) { return it == s.begin() ? s.end() : - It;
}

inline set <seg> :: iterator next (set <seg> :: iterator
    it) { return + + It;
```

```

}
pair <int,int> Solve (const vector <seg> & a)
{
    int n = (int) a.size();
    vector <event> e;
    for (int i =0; i <n; + + i) {
        e.push_back (event (min (a[i]. P.x, a[i]. Q.x)+1, I));
        e.push_back (event (max (a[i]. P.x, a[i]. Q.x)-1, I));
    }
    sort (e.begin(), E.end());
}

s.clear();
where.resize (a.size());
for (size_t i =0; i <e.size(); + +
I) { int id = e[i]. Id;
    if (e[i]. Tp == +1) { set
        <seg> :: iterator
            nxt = s.lower_bound (a[id]),
            Prv = prev (nxt);
        if (nxt!= s.end() && Intersect (* Nxt,
            a[id])) return make_pair (nxt-> id,
            id);
        if (prv!= s.end() && Intersect (* Prv,
            a[id])) return make_pair (prv-> id,
            id);
        where[id] = S.insert (nxt, a[id]);
    }
    else {
        set <seg> :: iterator
            nxt = next (where[id]),
            Prv = prev (where[id]);
        if (nxt!= s.end() && Prv!= S.end() &&
intersect (* Nxt, * prv))
            return make_pair (prv-> id, nxt->
            id); s.erase (where[id]);
    }
}

return make_pair (-1-1);
}

```

The main function here - `solve()` That returns numbers found intersecting segments or $(-1, -1)$ If no intersection.

Checking for the intersection of two segments are supplied by the `intersect()` By algorithm based on the oriented area of the triangle.

Queue lengths in the global variable `s` - `set < event >`. Iterators indicating the position each segment in the queue (for easy removal from the queue lengths), are stored in the global array `where`.

Also introduced two auxiliary functions `prev()` and `next()` Which returns an iterator to the previous and the following elements (`or end()` If one does not exist).

Constant `EPS` denotes the error of comparing two real numbers (mostly it is used checking two segments at the intersection).

Z-row function and its calculation

Suppose given a string s length n . Then **Z-function** ("Z-function") from this line - an array of length n , element is equal to the greatest number of characters, starting position i coinciding with the first character s . In other words, $z[i]$ - This is the greatest common prefix of s and i th suffix.

Note. In this article, to avoid indeterminacy, we assume that the string 0-indexed - ie is the first character has an index 0 And the last - $n - 1$.

The first element of Z-functions $z[0]$ is generally considered uncertain. In this article, we will assume that it is zero (although none in the algorithm, any given implementation, this does not change anything).

This article provides an algorithm for calculating the Z-function over time $O(n)$ as well as various applications. This algorithm.

Examples

An example, as computed Z-function for multiple lines:

- "aaaaa":

```
z[0] = 0,  
z[1] = 4,  
z[2] = 3,  
z[3] = 2,  
z[4] = 1.
```

- "aaabaab":

```
z[0] = 0,  
z[1] = 2,  
z[2] = 1,  
z[3] = 0,  
z[4] = 2,  
z[5] = 1,  
z[6] = 0.
```

- "abacaba":

```
z[0] = 0,  
z[1] = 0,  
z[2] = 1,  
z[3] = 0,  
z[4] = 3,  
z[5] = 0,  
z[6] = 1.
```

Trivial algorithm

Formal definition can be represented as the following elementary realization for $O(n^2)$:

```
vector<int> Z_function_trivial (string s)  
{ int n = (int) s.length(); vector  
<int> z (n);  
for (int i = 1; i < n; ++ i)  
    while (i + z[i] < N && s[z[i]] == s[i +  
        z[i]]) ++ z[i];  
    return z;  
}
```

We just for each position i sort out the answer for her $z[i]$. Starting with zero, and as long as we do not

a mismatch is found or we reach the end of the line.

Of course, this implementation is too inefficient, we now turn to the construction of an efficient algorithm.

Efficient algorithm to compute Z-function

To obtain an efficient algorithm to compute the values will in turn - from $i = 1 \text{ to } n - 1$. And wherein when calculating the next try values $z[i]$ maximum use of the calculated values.

Called for brevity substring that matches the prefix of s , **segment matches**. For example, the value of the desired Z-function - is the length of the segment matches, starting at position i .

(And end it will be in a position $i + z[i] - 1$).

To do this, we will support **coordinates of the rightmost segment matches** of all detected segments will keep the one that ends just right. In a sense, the index r - It's such a boundary, to which our line already been scanned by the algorithm, and everything else - is not yet known.

Then, if the current index, for which we want to calculate the next value Z-function - is i . We have one of two options:

- $i > r$ - ie current position is **outside** what we have already processed.

And

Then we look **trivial algorithm**, ie just trying values $z[i] = 0, z[i] = 1$, etc. Note that, in the end, if $z[i] > 0$ we would be required to update the coordinates of the rightmost segment $[l; r]$. Because $i + z[i] - 1$ would be guaranteed more r .

- $i \leq r$ - ie the current position is inside the segment matches $[l; r]$.

Then we can use the already calculated **previous** Z-values of the function to initialize the value $z[i]$ not zero, and maybe some big number.

For this we note that the substring and **coincide**. This means in that the initial approximation for $z[i]$ we can take the corresponding value in the interval $s[0 \dots r - l]$ namely, the value $z[i - l]$. An

However, the value $i - l$ to be too large: so that when it is applied to the position "Come out" of bounds r . This can not be allowed because about the characters to the right r we know nothing, and they may differ from those required.

We present **example** such a situation, for example, line:

aaaavaaa

When we get to the last position ($i = 6$), The current segment will be the rightmost. Items

Given this interval corresponds to the position $6 - 5 = 1$ this value to initialize $z[6]$ impossible, it is completely incorrect. The maximum value of what we could initialize - it 1. Because it is the largest value that is not beyond the interval $[l; r]$. Thus, as **the initial approximation** for $z[i]$ safe to take only such expression:

$$z_0[i] = \min(r - i + 1, z[i - l]).$$

Initializing $z[i]$ this value, we continue to act again **trivial algorithm**

- Because after the border r generally able to detect the segment continued coincidence predict that only one previous values Z-function, we could not.

Thus, the entire algorithm is a two cases differ only in the fact that **initial value** In the first case it is assumed to be zero, while in the second $z[i]$

- Is determined by the previous values for the above formula. After that both branches of the algorithm can be reduced to the implementation of **trivial algorithm**, Which starts immediately with the specified initial value.

The algorithm turned out very simple. Despite the fact that for each i it somehow trivial algorithm runs - we have made significant progress, having an algorithm that runs in linear time. Why this is so, consider the following, after the implementation of the present algorithm.

Implementation

Implementation turns out rather laconic:

```
vector<int> Z_function (string s) {
```

```

int n = (int) s.length();
vector<int> z(n);
for (int i = 1, l = 0, r = 0; i < n; + i)
    if (i <= r)
        z[i] = Min (r-i+1, z[i-1]);
    while (i + z[i] < N && s[z[i]] == s[i + z[i]]);
        z[i]])) ++ Z[i];
    if (i + z[i]-1 > R)
        l = i, r = i + z[i]-1;
}
return z;
}

```

Comment on this implementation.

The whole solution is given as a function that returns an array of string to length n - Computed Z-function.

Array z initially filled with zeros. Current rightmost segment is assumed to be a match, ie, deliberately small segment, which does not get any i .

Inside the loop on $i = 1 \dots n - 1$ we first by the above algorithm determines the initial value

$z[i]$

- Or it will remain zero, or calculated based on the formula above.

$z[i]$

Thereafter, the trivial algorithm that attempts to increase the value of as much as possible.

In the end, the current update of the rightmost segment coincidence, of course, if this update is required - ie if $i + z[i] - 1 > r$.

Asymptotic behavior of the algorithm

We prove that the above algorithm runs in linear time with respect to the length of the string, ie for $O(n)$. The proof is very simple.

We are interested in a nested loop **while** - Because everything else - the only constant operations performed $O(n)$ times.

We show that **each iteration** This cycle **while** lead to an increase in the right border r unit.

For this we consider both branches of the algorithm:

• $i > r$

In this case, either the

circle while will not make a single iteration (if $s[0] \neq s[i]$), Or else make a few iterations, each time advancing by one character to the right, starting from the position i And then - right border

r Be sure to update.

Since $i > r$, We conclude that indeed, each iteration of the cycle increases the new value r unit.

• $i \leq r$

In this case, we initialize the value of the above formula, a certain number of z_0 . Compare this to the initial value z_0 with the value $r - i + 1$. Obtain three choices:

$z_0 < r - i + 1$

We prove that in this case no one iteration cycle **while** will do.

It is easy to prove, for example, on the contrary: if the cycle **while** made at least one iteration, it would mean that we have defined value z_0 was inaccurate, this is less than the length of the match. But since line $s[l \dots r]$ and $[0 \dots r - l]$ coincide, this means that the gap $z[i - l]$ Cost incorrect value: less than should be.

Thus, in this embodiment of correct values $z[i - l]$ and the fact that it is less $- i + 1$ Follows that this value coincides with the desired value $z[i]$.

$z_0 = r - i + 1$

In this case, the cycle **while** can perform several iterations, but each of them will lead to increase the value of the new unit: because we compare the first character is $s[r + 1]$, which climbs beyond the interval $[l; r]$.

$z_0 > r - i + 1$

This option is essentially impossible, by definition z_0 .

Thus, we have proved that each iteration of the inner loop leads to promotion pointer t right. Because t could not be more $n - 1$. This means that all this makes the cycle less than $n - 1$ iteration.

As the rest of the algorithm obviously works for, then we have proved that the whole algorithm $O(n)$ Z-function calculation is performed in linear time.

Applications

Consider several uses Z-functions for specific tasks. These applications will be largely

similar applications [prefix function](#).

Search the substring

To avoid confusion, we call one line **text** t And the other - **sample** p . Thus, the challenge is to find all occurrences of the sample p in text t .

To solve this problem we form a line, ie ~~assign text to the sample through the separator character (which is not found anywhere in the lines themselves).~~

Count for the resulting Z-line function. Then for any interval i $[0; length(t)]$ by the corresponding value $z[i + length(p) + 1]$. You can see whether the sample p with t : if the value is Z-power function $length(p)$. Then yes, included, otherwise - no.

Thus, the asymptotic behavior of the solutions turned $O(length(t) + length(p))$. Memory consumption has the same asymptotic behavior.

Number of distinct substrings in a string

Given a string s length n . Requires her to count the number of distinct substrings.

We solve this problem iteratively. Namely, learn, knowing the current number of different substrings recalculate this amount, when added to the end of one character.

So let - the current number of different substrings s And we append symbol c . Obviously, the result could be some new substring ending in this new symbol c (Namely, all the strings that end with this symbol, but never met before).

Take a string and ~~invert it~~ (write the characters in reverse order). Our task - to count how many rows of prefixes that are not found anywhere else in it. But if we think

Z-string function and find its maximum value Then, obviously, at the first character (not first) its prefix length z_{max} But not greater length. Clearly, a shorter length prefixes already found it exactly.

So, we have found that the number of new substrings that appear when append a character is $len - z_{max}$. Wher
 - The current length of the string after the character c $len - z_{max}$ ein attribution.

Consequently, the asymptotic solutions for the string length is $O(n^2)$. and the number of different

It is worth noting that in exactly the same can be recalculated for $O(n)$ substrings append characters at the beginning, as well as removing the character from the end or the beginning.

Row compression

Given a string s length n . Required to find the shortest its "compressed" representation, ie find the following line t shortest length that s can be represented as a concatenation of one or more copies of t .

To solve calculate Z-line function s And find the first position i such that $i + z[i] = n$ And wherein n divided into i . Then the line s can be compressed to length line i .

Proof of such a decision does not differ from that of a solution using [prefix function](#).

Problem in online judges

List of tasks that can be solved using Z-function:

- [UVA # 455 "Periodic Strings"](#) [Difficulty: Medium]

- UVA # 11022 "**String Factoring**" [Difficulty: Medium]

Prefix function. Algorithm Knuth-Morris-Pratt

Prefix function. Definition

Given a string $s[0 \dots n - 1]$. Required to calculate it for the prefix function, ie array of numbers $\pi[0 \dots n - 1]$ defined as follows: it is a maximum length of the longest proper suffix substrings $s[0 \dots i]$ coinciding with its prefix (own suffix - hence does not coincide with the whole line). In particular, the value of $\pi[0]$ assumed to be zero.

Mathematically, the definition of the prefix function can be written as follows:

$$\pi[i] = \max_{k=0 \dots i} \{ k : s[0 \dots k - 1] = s[i - k + 1 \dots i] \}.$$

For example, the string "abcabcd" prefix function is: [0, 0, 0, 1, 2, 3, 0]. Which means:

- the row "a" no nontrivial prefix matching the suffix;
- in line "ab" no nontrivial prefix that matches the suffix;
- the row "abc" no nontrivial prefix matching the suffix;
- the row "abca" prefix length 1 coincides with the suffix;
- the row "abcab" prefix length 2 coincides with the suffix;
- the row "abcabc" prefix length 3 coincides with the suffix;
- the row "abcabcd" no nontrivial prefix that matches the suffix.

Another example - the string "aabaaab" it is equal to: [0, 1, 0, 1, 2, 2, 3].

Trivial algorithm

Immediately following the definition, it is possible to write such an algorithm for computing the prefix function:

```
vector<int> Prefix_function (string s)
{
    int n = (int) s.length();
    vector<int> Pi(n);
    for (int i = 0; i < n; ++i)
        for (int k = 0; k <= i; ++k)
            if (s.substr(0, k) == s.substr(i-k+1, k))
                pi[i] = k;
    return pi;
}
```

As you can see, it will work for $O(n^3)$, which is too slow.

Efficient algorithm

This algorithm was developed by Knuth (Knuth) and Pratt (Pratt) and independently by Morris (Morris) in 1977 (as a basic element for the search algorithm of the substring).

The first optimization

The first important point - that the value of $\pi[i + 1]$ no more than one unit exceeds the value $\pi[i]$ any i .

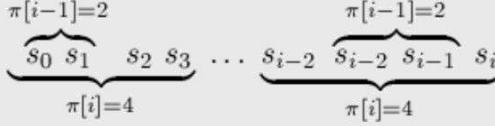
Indeed, otherwise if
and having a
at position length
suffix ending at position

Then consider this suffix ending
- Removing from it the last character, we obtain
and having a length
That better he came to

$$i+1 \qquad \pi[i+1] > \pi[i] + 1$$

$$i \qquad \pi[i+1] \qquad \pi[i+1]-1 \qquad \pi[i]$$

contradiction. An illustration of this contradiction (in this example $\pi[i-1]$ must be equal to 3):



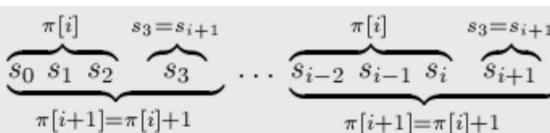
(In this diagram the upper braces denote two identical substrings of length 2, the lower braces - two identical substrings of length 4)

Thus, the transition to the next position next element prefix function could either increase by one, or do not change or decrease by any amount. This fact allows us to asymptotic behavior to reduce $O(n^2)$ as one step value could grow to a maximum of one, the total for the entire row could happen maximum n increases by one, and as a result (because the value never could be less than zero), the maximum n reductions. That will have string comparisons, ie we have reached asymptotic behavior $O(n^2)$.

The second optimization

Come on - get rid of the obvious comparisons of substrings. To do this, try to maximize the information calculated in the previous steps.

Thus, suppose we have calculated the value of the prefix function for some i . Now, if we can say with certainty that $\pi[i+1] = \pi[i] + 1$, illustrates this scheme:



(This scheme again the same curly brackets denote the same substring)

Suppose now that, on the contrary, $s[i]$ appears after $\pi[i]$. Then we need to try and try substring at length. In order to optimize I would like to go directly to a (maximum) length $j < \pi[i]$, that still holds the prefix property position ie $s[0 \dots j-1] = s[i-j+1 \dots i]$:

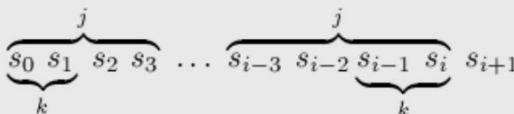
Indeed, when we find such a length, we will again be sufficient to compare the characters $s[i+1] \text{ and } s[j]$

- If they match, it can be argued that $\pi[i+1] = j + 1$. Otherwise we will have to find again minimal (the next largest) value for which the prefix property, and so on.

It may happen that such values j run out - this happens when $j = 0$. In this case, if $s[i+1] = s[0]$ Then $\pi[i+1] = 1$ Otherwise $\pi[i+1] = 0$.

Thus, the general scheme of the algorithm we already have, but remained unresolved question of finding such effective length

hs j . We pose this question formally, at the current position and length (for which the prefix property ie $s[0 \dots j-1] = s[i-j+1 \dots i]$) is required to find the greatest $k < j$ Such that in still holds the prefix property:



After such a detailed description is practically begs that this value is nothing, as the value of the prefix function, which has been calculated previously subtraction unit appears because $\pi[i]$

0-indexed rows). Thus, finding these lengths k we can for $O(1)$ each.

The final algorithm

So finally we constructed an algorithm which does not contain explicit strings and performs comparisons $O(n)$ action. We present here a summary chart:

- Read the values of the prefix function will in turn from to (Meaning just assign zero).
- For calculation of the current value we led a variable Indicating the current length of treated sample. Initially .

$\pi[i]$ $i = 1 \ i = n - 1$ $\pi[0]$

$$\pi[i] \quad j = \pi[i - 1] \quad j$$

an

- Test sample length, which compare characters $s[j]$ and $s[i]$. If they match - is

believe $\pi[i] = j + 1$ and proceed to the next index $i + 1$. If the characters are different, the decrease length j Setting it equal $\pi[j - 1]$ And repeat this step of the algorithm from the beginning.

- When we reached the length n and did not find a match, then the process stops busting samples and believe $\pi[i] = 0$ and proceed to the next index $i + 1$.

Implementation

Algorithm eventually turned in a surprisingly simple and concise:

```
vector<int> Prefix_function (string s)
{
    int n = (int) s.length();
    vector<int> Pi (n);
    for (int i = 1; i < n; ++ i)
        if (s[i] == s[pi[i-1]])
            pi[i] = pi[i-1];
    return pi;
}
```

As you can see, this algorithm is **online** algorithm, i.e. it processes the data in the course of income - can, for example, to read the string one character at once and handle this character, finding the answer to the next position. The algorithm requires storage of the string itself and the previous calculated values prefix function, however, is easy to see if we know beforehand the maximum value that can take a prefix function on the whole line, it is enough to store only one more than the number of the first character and values prefix function.

Applications

Search the substring. Algorithm Knuth-Morris-Pratt

This problem is a classical application of prefix function (and, indeed, it has been opened, and in this regard). Given a text t and line s Required to find and display the positions of all occurrences of the string s in text t .

For convenience through n length of the string s And through m - Length of the text t .

Form the string $s + \# + t$ Where the symbol $\#$ - a separator that should not anywhere else meet. Count for this string prefix function. Now consider its value, but the first

$n + 1$ (Which, as seen, to include row s and separator). By definition, the value of $\pi[i]$ shows of the longest length of the substring, ending at position i and coinciding with the prefix. But in this case - in fact the length of the largest block matching with the string s and ending at position

i . More than n , This length can not be - at the expense of the separator. But equality (Where It is achieved), means that the position i occurrence of the string ends with the title s (Just do not forget that all positions are measured in line glued $s + \# + t$).

Thus, if at any position i turned $\pi[i] = n$ The gap

$i - (n + 1) - n + 1 = i - 2n$ line t begins the next occurrence of the string s in row t .

As mentioned in the description of an algorithm to compute the prefix function, if it is known that the values of the prefix function will not exceed a certain value, it is enough to store the entire string and not the prefix function, but only the beginning. In our case, this means that you need to store in memory and only a string value $s + \#$ prefix functions on it, and then read one character string t and recalculate the current value of the prefix function.

Thus, the algorithm Knuth-Morris-Pratt solves this problem for $O(n + m)$ time and $O(n)$ memory.

Counting the number of occurrences of each prefix

Here we consider two problems at once. Given a string s length n . In the first embodiment is required for each prefix to count how many times it occurs in the very same line s . The second problem is given embodiment other string t And is required for each prefix $s[0 \dots i]$ count how many times it occurs in t .

We solve the first problem first. Consider in any position i meaning of the prefix function in it. By definition, it means that the position i occurrence of the string ends with the prefix s length $\pi[i]$ And

no greater prefix end at position can not. At the same time, in positions could terminate and occurrence of prefixes smaller lengths (and, obviously, not necessarily length i). However, as easy to see that we have come to the same question, to which we have already answered when considering algorithm prefix computation functions for a given length j . I must say, what of the longest of its own suffix coincides with its prefix. We have already seen that the answer to this question $\pi[j - 1]$. But then in this task, if the position is terminated occurrence of a string of length $\pi[i]$. Coinciding with the prefix, as in occurrence of a substring ends length $\pi[\pi[i] - 1]$. Coinciding with the prefix, and it applies to those same arguments, so also ends, and the occurrence of length $\pi[\pi[\pi[i] - 1] - 1]$ etc. (until the index becomes zero). Thus, for calculating the response, we need to perform such a cycle:

```
vector<int> Ans(n + 1); for (int i = 0; i < n; ++i)
    Ans[pi[i]]++; for (int i = n - 1; i > 0; --i)
    ans[pi[i - 1]] += Ans[i];
```

Here, for each value of the prefix function, first count how many times he had met in the array and then considered a somewhat dynamic, if we know that the prefix length occurs exactly once, what exactly this number must be added to the number of occurrences of its own suffix length, coinciding with its prefix; then have this suffix (of course, less than i length) executed "bouncing"

$\pi[]$
 $ans[i]$

This amounts to their suffix, etc.

Now consider the second problem. We apply a standard trick: assign to a row s line through the separator, ie is, we get the string $# + t$. And calculate it for the prefix function. The only difference from the first objective will that it is only necessary to consider those values of the prefix functions which relate to the line t all for $i \geq n + 1$.

Number of distinct substrings in a string

Given a string s length n . Requires her to count the number of distinct substrings.

We solve this problem iteratively. Namely, learn, knowing the current number of different substrings recalculate this amount, when added to the end of one character.

So let - the current number of different substrings k . And we append symbol c . Obviously, the result could be some new substring ending in this new symbol c . Namely, added as new ones substring ending in symbol c and not previously encountered.

Take line $t = s + c$ and invert it (write the characters in reverse order). Our task - count how many rows t such prefixes that are not found anywhere else in it. But if we think string prefix function and find its maximum value π_{\max} . Then obviously occurs in a row (not beginning) of its length prefix π_{\max} . But not greater length. Clearly, a shorter length prefixes certainly meet it.

So, we have found that the number of new substrings that appear when append symbol c equal $s.length() + 1 - \pi_{\max}$.

Thus, for each symbol is appended for we can count the number of different substrings. Thus, for $O(n^2)$ we can find a number of different substrings for any given line.

It is worth noting that you can completely analogous to recalculate the number of different substrings and append a character in the beginning, as well as removing the character from the end or the beginning.

Row compression

Given a string s length n . Required to find the shortest its "compressed" representation, ie find the following line t shortest length that s can be represented as a concatenation of one or more copies of t .

It is clear that the problem is in finding the length of the search string t . Knowing the length of the answer to the problem would be, for example, the prefix string s this length.

Count on row s prefix function. Consider its last value, ie $\pi[n - 1]$. And introduce designation $k = n - \pi[n - 1]$. We show that if n divisible by k then it will be a long answer, otherwise efficient compression does not exist, and the answer is.

Indeed, let n divided into k . Then, the string can be represented as a series of blocks of length, and, k by definition, the prefix function, prefix length $n - k$ will coincide with its suffix. k really fit the answer.

We show that this response is optimal. Indeed, otherwise, if there was a minimal, then the prefix and function at the end would be more than $n - k$ le a contradiction.

Suppose now that n not divisible by. We show that this implies that the length of the response is n . We prove by contradiction
- Assume that the answer exists and is of length p (p divider n). Note that the prefix function must necessarily be more $n - p$
this suffix should partially cover the first block. Now consider the second block row; because coincides with the prefix suffix and prefix
and suffix cover this unit, and
their displacement relative to each other does not divide the length of the block
(as would otherwise
block coincide. But then the line consists of the same character here
is, so we arrive at a contradiction.

k shared n all characters
 $k = 1$ And the answer must exist, ie

Building a vending machine prefix function

We return to have been used repeatedly receive the concatenation of two strings with a separator, ie data
strings and calculation functions for the prefix string $\# + t$. Obviously, since character $\#$
separator, the value of the prefix function will never exceed $s.length()$. It follows that, as
mentioned in the description of an algorithm to compute the prefix function, sufficient to store only string values and $\#$
prefix function for her, and for all subsequent characters prefix function to compute on the fly:

$s_0 s_1 \dots s_{n-1} \# t_0 t_1 \dots t_{m-1}$
need to save need not to save

Indeed, in this situation, knowing the next character $c \in t$ and the value of the prefix function in the
previous item, you can calculate a new value-prefix function does is not using all the previous
characters in the string t prefix and values in their function.

In other words, we can construct **automatic**: State it will be the current value of the prefix function, the
transitions from one state to another will be effected under the symbol:

$s_0 s_1 \dots s_{n-1} \# \underbrace{\dots}_{\pi[i-1]} \Rightarrow s_0 s_1 \dots s_{n-1} \# \underbrace{\dots}_{\pi[i-1]} + t_i \Rightarrow s_0 s_1 \dots s_{n-1} \# \dots \underbrace{t_i}_{\pi[i]}$

Thus, even without a line t We can construct a pre-transition table $(\text{old-}\pi, c) \rightarrow \text{new-}\pi$ using the
same prefix computation algorithm functions:

```
string s; // input string
const int alphabet = 256; // cardinality of the alphabet characters, usually less

s += '#';
int n = (int) s.length();
vector<int> Pi = prefix_function(s);
vector<vector<int>> Aut(n, vector<int>(alphabet));
for (int i = 0; i < n; ++i)
    for (char c = 0; c < alphabet; ++c)
        if (c == s[i]) ++i;
        while (j > 0 && c != s[j])
            j = pi[j - 1];
        if (c == s[j]) ++j;
        aut[i][c] = j;
    }
```

However, in this case, the algorithm will work for $O(n^2k)$ (- Cardinality of the alphabet). But note that
instead of the internal cycle j Which gradually shortens response, we can use already
calculated part of the table: moving from values j to the value $\pi[j - 1]$. We actually say that the transition
from state (j, c) lead to the same state as transition $(\pi[j - 1], c)$ accurately counted (ie
 $\pi[j - 1] < j$)

```
string s; // input string
const int alphabet = 256; // cardinality of the alphabet characters, usually less

s += '#';
```

```
int n = (int) s.length(); vector  
<int> Pi = prefix_function (s);  
vector <vector <int>> Aut (n, vector <int>(alphabet));
```

```

for (int i = 0; i <n; ++ i)
    for (char c = 0; c <alphabet; +
        + C) if (i> 0 && C! =
        S[i])
            aut[i] [c] = Aut[pi[i-1]] [c];
        else
            aut[i] [c] = I + (c == s[i]);

```

The result was a very simple implementation of the construction of the machine working for $O(nk)$.

Can be useful when a machine? To begin with we recall that we consider the prefix function for strings $s + \# + t$. And its value is usually used for the sole purpose to find all occurrences of the string s in row t .

Therefore, the most obvious benefit of constructing such a machine - **acceleration calculation prefix function** row . Constructing a row machine we no longer need $s + \#$ no string s Nor value-prefix function in it, and do not need no calculations - all transitions (ie how will change the prefix function) predposchitany already in the table.

But there is a second, less obvious use. This is the case when the line t is a giant line, built under a rule. This can be eg Gray Line or line formed by recursive combination of several short lines, submitted for entry.

Suppose for definiteness that we solve **such a task**: Given a room . Required to count the number of occurrences of the length $n \leq 10^5$ string in Gray line, and a line with $k \leq 10^5$ Gray-th row. Recall line Gray defined as follows:

In such cases, even just building line will be impossible due to its astronomical length (eg, - Gray th row has length). Nevertheless, we can calculate the value of the prefix function at the end this line, knowing the value of the prefix function, which was before the line.

So, apart from the machine also calculate such quantities: - The value of the machine achieved after "feeding" him a line if before this machine was in a state $G[i][j]$. The second value - The number of occurrences of g_i the string s in line if g_i "feeding" this line j . g_i machine is in state $K[i][j]$ $[j]$. In fact, $K[i][j]$ - The number of times that the automatic take a value $s.length()$ for time "feeding" lines. It is clear that the answer to the problem would be the value $K[k][0]$.

How to calculate these values? First, the baseline values are. And all subsequent values can be calculated by using the previous values and automatic. So, to calculate

plus the first character of the these values for some, we remember that the line g_i comprises g_{i-1} alphabet plus again g_{i-1} . Then after "feeding" the first piece (g_{i-1}) Machine to a state $G[i-1][j]$ Then after "feeding" symbol char_i it will enter:

$\text{mid} = \text{aut}[G[i-1][j]][\text{char}_i]$

After this machine "fed" the last piece, ie g_{i-1} :

easily be considered as the sum of the amounts for the three Quantities $K[i][j]$ pieces: a string g_{i-1} Symbol char_i And again line g_{i-1} :

So we have solved the problem for strings Gray can similarly solve a class of such problems. For example, exactly the same method, we solve **the next challenge**: Given a string S , And samples, each of which is defined as follows: a string of normal characters, which can occur among the recursive inserting other

in the form of rows . Which means that in this place should be inserted Example of such a scheme:

Guaranteed that this description does not contain cyclic dependencies. Constraints are such that if explicitly disclose recursion and find strings t_i , Their lengths can reach about 100^{100} .

Required to find the number of occurrences of the string s in each row t_i .

The problem is solved in the same building machine prefix function, then it is necessary to calculate and add to it the transitions for whole rows t_i . In general, it's just a more general case in comparison with the problem of lines Gray.

Problem in online judges

List of tasks that can be solved using the prefix function:

- UVA # 455 "**Periodic Strings**" [Difficulty: Medium]
- UVA # 11022 "**String Factoring**" [Difficulty: Medium]
- UVA # 11452 "**Dancing the Cheeky-Cheeky**" [Difficulty: Medium]
- SGU # 284 "**Grammar**" [Difficulty: high]

Hashing algorithms in problems on the line

Hashing algorithms lines help to solve a lot of problems. But they have a big disadvantage: that most often they are not 100% us, as there are a number of lines which hashes match. Another thing is that in most applications it can be ignored, since the probability of matching hash is still very small.

Definition and calculation of hash

One of the best ways to determine a hash function from a string S as follows:

```
h (S) = S [0] + S [1] * P + S [2] * P ^ 2 + S [3] * P ^ 3 + ... + S [N] * P ^ N
```

where P - some number.

Reasonable to choose for P prime, roughly equal to the number of characters in the input alphabet. For example, if predpolayutsya string consisting only of letters small, a good choice is P = 31. When the letters can be uppercase, and small, that may for example be P = 53.

All pieces of code in this article will use P = 31.

Hash value itself should preferably be stored in the largest numeric type - int64, aka long long. Obviously, when the string length is about 20 characters will be an overflow value. The key point - that we do not pay attention to these overflow, as if taking a hash modulo 2^{64} .

Example of calculating the hash, if allowed only small letters:

```
const int p = 31;
long long hash = 0, p_pow = 1;
for (size_t i = 0; i <s.length (); ++ i)
{
    // it is desirable to take 'a' from the code letters
    // add the unit to have a line like 'aaaaaa' hash was nonzero hash +
    = (s [i] - 'a' + 1) * p_pow;
    p_pow *= p;
}
```

In most applications, it makes sense to calculate all the necessary power P in any array.

Example task. Search for duplicate rows

Already we are able to effectively solve this problem. Given a list of strings S [1 .. N], each no longer than M characters. Suppose you want to find all duplicate rows and divide them into groups so that each group had only the same line.

Standard sorting lines we would have an algorithm with complexity O (NM log N), while using hashes we get O (N M + N log N).

Algorithm. Calculate the hash of each row, and sort the words in this hash.

```
vector <string> s (n);
// Lines ... reading ...

// consider all degree p, say, to 10,000 - the maximum length of rows
const int p = 31;
vector <long long> p_pow
(10000); p_pow [0] = 1;
for (size_t i = 1; i <p_pow.size ();
     + + i) p_pow [i] = p_pow [i-
     1] * p;

// believe hashes from all rows
// in the array store the hash value and the number of strings in the array s
vector <pair <long long, int>> hashes (n);
for (int i = 0; i <n; + + i)
{
```

```
long long hash = 0;
```

```

        for (size_t j = 0; j <s [i]. length (); ++ j)
            hash += (s [i] [j] - 'a' + 1) * p_pow
            [j]; hashes [i] = make_pair (hash, i);
    }

    // Sort by hashes
    sort (hashes.begin (), hashes.end ());

    // Output response
    for (int i = 0, group = 0; i <n; ++ i)
    {
        if (i == 0 || hashes [i]. first! = hashes [i-1].
            first) cout << "\nGroup" << ++ group <<
            ":";
        cout << ' ' << hashes [i]. second;
    }
}

```

Hash substring and its fast computation

Suppose we are given a string S, and given indices I and J. required to find a hash of the substring S [I.. J]. By definition:

$$H [I.. J] = S [I] + S [I + 1] * P + S [I + 2] * P^2 + \dots + S [J] * P^{(J-I)}$$

where:

$$\begin{aligned} H [I.. J] * P [I] &= S [I] * P [I] + \dots + S [J] * P [J], \\ H [I.. J] * P [I] &= H [0 .. J] - H [0 .. I-1] \end{aligned}$$

The resulting property is very important.

Indeed, it turns out that, **knowing only the hashes of all prefixes line S, we O (1) to obtain the hash any string.**

The only problem that comes out - is that we must be able to divide by P [I]. In fact, it is not so simply. Since we compute hash modulo 2^{64} , the divide by P [I], we have to find the inverse element thereto in the field (e.g., using [The extended Euclidean algorithm](#)) And perform a multiplication by the inverse of this.

However, there's an easier way. In most cases, **instead of sharing hashes raising at P, we can reverse these multiply their power.**

Suppose given two hash: one multiplied by P [I], and the other - on the P [J]. If I < J, then multiply Stationery hash P [JI], or else we multiply the second hash P [IJ]. Now we are led to a degree of hashes, and can compare them quietly.

For example, the code that calculates hashes of all prefixes and then O (1) compares two strings:

```

string s; int i1, i2, len; // Input

// consider all degrees
p const int p = 31;
vector <long long> p_pow (s.length ());
p_pow [0] = 1;
for (size_t i = 1; i <p_pow.size ();
     + + i) p_pow [i] = p_pow [i-
     1] * p;

// believe hashes from all prefixes
vector <long long> h (s.length ());
for (size_t i = 0; i <s.length ();
+ + i)
{
    h [i] = (s [i] - 'a' + 1) * p_pow
    [i]; if (i) h [i] += h [i-1];
}

// obtain hashes two substrings
long long h1 = h [i1 + len -
1]; if (i1) h1 -= h [i1 -
1];

```

```
1]; long long h2 = h [i2 +  
len-1]; if (i2) h2 -= h  
[i2-1];
```

```

// Compare them
if (i1 < i2 && h1 * p_pow [i2-i1] == h2 || 
    i1 > i2 && h1 == h2 * p_pow [i1-i2])
    cout << "equal";
else
    cout << "different";

```

Application hashing

Here are some typical uses hashing:

- Rabin-Karp algorithm for the substring search for O (N)
- Determination of the number of distinct substrings in O ($N^2 \log N$) (see below)
- Determination of the number of palindromes in a string

Determination of the number of distinct substrings

Suppose we are given a string S of length N, consisting of only small letters. Required to find the number of different substrings in this string.

To solve Move on queue length of the substring: L = 1 .. N.

For each L, we construct an array of hashes of substrings of length L, and let hashes to the same degree, and we can sort this array. A number of different elements in this array is added to the answer.

Implementation:

```

string s; // Input string
int n = (int) s.length ();
// consider all degrees
p const int p = 31;
vector <long long> p_pow (s.length ());
p_pow [0] = 1;
for (size_t i = 1; i <p_pow.size ();
     + + i) p_pow [i] = p_pow [i-
     1] * p;
// believe hashes from all prefixes
vector <long long> h (s.length ());
for (size_t i = 0; i <s.length ();
+ + i)
{
    h [i] = (s [i] - 'a' + 1) * p_pow
    [i]; if (i) h [i] += h [i-1];
}
int result = 0;
// Iterate through the
length of the substring for
(int l = 1; l <= n; + + l)
{
    // looking for the answer to the current length

    // get hashes for all substrings of length
    l vector <long long> hs (n-l +1);
    for (int i = 0; i <n-l +1; + + i)
    {
        long long cur_h = h [i + l-
        1]; if (i) cur_h -= h [i-
        1];
        // give all hashes to the same extent
        cur_h *= p_pow [n-i-
        1]; hs [i] = cur_h;
    }
    // Count the number of different
    hashes sort (hs.begin (), hs.end ());

```

```
hs.erase (unique (hs.begin (), hs.end (), hs.end
()) ; result += (int) hs.size ();
}

cout << result;
```

Rabin-Karp algorithm for the substring search for O (N)

This algorithm is based on hashing strings, and those who are not familiar with the topic, refer to the "Hashing algorithm in problems on the line".

Authors algorithm - Rabin (Rabin) and Karp (Karp), 1987.

Given a string S and text T, consisting of small letters. Required to find all occurrences of the string in the text S T in time $O(|S| + |T|)$.

Algorithm. Calculate the hash for a string S. Calculate the hash value for all prefix string T. Now iterate over all the strings of length $|S|$ and each is comparable with $|S|$ in $O(1)$.

Implementation

```
string s, t; // Input

// consider all degrees
p const int p = 31;
vector <long long> p_pow (max (s.length (), t.length
()), p_pow [0] = 1;
for (size_t i = 1; i <p_pow.size ();
     + + i) p_pow [i] = p_pow [i-
     1] * p;

// believe hashes from all prefixes line T
vector <long long> h (t.length ());
for (size_t i = 0; i <t.length ();
     + + i)
{
    h [i] = (t [i] - 'a' + 1) * p_pow
    [i]; if (i) h [i] += h [i-1];
}

// consider a hash of
strings S long long h_s =
0;
for (size_t i = 0; i <s.length (); + + i)
    h_s += (s [i] - 'a' + 1) * p_pow [i];

// iterate through all the strings of length  $|T| + |S|$  and
compare them for (size_t i = 0; i + s.length () - 1
<t.length (); + + i)
{
    long long cur_h = h [i + s.length () - 1]; if (i) cur_h -= h [i-1];
    // Give the hashes to the same extent and
    compare if (cur_h == h_s * p_pow [i])
        cout << i << '\n';
}
```

Expression parsing. Reverse Polish Notation

A string that represents a mathematical expression containing numbers, variables, various operations.

Required to compute its value for $O(n)$. Wherein n - Length of the string.

Here we describe an algorithm that translates this expression in the so-called **Reverse Polish notation** (Explicitly or implicitly) and in it evaluates the expression.

Reverse Polish Notation

Reverse Polish Notation - a form of writing mathematical expressions in which the operators are located after their operands.

For example, the following expression:

```
a + b * c * d + (e - f) * (g * h + i)
```

in reverse polish notation is written as follows:

```
abc * d * +ef - gh * i + *+
```

Reverse Polish Notation was developed by the Australian philosopher and expert in the theory of computing machines Charles Hamblin in the mid-1950s on the basis of Polish notation, which was proposed in 1920 by the Polish mathematician Jan Lukasiewicz.

Ease of RPN is that the expressions presented in a form very **easily calculate**, and in linear time. Head stack initially it is empty. Will move from left to right the expression in reverse polish notation; if the current element - a number or a variable, then put on top of the stack of its value; if the current element - an operation that took out the top two elements of the stack (or one if unary operation), to apply the operation and put the result back on the stack. In the end, the stack will be exactly one element - the value of the expression.

Obviously, this simple algorithm is executed for $O(n)$ ie order of the expression.

Analysis of the simplest expressions

While we consider only the simplest case: all operations **binary**, (i.e., the two arguments), and all the **left-associative** (ie, with equal priority are evaluated from left to right). Parentheses are allowed.

Head of two stacks: one for numbers, one for operations and parentheses (ie, stack characters). Initially, both the stack empty. For the second stack will support the precondition that all operations are ordered it on a strict low priority, if we move from the top of the stack. If the stack has the opening parenthesis, then ordered every unit operations, between parentheses, and the entire stack in this case is not necessarily ordered.

Will go on the line from left to right. If the current element - a figure or a variable, then put onto the stack of the number / variable. If the current element - opening bracket, then put it onto the stack. If the current element

- Closing bracket, then we push the stack and perform all operations as long as we do not learn the opening brace (that is to say, meeting a closing parenthesis, we perform all operations that are inside the parentheses). Finally, if the current element - an operation that, while on top of the stack is an operation with the same or higher priority will be to push and execute it.

After we process the entire string, the stack operations may still remain some of the operations that have not yet been calculated, and the need to perform all of them (ie, act in a similar case when we meet a closing parenthesis).

Here is the implementation of this method on the example of normal operations $+ - * / \%$:

```
bool delim (char c) {
    return c == ',';
}

bool is_op (char c) {
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '%';
}

int priority (char op) {
    return
```

```

        op == '+' || Op == '-' ? 1 :
        op == '*' || Op == '/' || Op == '%' ? 2
        : -1;
    }

void process_op (vector <int> & St, char op) {
    int r = st.back(); st.pop_back();
    int l = st.back(); st.pop_back();
    switch (op) {
        case '+': st.push_back (l + r); break;
        case '-': st.push_back (l - r); break;
        case '*': st.push_back (l * r); break;
        case '/': st.push_back (l / r); break;
        case '%': st.push_back (l% r); break;
    }
}

int calc (string & s) {
    vector <int> St;
    vector <char> Op;
    for (size_t i =0; i <s.length(); + + I)
        if (! Delim (s[i]))
            if (s[i] == '(')
                op.push_back ('(');
            else if (s[i] == ')') {
                while (op.back() != '(')
                    process_op (st, op.back()), op.
pop_back();
                op.pop_back();
            }
            else if (is_op (s[i])) {
                char curop = s[i];
                while (! Op.empty() &&
Priority(op.back()) >= Priority(s[i]))
                    process_op (st, op.back()), op.
pop_back();
                op.push_back (curop);
            }
            else {
                string operand;
                while (s[i] >= 'A' && S[i] <= 'Z' || isdigit
(s[i]))
                    operand += s[i + +];
                - I;
                if (isdigit (operand[0]))
                    st.push_back (atoi (operand.c_str()));
                else
                    st.push_back
(get_variable_val (operand));
            }
        while (! Op.empty())
            process_op (st, op.back()),
Op.pop_back(); return st.back();
}

```

Thus, we have learned to evaluate the expression for, and thus we implicitly used the $O(n)$ Reverse Polish Notation: we placed the operation in such a manner, when the time of calculating the next operation both its operands have already been computed. Slightly modifying the above algorithm can obtain an expression in reverse polish notation and explicitly.

Unary operations

Now suppose that the expression contains a unary operation (ie one argument). For example, are particularly common unary plus and minus.

One difference in this case is the need to determine whether the current operation is a unary or binary.

You may notice that before unary operation is always either another operation, or an opening parenthesis,

or nothing at all (if it stands at the beginning of the line). Before the binary operation, in contrast, is always either operand (number / variable) or a closing parenthesis. Thus, it is enough to have some flag to indicate whether the next operation to be unary or not.

More net realizable subtlety - how to distinguish between binary and unary operation when removed from the stack and

computation. You can, for example, for unary operations instead of the symbol $s[i]$ put on the stack $-s[i]$.

Priority for unary operations should be chosen such that it was more of the priorities of all binary operations.

In addition, it should be noted that the unary operations are actually right associative - if row are several unary operations, they must be processed from right to left (for a description of this case, see below; code here already accounts for right-associativity).

Implementation for binary operations $+ - * /$ and unary operations $+-$:

```
bool delim (char c) {
    return c == ' ';
}

bool is_op (char c) {
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '%';
}

int priority (char op) {
    if (op < 0)
        return op == '-+' || op == '--' ? 4;
    return
        op == '+' || op == '-' ? 1 :
        op == '*' || op == '/' || op == '%' ? 2
        : -1;
}

void process_op (vector <int> & St, char op)
{
    if (op < 0) {
        int l = st.back(); st.pop_back();
        switch (-Op) {
            case '+': St.push_back (l); break;
            case '-': St.push_back (-l); break;
        }
    }
    else {
        int r = st.back(); st.pop_back();
        int l = st.back(); st.pop_back();
        switch (op) {
            case '+': St.push_back (l + r); break;
            case '-': St.push_back (l - r); break;
            case '*': St.push_back (l * r); break;
            case '/': St.push_back (l / r); break;
            case '%': St.push_back (l % r); break;
        }
    }
}

int calc (string & s) {
    bool may_unary = true;
    vector <int> St;
    vector <char> Op;
    for (size_t i = 0; i < s.length(); +
        + i) if (! Delim (s[i]))
        if (s[i] == '(') {
            op.push_back ('(');
            may_unary = true;
        }
        else if (s[i] == ')') {
            while (op.back() != '(')
                process_op (st, op.back()), Op.
pop_back();
            op.pop_back();
            may_unary = false;
        }
        else if (is_op (s[i])) {
            char curop = s[i];
```

```

    if (may_unary && isunary (curop)) curop =
        while (! Op.empty() && (
            curop> = 0 && Priority(op.back())
            || Curop < 0 && Priority(op.back())
        )
            process_op (st, op.back(), Op.
pop_back();
            op.push_back (curop);
            may_unary = true;
        }
        else {
            string operand;
            while (s[i] >= 'A' && S[i] <= 'Z' || isdigit
(s[i]))
                operand += s[i + +];
            - i;
            st.push_back (get_val (operand));
            may_unary = false;
        }
    while (! Op.empty())
        process_op (st, op.back()),
    Op.pop_back();return st.back();
}

```

It is worth noting that in the simplest cases, for example, when a unary operations are allowed only $+$ and right-
 $_$ associativity has no role, therefore, in such situations, no complications in the scheme can be
not to impose. le cycle:

```

    while (! Op.empty() && (
        curop> = 0 && Priority(op.back())
        || Curop < 0&& Priority(op.back())
    )
        process_op (st, op.back(), op.
pop_back();

```

Can be replaced by:

```

    while (! Op.empty() && Priority(op.back())
> = Priority(curop))
        process_op (st, op.back(), op.
pop_back();

```

Right-associativity

Right-associativity of an operator means that operators with equal priority are evaluated from right to left (respectively, left associativity - when left to right).

As noted above, unary operators are usually right associative. Another example - usually exponentiation considered right-associative (really, a^b^c is generally perceived as $a^{(b^c)}$, and not $(a^b)^c$).

What are the differences should be made to the algorithm to correctly handle right-associativity? In fact, the most minimal changes are needed. The only difference will be shown only at equality priorities, and it is that the operations of equal priority that are on top of the stack should not be used to perform the current operation.

Thus, the only difference should be made to function calc:

```

int calc (string & s) {
...
    while (! Op.empty() && (
        left_assoc(curop) && Priority(op.

```

```
back()) >= Priority(curop) | ||! Left_assoc(curop) && Priority  
(op.back()) > Priority(curop)))  
...  
}
```

Suffix array

Given a string $s[0 \dots n - 1]$ length n

i Th suffix line called substring $s[i \dots n - 1] i = 0 \dots n - 1$.

Then **suffix array** line called permutation of the suffixes

$p[0 \dots n - 1], p[i] \in [0; n - 1]$, Which specifies the order of suffixes in lexicographic order sorting. In other words, you want to sort all the suffixes specified string.

For example, in row $s = abaab$ Suffix array is equal to:

Building for $O(n \log n)$

Strictly speaking, the algorithm described below will not perform sorting suffixes and **cyclic shifts** line. However, this algorithm is easy to obtain and suffix sorting algorithm: enough to attribute end of the line in any character which is certainly less than any of the characters, which may consist of

string (for example, it may be a dollar or Sharp, C language for this purpose, you can use an existing null character).

Immediately, we note that since we sort of cyclic shifts, then we will substring

consider **cyclic**: A substring $s[i \dots j]$ When $i > j$ Understood

substrings $[i \dots n - 1] + s[0 \dots j]$. In addition, all pre-indices are taken modulo the length of the line (In order to simplify the formulas, I will omit the explicit taking indices modulo).

phases. At the second

The issue before us algorithm consists of approximately $\log n$ phase ($k = 0 \dots [\log n]$) sorted cyclic substring length. At the last, 2^k second phase will be sorted substring length $2^{[\log n]} > n$. Which is equivalent to sorting cyclic shifts.

In addition to every phase permutation algorithm $p[0 \dots n - 1]$ cyclic substring indexes will support for each cyclic substring starting at the position with a length i 2^k , room $c[i]$ equivalence class, which belongs to this substring. In fact, of substrings can be identical, and the algorithm will need information about it. In addition, the numbers of the classes of equivalence $c[i]$ will give so that they retain the order information and, if a suffix is less than the other, the class number and he must get smaller. Classes will be numbered for ease from scratch. Number of equivalence classes will be stored in the variable **classes**.

We present **example**. Consider the string $aaba$. Array values $p[]$ at each stage with zero on the second as follows:

It is worth noting that an array of ambiguity possible. For example, zero-phase array can

equal: $p = (3, 1, 0, 2)$ Then what option will depend on the specific implementation of the algorithm, but all options are equally valid. At the same time, in an array $c[]$ no ambiguity could not be. We now

construct **algorithm**. Input:

```
char * s; // input string
int n; // string length
// Constants
const int maxlen = ...; // maximum line length const
int alphabet = 256; // alphabet size, <= maxlen
```

On **zero phase** we need to sort the cyclic length of the substring 1 le individual characters of the string and divide them into equivalence classes (just the same symbol must be assigned to the same equivalence class). This can be done trivially, for example, sorting, counting. For each character's count how many times he met. Then restore the information on this array.

After

$p[]$

, the passage through the array $p[]$ and comparing the character array is constructed $c[]$.

```

int p[maxlen], Cnt[maxlen], C[maxlen];
memset (cnt, 0, Alphabet * sizeof(int));
for (int i =0; i <n; ++ I)
    ++
    Cnt[s[i]];
for (int i =1; i <alphabet; ++ I)
    cnt[i] += Cnt[i-1];
for (int i =0; i <n; ++ I)
    p[- Cnt[s[i]]] = I;
c[p[0]] = 0;
int classes = 1;
for (int i =1; i <n; ++ I) {
    if (s[p[i]] != s[p[i-1]]) ++ Classes;
    c[p[i]] = Classes-1;
}

```

Further, suppose we have

fulfilled $k - 1$ Nd phase (ie, calculated values of arrays $p[]$ and $c[]$) and for her) now learn for $O(n)$ perform the following, k Th, phase. Since all phases required algorithm with time $O(n \log n)$. It will give us

consists of two substrings of

For this we note that the length of the cyclic substring 2^k length can compare with each other for $O(1)$. Using information from the previous phase - the rooms equivalence classes. Thus, for the length of the substring starting at the position i , All the information contained in a pair of numbers $(c[i], c[i + 2^k])$ (Again, we use an array $c[]$ the previous phases).

This gives us a very simple solution: sort substring length just by that pairs of numbers. It will give us the desired order, ie, array. However, the conventional sort that runs in time

$O(n \log n)$. We are not satisfied - it will give an algorithm for constructing suffix array with the $O(n \log^2 n)$ time (but this algorithm is somewhat easier to write than described below).

How quickly perform the sorting pairs? Since the elements of pairs do not exceed n , Then you can sort by count. However, to achieve the best hidden in the asymptotic constant instead of sorting couples come to just sort of numbers.

Here we use the technique, which is based on the so-called **cell sorting**:

to sort couples sort them first by the second element, and then - the first elements of (but certainly stable sorting, ie, does not violate the relative order of elements with equal).

from the previous However, a separate second elements are already in order - this order specified in the array $p[]$ phase.

Then, to arrange for a second pair of elements, you just have to each array element $p[]$ take $\frac{2^{k-1}}{2^{k-1}}$ this will give us the sort order for the second element pairs (after $p[]$ gives ordering substring length $\frac{2^{k-1}}{2^{k-1}}$ And the transition to the line twice the length of the substring these become their second halves, so from position of the second half of the first half length is subtracted).

Thus, with just a subtraction of elements of the array $p[]$ we produce sorting by The second element pairs. Now you have to produce a stable sort on the first elements of pairs, it is can be done in $O(n)$ using counting sort.

It remains only to count numbers $c[]$ equivalence classes, but they were easy to get, just passing by received a new permutation $p[]$ and comparing adjacent elements (again, comparing two numbers as a pair).

We present **implementation** Algorithm execution of all the phases other than zero. Introduced additional temporary arrays pn and cn (pn - Contains a permutation in the sort order for the second element pairs cn - New numbers of equivalence classes).

```

int pn[maxlen], Cn[maxlen];
for (int h =0; (1<< H)<N; + +
H) {
    for (int i =0; i <n; + + I) {
        pn[i] = P[i] - (1<<
        H);
        if (pn[i] < 0) pn[i] += N;
    }
}

```

```
memset (cnt, 0, Classes * sizeof(int));
for (int i =0; i <n; + + I)
    + + Cnt[c[pn[i]]];
for (int i =1; i <classes; +
    + I) cnt[i] + =
    Cnt[i-1];
for (int i = n-1; i> =0; - I)
```

```

    p[- Cnt[c[pn[i]]]] = Pn[i];
    cn[p[0]] = 0;
    classes = 1;
    for (int i = 1; i <n; + +
    I) {
        int mid1 = (p[i] + (1<< H)) % N, mid2 = (p[i-1] + (1<< H)) % N;
        if (c[p[i]] != C[p[i-1]] || C[mid1] != C[mid2])
            ++
            Classes;
        cn[p[i]] = Classes-1;
    }
    memcpy (c, cn, n * sizeof(int));
}

```

memory. However, if we also take into account the size of the alphabet,

$O(n \log n)$ time and $O(\text{And the size of } k)$

While the work becomes $O((n+k) \log n)$ memory - $O(n+k)$.

Applications

Finding the smallest cyclic shift line

The above algorithm produces a sort of cyclic shifts (if the line does not ascribe to the dollar), but because $p[0]$ give the desired position of the smallest cyclic shift. Hours - $O(n \log n)$.

Search the substring

Suppose you want to search for a string in the text t online (i.e., advance the string s unknown).

We construct the suffix array for text t for $O(|t| \log |t|)$. Now substring s . We seek the following way: we note that the desired entry must be prefixed with a suffix t . Since suffixes we ordered (it gives us a suffix array), the substring s . You can search with a binary search on suffix string. Comparison of the current suffix and substring s inside a binary search can be made trivial for $O(|p|)$. Then the asymptotic behavior of the substring search in the text becomes $O(|p| \log |t|)$.

Comparison of the two substrings

Required for a given line s performing some of its preprocessing learned in responding $O(1)$ to compare two arbitrary substring (ie, checking that the first substring is equal to / less than / greater than the second).

We construct the suffix array for, while maintaining intermediate results: we need arrays $c[]$ of each phase. Therefore, the memory required is also $O(|s| \log |s|)$.

Using this information, we can compare any two for a substring of length equal to a power of two: it is enough to compare the numbers of equivalence classes of the corresponding phase. Now it is necessary to generalize this method by string of arbitrary length.

Suppose now entered another request compare two substrings of length with origins in the indices i and j and. Find the greatest length of the unit shall be placed inside a substring length, ie most k such that $2^k \leq l$. Then the comparison of the two substrings can be replaced by comparing two pairs of overlapping blocks of length 2^k : First it is necessary to compare two blocks, starting at positions and, while equality - compare two block ending positions $i + l - 1$ and $j + l - 1$:

Thus, the implementation is roughly this (it is assumed here that the calling procedure itself generates as do it in constant time is not so

k easy (apparently, most likely - predposchetom), but

Anyway, it has no relation to the use of suffix array):

```

int compare (int i, int j, int l, int k) {

```

```
pair <int,int> A = Make_pair (c[k] [i], C[k] [i + l-(1<<(k-1))]);
pair <int,int> B = Make_pair (c[k] [j], C[k] [j + l-(1<<(k-1))]);
    return a == b? 0: A <b? -1: 1;
}
```

The greatest common prefix of two substrings: method with additional memory

Required for a given line s performing some of its preprocessing learned in responding longest common prefix (longest common prefix, lcp) for two arbitrary positions with suffixes i and j .

The method described herein requires an additional memory; another method using a linear memory, but Non-constant response time is described in the next section.

We construct the suffix array for, while maintaining intermediate results: we need arrays $c[]$ of each phase. Therefore, the memory required is also $O(|s| \log |s|)$.

Suppose now entered another request: a pair of indices i and j . We use the fact that we can compare any two substrings of length, which is a power of two. To do this, we will sort out a power of two (high to low), and the current degree check if the substring of this length are the same, then the answer has to add this power of two, and the greatest common prefix continue to look to the right of the same parts, ie to i and j we must add the current power of two.

Implementation:

```
int lcp (int i, int j) {
    int ans = 0;
    for (int k = log_n; k >= 0; - K)
        if (c[k][i] == c[k][j])
            { ans += 1<< K;
              i += 1<<
              K; j += 1<< K;
            }
    return ans;
}
```

Here in \log_n denotes a constant equal to the logarithm n base 2, rounded down.

The greatest common prefix of two substrings: method without additional memory. The greatest common prefix of two adjacent suffixes

Required for a given line s performing some of its preprocessing learn responding longest common prefix (longest common prefix, lcp) for two arbitrary positions with suffixes i and j .

Unlike the previous method described herein will perform preprocessing time line for the memory. The result will be an array $O(n \log n)$ of preprocessing (which itself

is an important source of information on line, and therefore be used for other tasks). The answers to the inquiry will be made as a result of the query RMQ (minimum interval, range minimum query) in this array, so when different implementations can be obtained as the logarithmic and constant time operation.

The basis for this algorithm is the following idea: find some way most common prefixes for each **neighboring pairs in sorted order of suffixes**. In other words, we construct an array, where is the greatest common prefix and suffix, this array will give us the answer to any two adjacent rows of suffixes. Then the answer for any two suffixes, not necessarily adjacent, you can get through this array. In fact, even received a request from some

numbers and suffixes . Find these codes in the suffix array, ie let $p[i] p[i + 1]$ an - Their position in the d array

(We order them, ie let). Then the response to this request will be a minimum in the array Taken can be replaced with

on the interval . In fact, the transition from the suffix the suffix whole chain transitions, starting with the suffix and ending in the suffix But comprising all intermediate suffixes are in the sort order between them.

Thus, if we have such an array , The answer to any request longest common prefix reduces to request **minimum on the interval** array . This classical problem on the minimum interval (range minimum query, RMQ) has multiple solutions with different asymptotic behavior described [here](#).

So, our main task - **construction** this array . We will build it in the course of the algorithm suffix array construction: at each iteration, the current will build an array for cyclic substring of the current length.

After a goalless array iteration Obviously, should be zero.

Suppose now that we have completed iteration, received from her array. As we remember, in the algorithm

$k - 1$

lcp

And should be in the current Th k

iteration recalculate the array, it received a new meaning

constructing suffix array cyclic substring length broke in half into two substrings of length

2^{k-1} ; use this same technique to create the pattern and lcp.

So, even for the current iteration algorithm for computing the suffix array has done his job, found a new meaning permutation substrings. We will now go through this array and see a pair of adjacent substrings:

$p[i] \text{ and } p[i+1]$. Each substring breaking in half, we have two different situations:

1) The first half of substrings at the positions i and $i+1$ are different, and 2) the first match halves (Remember, this comparison can be easily produced by simply comparing the numbers of classes

$c[i]$ the previous

iteration). Let us consider each of these cases separately.

1) First half substrings differ. Note that while in the previous step, these must first halves were nearby. In fact, the equivalence classes could not disappear (and can only

appear), so all the different length of the substring 2^{k-1} give (as the first halves) of the current iteration different substrings of length 2^{k-1} in the same order. Thus, for determining $\text{lcp}[i]$ in it case you should just take the appropriate value from the array lcp' .

2) First half match. Then the second half could be the same or different and; while if they are different, then they do not necessarily have to be adjacent in the previous iteration. Therefore, in this case, there is no easy way to determine. To define it, we must do the same, as we $\text{lcp}[i]$

then going to calculate the greatest common prefix of any two suffixes: minimum necessary to execute the query (RMQ) on the corresponding interval array lcp' .

We estimate the **asymptotics** this algorithm. As we have seen in the analysis of these two cases, only the second case gives increase in the number of equivalence classes. In other words, we can say that every new equivalence class appears with a single query RMQ. Since all equivalence classes can

to be, then we should at least look for the asymptotics $O(\log n)$. And for this we need to use what is already-what data structure to the lowest point on the segment; this data structure will have to be rebuilt on each iteration (which only $O(\log n)$). A good option is a data structure **Segment tree**: Its can be constructed $O(n)$ And then perform searches for $O(\log n)$ That just gives us the final asymptotics $O(n \log n)$.

Implementation:

```
int lcp[maxlen], Lcpn[maxlen], Lpos[maxlen], Rpos[maxlen];
memset (lcp, 0, sizeof lcp);
for (int h=0; (1<< H)<N; ++ H) { for (int i=0; i<n; ++ I)
    rpos[c[p[i]]] = I; for (int i=n-1; i>=0; - I)
    lpos[c[p[i]]] = I;
    All ... build action suffix. array except the last
line (memcpy) ...
    rmq_build (lcp, n-1);
    for (int i=0; i<n-1; ++ I) {
        int a = p[i], b = p[i+1];
        if (c[a] != c[b])
            lcpn[i] = Lcp[rpos[c[a]]];
        else {
            int aa = (a + (1<< H)) % N, bb = (b + (1<< H)) % N;
            lcpn[i] = (1<< H) + Rmq (lpos[c[aa]], Rpos[c[bb]]-1);
            lcpn[i] = Min (n, lcpn[i]);
        }
    }
    memcpy (lcp, lcpn, (n-1) * sizeof(int));
    memcpy (c, cn, n * sizeof(int));
}
```

Here, in addition to an

array lcp introduced a temporary array lcpn with its new value. Also supported array pos Which for each substring keeps its position in the permutation p . Function rmq_build - a function that builds a data structure for a minimum of the array, the first argument passed to the size of its second argument. Function rmq returns for at least a segment: the first argument by the second, inclusive.

Most of the algorithm for constructing suffix array only had to make up the array, since during the calculation lcp we need the old values of the array.

It is worth noting that our implementation is a common prefix length for **cyclic substring**, Whereas in practice often is necessary length of the common prefix for the suffixes in their conventional sense. In

this case, you simply limit values lcp at the end of the algorithm:

```
for (int i = 0; i <n-1; + + I)
    lcp[i] = Min (lcp[i], Min (n-p[i], N-p[i +1]));
```

For **any** two suffixes length of their longest common prefix can now be found at least in the relevant segment of the array lcp :

```
for (int i = 0; i <n; +
    + I) pos[p[i]]
    = I;
rmq_build (lcp, n-1);

Received a request ... (i, j) on finding LCP ...
int result = rmq (min(i, j), Max(i, j)-1);
```

The number of different substrings

Perform **preprocessing** Described in the previous section for $O(n \log n)$ time and $O(n)$ memory we for each pair of adjacent in the sort order suffixes find the length of their longest common prefix.

We now find this information on the number of distinct substrings in a string.

To do this, we will consider what new substring starting at position $p[0]$ position $p[1]$ And t etc. In fact, we take another in the sort order suffix, and look what it prefixes give new substring. Thus, we obviously do not lose sight of any of the substrings.

Using the fact that suffixes have already sorted, it is easy to understand that the current suffix $p[i]$ give in as new substring all their prefixes, except match the prefix suffix $p[i - 1]$. ie all its prefixes, except $\text{lcp}[i - 1]$. First, give the new substring. Since the length of the current suffix is $n - p[i]$, We finally obtain that the current suffix $p[i]$ gives $n - p[i] - \text{lcp}[i - 1]$ new substrings. Summing it all suffixes (for the very first, just $n - p[0]$), We obtain **answer** Problem:

Problem in online judges

Tasks that can be solved using suffix array:

- UVA # 10679 "I Love Strings!!!" [Difficulty: Medium]

Suffix automaton

Suffix automaton (Or directed acyclic graph of words) - it powerful data structure that allows to solve many problems of string.

For example, using the suffix automaton can search for all occurrences of one string to another, or to count the number of different substrings - both problems it can solve in linear time.

On an intuitive level, suffix automaton can be understood as concise information about **all substring** this string.

Impressive fact is suffix automaton that contains all

information in a compressed form so that the length of the string, it requires only $O(n)$ memory. Moreover, it may also be constructed during $O(n)$ (If we consider the size of the alphabet constant; otherwise - during $O(n \log k)$).

Historically, first linearity size suffix automaton was opened in 1983 of Blumer et al, and 1985

- 1986. were presented the first algorithms for its construction in linear time (Crochemore, Blumer et al.) For more details - see the list of references at the end of the article.

English suffix automaton is called "suffix automaton" (in the plural - "suffix automata"), and directed acyclic graph of words - "directed acyclic word graph" (or simply "DAWG").

Definition of suffix automaton

Definition. **Suffix automaton** for this row S called a minimal deterministic finite automaton that accepts all suffixes line S .

Decipher this definition.

- Suffix automaton is a directed acyclic graph in which the vertices are called **states** And arcs of the graph - it **transitions** between these states.
- One of the states t_0 is called **initial state** And it must be the source of the graph (ie reachable from it all other states).
- Each **transition** in the machine - this arc, labeled by some symbol. All transitions originating from a state are required to have **different** tags. (On the other hand, the state transitions need not be for some symbols.)
- One or more states labeled as **terminal states**. If we pass from initial state along any path to a terminal state, and write with the labels of all arcs traversed, you get a string that is bound to be one of the suffix string S .
- Suffix automaton contains the minimum number of vertices among all machines that satisfy the above conditions. (Minimum number of transitions is not required as subject to a minimum number of states in the machine can not be "extra" ways - otherwise it would violate the previous property.)

The simplest properties of suffix automaton

The simplest, yet most important property of suffix automaton is that it contains information about all the substring S . Namely **either way** from the initial state, if we write tags arcs along this path, there is necessarily **substring** line S .

Conversely, any substring S corresponds to a path starting in the initial state t_0 .

To simplify the explanation, we will say that the substring **corresponds** the path from the initial state, tags along which form this string. Conversely, we say that any path **corresponds** string that the label form of arcs.

Each state machine suffix is one or more paths from the initial state. We say that a state **corresponds** set of rows corresponding to all these paths.

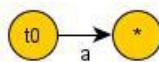
Examples constructed suffix automata

We give examples of suffix automata constructed for a few lines.

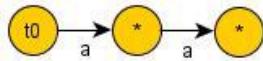
The initial state will be denoted here by t_0 And terminal conditions - asterisk. For a line $s = " "$:

t_0

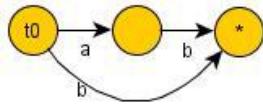
For a line $s = "a"$:



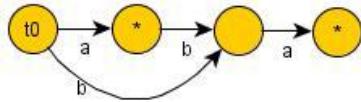
For a line $s = "aa"$:



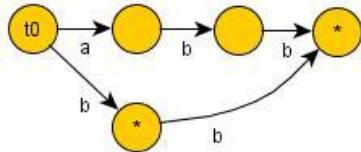
For a line $s = "ab"$:



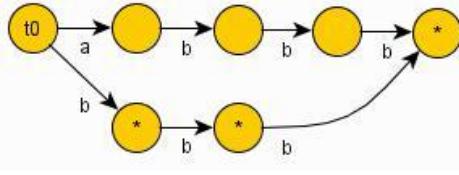
For a line $s = "aba"$:



For a line $s = "abb"$:



For a line $s = "abbb"$:



Algorithm for constructing the suffix automaton in linear time

Before you go directly to the description of the algorithm construction, it is necessary to introduce a few new concepts and prove the simple, but very important for understanding the suffix automaton lemma.

Position endings endpos Their properties and relation with suffix automaton

Consider any non-empty substring t line s . Then call **set of endings** $\text{endpos}(t)$ set of all positions in a row s in which the occurrence of the string end with t .

We will call two substrings t_1 t_2 endpos -Equivalent if their sets of endings coincide: $\text{endpos}(t_1) = \text{endpos}(t_2)$. Thus, all non-empty substring of string can be broken several **equivalence classes** according to their sets endpos .

It turns out that the suffix automaton endpos -**Equivalent substrings** corresponds to the same state. In other words, the number of states in the suffix automaton equal to the number of equivalence classes of all substrings, plus one initial state.

As each suffix automaton correspond one or more substrings having the same value endpos .

This statement we take for granted, and describe an algorithm for constructing suffix machine on that assumption - as we shall see later, all the required properties of suffix automaton except minimal, will be fulfilled. (A minimality follows from Theorem Nerode - see References).

Here are some simple but important statements about values endpos .

Lemma 1. Two nonempty substrings u and w are equivalent if and only if line u is found in row s only as a suffix string w .

The proof is almost obvious. One way: if u and w have the same position endings occurrence that u is a suffix w And is present in S Only as a suffix w . Conversely, if u is a suffix w and this occurs only as a suffix, then the values endpos by definition equal.

Lemma 2. Consider two non-empty substrings u and w ($\text{length}(u) \leq \text{length}(w)$). Then their set endpos either do not intersect or $\text{endpos}(w)$ is entirely contained in $\text{endpos}(u)$, Wherein this depends on whether or not the suffix w

Proof. Assume that a plurality of $\text{endpos}(u)$ and $\text{endpos}(w)$ have at least one common element. Then it means that the line u and w terminate at the same location, i.e. u - Suffix w . But then every occurrence of the string w contains at its end occurrence of the string u That means that its set $\text{endpos}(w)$ entirely embedded in many $\text{endpos}(u)$.

Lemma 3. Consider a class endpos -Equivalence. Sort all substrings included in this class in nonincreasing length. Then the resulting sequence each substring will be one shorter than the previous one, and still be a suffix of the previous. In other words, **substring belonging to one equivalence class are actually suffixes each other and take all kinds of different lengths in an interval $[x; y]$.**

Proof.

Fix an equivalence class. If it contains only one row, then the correctness of the lemma is obvious. Suppose now that the number of rows is more than one.

According to Lemma 1, two different endpos -Equivalent strings are always such that one is proper suffix another. Consequently, in the same class endpos -Equivalence can not be lines of equal length.

We denote w length, and through u - The shortest line in this equivalence class. According to Lemma 1, the line u is a proper suffix of. We now consider any suffix string w in length $\text{length}(u); \text{length}(w)$, And show that it is contained in the same equivalence class. In the fact, this suffix can be included in s w Only as a suffix string (as shorter suffix u includes only a suffix string w). Consequently, according to Lemma 1, this endpos -equivalent line, as required.

Suffix links

Consider a state machine string class with the same values $v \neq t_0$ endpos . As we now know as v corresponds to some longest of these lines, the all others will be suffixes. w

We also know that the first few lines of suffixes w (If we consider the suffixes in descending order their length) are in the same equivalence class, and all other suffixes (minimum blank suffix) - in some other classes. We denote t first such suffix - in it and we will carry out suffix link.

In other words, **suffix link** $\text{link}(v)$ leads to a state which corresponds **of the longest suffix** line, which is in a **different class** endpos -Equivalence.

Here we assume that the initial state t_0 corresponds to a single equivalence class (containing only an empty string), and we believe $\text{endpos}(t_0) = [-1 \dots \text{length}(s) - 1]$.

Lemma 4. Suffix links form **tree**, whose root is the initial state t_0 .

Proof. Consider an arbitrary state $v \neq t_0$ Suffix link $\text{link}(v)$ leads THEREOF state, which correspond to lines of strictly smaller length (this follows from the definition of suffix links and Lemma 3). Therefore, moving on suffix links, we will sooner or later come from the state v in the initial state t_0 , Which corresponds to an empty string.

Lemma 5. If we construct a set of all available endpos tree (Principle "Set-parent contains as a subset of all of their children"), it will be the same as the structure of the tree suffix links.

Proof.

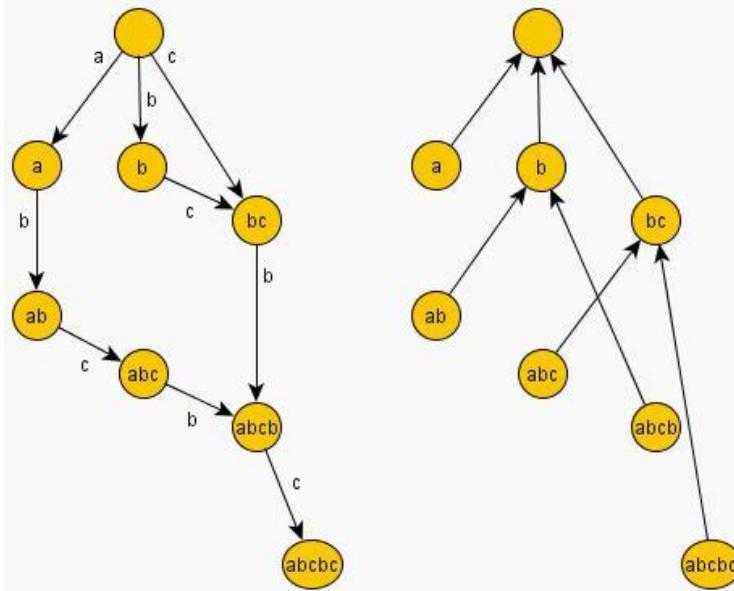
What sets of endpos You can construct a tree from Lemma 2 (that any two sets endpos either not intersect, or one contained in the other).

Now consider an arbitrary state $v \neq t_0$ and the suffix link $\text{link}(v)$ From the definition suffix links and Lemma 2:

$\text{endpos}(v) \subset \text{endpos}(\text{link}(v))$,

which together with the previous lemma proves our assertion: suffix tree reference is essentially a timber is laid sets endpos .

Here is an example tree suffix links in suffix automaton built for the line "abcabc":



Subtotal

Before proceeding to the very algorithm systematize knowledge accumulated above and introduce a couple of auxiliary notation.

- Set of substrings can be divided into equivalence classes according to their sets closure .
 - Suffix automaton consists of an initial state , As well as one of each class endpos . -Equivalence.
 - Each state corresponds to one or multiple rows. We denote Long out t_0
 - such strings v corresponds to one or multiple rows. We denote $\text{longest}(v)$ through its length. We denote $\text{shortest}(v)$ shortest of these lines, and the length through $\text{minlen}(v)$.
 - Then all strings that match this condition are various suffixes line $\text{longest}(v)$ and have all sorts of lengths in the interval $[\text{minlen}(v); \text{len}(v)]$.
 - For each state, $v \neq t_0$ defined suffix link leading to a condition that corresponds to the suffix string $\text{longest}(v)$ length $\text{minlen}(v) - 1$. Suffix links form a tree rooted in t_0 . And this tree is, in fact, is a tree inclusion relations between the sets endpos .
 - Thus, $\text{minlen}(v)$ for $v \neq t_0$ expressed by the suffix links $\text{link}(v)$ as:
- $$\text{minlen}(v) = \text{len}(\text{link}(v)) + 1.$$
- If we start from an arbitrary state v_0 and we will go on suffix links, then sooner or later we get to the initial state. In this case we get a sequence of disjoint intervals $[\text{minlen}(v_i); \text{len}(v_i)]$ That in the union will give one continuous segment.

Algorithm for constructing the suffix automaton in linear time

Proceed to the description of the algorithm. The algorithm will online will add one character string len link Rearranging appropriately the current machine.

To achieve linear memory consumption in each state, we will only store a value and a list of transitions from this state.

Initially machine consists of a single state t_0 We agree to assume zero state (the remaining states will receive numbers $1, 2, \dots$). Assign this status $\text{len} = 0$. An value link assign for convenience—1 (Indicating a link to a fake, non-existent state).

Accordingly, the whole problem now boils down to is to implement treatment uploads one symbol c at the end of the current line. We describe this process:

- Let $last$ - A state corresponding to the entire current line to add characters. (Originally $last = 0$, and after the addition of each character, we will change the value $last$.)
- Create new state cur By placing him $len(cur) = len(last) + 1$. Value $link(cur)$ while feel uncertain.
- We make such a cycle: initially we are able to $last$; if he does not go out in the letter, then add this transition in the letter to the state cur . And then go on suffix link checking again - if no transition, then attach. If at any time it happens that such a transition is already there, then stop - and denote p number of the state in which it occurred.
- If never happened, that the transition to the letter c already had, and we did come down to a fictitious state \underline{in}_1 we got to the suffix link from the initial state), we can simply assign $link(cur) = 0$ and exit.
- Assume now that we have stayed at some state p , From which there was a shift in the letter; c . We denote q the state, where does this transition is available.

Now we have two cases depending on whether $len(p) + 1 = len(q)$ or not.

If $len(p) + 1 = len(q)$, We can simply assign $link(cur) = q$ and exit.

Otherwise, all the more difficult. It is necessary to make "Cloning" state: create new state $clone$, Copy the data from the top (the suffix link transitions), for $clone$: It is necessary to q except for the value $len(clone) = len(p) + 1$.

After cloning, we spend a suffix link from cur in this condition $clone$. Also redirect of a suffix link $q \ clone$.

Finally, the last thing we need to do - is to go from state p by suffix links, and for each of the next state to check if there was a shift in the letter; c in the state q . Then forward it to the state $clone$ (And if not, then stop).

In any case, whatever finished this procedure, we end the update value $last$ assigning it cur .

If we also need to know which vertices are **terminal** And what - no, we can find all terminal nodes after constructing suffix automaton for the entire row. For this we consider the state corresponding to the entire string (which, obviously, we have stored in a variable), and we will walk in his suffix links, until we reach the initial state, and mark each state covered as a terminal. Easy to understand, thus we label states corresponding to all suffixes line s That we required.

In the next section we look at each step of the algorithm and show its **correctness**.

Here we only note that from the algorithm can be seen that the addition of one character leads to the addition of one or two states in the machine. Thus, **linearity of the number of states** obvious.

Linearity of the number of transitions, and generally linear time algorithm less well understood, and they will be proved below, after the proof of the correctness of the algorithm.

Proof of the correctness of the algorithm

- Is called the transition **solid** if. Otherwise, $len(p) + 1 < len(q)$, The transition will be called **discontinuous**.
As can be seen from the description of the algorithm, continuous and discontinuous transitions lead to different branches of the algorithm. Solid transitions are so called because, appearing for the first time, they will never change. In contrast, discontinuous transitions may change if you add new characters to the line (can change the end of the arc transition).
- To avoid ambiguity, under the line s we mean the row for which the machine was built Suffix to add the current symbol c .
- The algorithm starts with the fact that we are creating a new state cur Which will match the entire string $s + c$. Clear why we must create a new state - as together with the addition of a new character, a new equivalence class - a class lines ending in adds character c .
After creating a new state of the algorithm traverses the suffix links, starting from a position corresponding to the entire string s . And tries to add a transition on the symbol c in the state cur . Thus, we assign to each suffix string s symbol c . But add new transitions we can only if they do not conflict with existing, so as soon as we meet an existing transition on the symbol c We must stop immediately.
- The simplest case - if we have reached and fictitious state -1 Adding all the new characters along the transition c . This means that the symbol c in row s not previously met. We have successfully added all the transitions, we only have to put the suffix link status cur - She obviously must be equal 0 Because of cur in this case correspond to all suffixes line $s + c$.

• The second case - when we stumbled upon an existing transition

in automatic line $x + c$ (Where - some suffix string)

was previously added in automatic (I.e., line $x + c$ already included as a substring in a string). Since we assume that the automatic row S built correctly, the new transitions we should not add any more.

However, there is a complexity in order to maintain the state of the suffix link cur . We need to spend a suffix link to a state in which the length of the string will be just this very

$x + c$ the len for this state must be equal $len(p) + 1$. However, such a state could not exist: in this case we need to make "Splitting" state.

- Thus, one possible scenario, a continuous transition turned, i.e. $len(q) = len(p) + 1$. In this case is simple, no cleavage is not necessary to produce, and we just spend a suffix link from state cur in the state q .

• Another more sophisticated version - when a discontinuous transition, ie

that of q corresponds not only necessary to us substring $x + c$ length $len(p) + 1$ But also substring greater length. We have no choice but to make "Splitting" state q . Smash the segment rows corresponding to her two subsegments, so the first will end up just long $len(p) + 1$.

How to produce this split? We "Clone" state

Making a copy of it with

q all the transitions because we do not

parameter $len(clone) = len(p) + 1$. We copy the $clone$ of q want no way to change the path passed through. Suffix link from $clone$ we are wherever led old suffix link from, and a link from q aim in $clone$.

After cloning, we spend a suffix link from cur in $clone$ - Something for which we cloned.

The last remaining step - redirect some members of the q transitions, forwarding them on. What exactly must redirect incoming transitions? Just redirect only the transitions corresponding

all suffixes line $w + c$ and as long as we get to a fictitious state in a state other than. q

Ie we must continue to move on suffix links, starting from the top, or we reach a state of transition which leads

-1

Proof linear number of operations

First, once the reservation that we consider the size of the alphabet **constant**. If it is not, then talk about linear time work will not work: the list of transitions from one vertex to be stored in the form of a balanced tree, which allows a fast search operations on the key and the key is added. Consequently, if we denote the size of the alphabet, then the asymptotic behavior of the algorithm will be

$O(n \log k)$ at $O(n)$ memory. However, if the alphabet is small enough, it is possible, sacrificing memory avoid balanced lists, and store transitions at each vertex in an array of length (for quick search on the key) and a dynamic list (for fast failover of all available keys). Thus we reach $O(n)$ time of the algorithm, but at the cost $O(nk)$ memory consumption.

Thus, we assume a constant alphabet size, ie each search operation on the symbol transition, add transition, searching for the next move - all of these operations, we believe in working $O(1)$.

If we look at all parts of the algorithm, it contains three places linear asymptotic behavior is not obvious:

- First place - it is run on the state of the suffix links $last$ with the addition of ribs on the symbol c .
- Runner - up state transitions when cloning q in brand new condition $clone$.
- Third place - redirect transitions leading to q On $clone$.

We use the well-known fact that the size of suffix automaton (as the number of states, and the number of transitions) **linear**. (Proof of linearity in the number of states is the algorithm itself, and the proof is linear in the number of transitions, we give below, after the implementation of the algorithm.).

Then the obvious linear total asymptotics **first and second place**: After each operation here adds a new automatic transition.

It remains to estimate the asymptotic behavior of the total **in third place** - Is where we redirect the transitions leading to the q On. Denoted. This suffix string S And with each iteration of its length decreases - and, hence, the position v as suffix string S increases monotonically with each iteration. Thus, if before the first iteration of the loop corresponding row was at a depth of ($k \geq 2$) On $last$ (Assuming depth number suffix links that need to go), then the last iteration of a string v will be the second suffix link along the path from cur (Which becomes the new value $last$).

Thus, each iteration of the loop results in that the position of strings $longest(link(link(last)))$ as suffix entire current line will increase monotonically. Consequently, all this cycle could not work more n iterations **QED**.

(It is worth noting that similar arguments can be used to prove the linearity of the first place, instead of referring to the proof of the linearity of the number of states.)

Implementation of the algorithm

First we describe a data structure that will store all the information about a particular transition (*Jump List*). If necessary, you can add a flag here is terminal, as well as other the information required. List transitions we store in a standard container *map* That allows achieve total $O(n)$ memory $O(n \log k)$ the whole processing time of the line.

```
struct state {
    int len, link; map
    <char,int> Next;
};
```

Suffix machine itself will be stored in an array of these structures *state*. As proved in the next section, if *MAXLEN* the maximum possible length of a line in the program, it is enough to have a memory for the states. We also keep a variable - state corresponding whole line at the moment.

```
const int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last;
```

Let function initializes Suffix Machine (creating machine with a single initial state):

```
void sa_init() {
    sz = last = 0;
    st[0].Len = 0;
    st[0].Link = -
    1; ++Sz;
    /* / / This code is only needed if the machine is built for
       many times
different lines:
    for (int i = 0; i < MAXLEN * 2;
         + + i) st[i].
        next.clear ();
    */
}
```

Finally, we present the implementation of the basic functions - which adds another character to the end of the current line, rearranging appropriately courier:

```
void sa_extend (char c) {
    int cur = sz + +;
    st[cur].Len = st[last].Len +
    1; int p;
    for (p = last; p != -1 && ! St[p].Next.count(c); p =
        st[p].Link) st[p].Next[c] = Cur;
    if (p == -1)
        st[cur].Link = 0;
    else {
        int q = st[p].Next[c];
        if (st[p].Len + 1 == St[q].
            Len) st[cur].Link = q;
        else {
            int clone = sz + +; st[clone].
            Len = st[p].Len + 1;
            st[clone].Next = st[q].Next;
            st[clone].Link = st[q].Link;
            for (; p != -1 && St[p].Next[c] == Q; p =
                st[p].Link) st[p].Next[c] = Clone;
            st[q].Link = st[cur].Link = clone;
        }
    }
    last = cur;
}
```

As mentioned above, if the sacrifice of memory (up to $O(nk)$ where k is the size of the alphabet), you can achieve build time machine in $O(n)$ even for any k . But you have to in each state store an array of size k (for a quick search on the desired letter of the transition) and the list of transitions (for fast failover or backup all transitions).

Additional properties of suffix automaton

The number of states

The number of states in the suffix automaton constructed for the line s length n , less than $2n - 1$ (For $n \geq 3$).

This is proved by the algorithm described above (because initially machine consists of a single initial state, the first and second steps are added in exactly the same state, and each of the remaining $n - 2$ steps could be added on top of the two because of the splitting of state).

However, this estimate **easy to show, and without the knowledge of the algorithm**. Recall that the number of states

one more than the number of different sets of values endpos (As there is an initial state, standing apart from the rest.) Moreover, these plurality of endpos form a tree on the principle of "top-parent contains as a subset of all children. "Consider this tree, and transform it a bit: while in

It has an interior vertex with one son, it means that this son does not contain at least one number parent; then create a virtual vertex with endpos , Equal to this number, and gain this son to the parent. As a result, we obtain a tree in which each internal node has degree is greater than one, and the number of leaves does not exceed

n . Consequently, only such a tree is not a $2n - 1$ top. Given the initial state, endpos which may be the same endpos other state, we obtain the estimate $2n$. However, this estimate may slightly improve, noting that only on test form "aaaaa ..." happens that there is a state other than the initial, endpos which comprises all the numbers from 1 to n ; but on this test score $2n$ clearly can not be achieved and, therefore, we have improved the final evaluation to $2n - 1$ (Due to the fact that no longer allow one of the initial state - in all the other tests the initial state is no longer the same as the other vertex).

Thus we have shown that assessment independently, without knowledge of the algorithm.

It is interesting to note that this estimate is sharp, ie there **test on which it is achieved**. This test is as follows:

"abbb ..."

When the processing of the rows on each iteration, beginning with the third split state will occur, and thereby can be reached score $2n - 1$.

The number of transitions

The number of transitions in the suffix automaton constructed for the line s length n , less than $3n - 4$ (For $n \geq 3$).

We prove this.

We estimate the number of continuous transitions. Consider a spanning tree of the longest paths in the machine, starting in the state. This framework will consist of continuous ribs, and thus their number is one less than the number of states, ie does not exceed $2n - 2$.

We now estimate the number of non-continuous transitions. Consider each discontinuous transition; let the current transition - a transition from the symbol c . Deliver him into the corresponding row $u + c + w$, Where the string

u (p, q) corresponding path length from the initial state in p And w - Long way from q in any terminal state. On the one hand, all these lines $u + c + w$ for all non-continuous transitions are different (because the strings u and w only continuous transitions are formed). On the other hand, each of these rows $u + c + w$ By definition the terminal state, the entire string is suffixed s .

Since the non-empty suffixes have line s only n pieces, and then the entire string s Among these lines $u + c + w$ could not be contained (as the whole line s corresponds to the path of n solid edges), then the total number of non-continuous transitions does not exceed $n - 1$.

Adding these two estimates, we obtain the estimate $3n - 3$. However, remembering that the maximum number of states can only be achieved on the test species and it is clearly not achieved score, obtain the final estimate $3n - 4$. As required.

It is interesting to note that there is also **test which this bound is attained**:

"abbb ... bbbc"

Communication with suffix tree. Construction of suffix tree

by suffix automaton and vice versa

Prove two theorems that establish mutual relationship between suffix automaton and suffix tree.

Outset that we believe that the input string is that each has its own suffix in the suffix tree top (as for arbitrary strings, generally speaking, is not true: for example, for the line). Usually this is achieved by assigning to end of line for some special character (usually denoted by a dollar sign).

For convenience, we introduce the notation: \bar{s} - A string s written in reverse order; $DAWG(s)$ - suffix automaton built for the line s ; $ST(\bar{s})$ - Is suffix tree line s .

We introduce the concept **expanding links**: Fix the top suffix tree v and the symbol c ; then extending the link leads to the top of the tree, the corresponding line $c + v$ (If this path $c + v$ ends in the middle of the edge, then draw a link to the lower end of the rib); if such a path $c + v$ is not in the tree, then extends link is not defined. In a sense, expanding links opposite suffix links.

Theorem 1. Wood formed by the suffix link $DAWG(s)$, Is the suffix tree $ST(\bar{s})$.

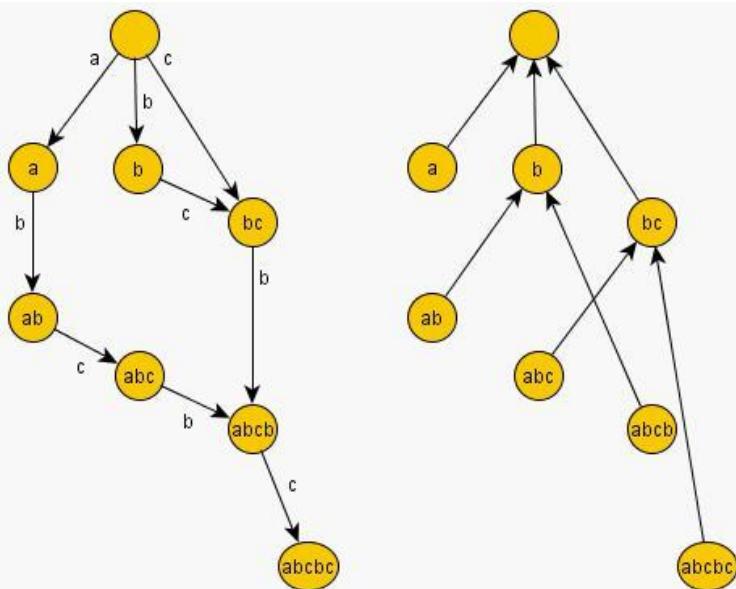
Theorem 2. - ~~DAWG(s)~~ graph of links extending suffix tree . Furthermore, solid edges in $DAWG(s)$ - Is inverted suffix links $ST(\bar{s})$.

These two theorems allow one of the structures (suffix tree or suffix automaton) to build another during $O(n)$. These two simple algorithms will be discussed below us in Theorems 3 and 4.

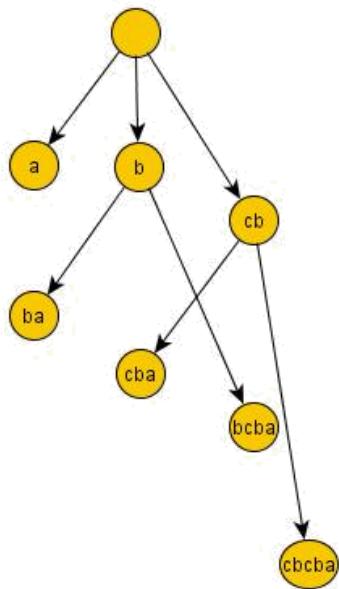
For illustrative purposes, let suffix tree automaton with its suffix links and the corresponding suffix tree for the inverted row.

For example, take a string $s = "abcbc"$.

$DAWG("abcbc")$ suffix tree and its links (for clarity, we sign each state its *longest*-Line):



$ST("cbcba")$.



Lemma. The following three statements are equivalent for any two substrings u and w :

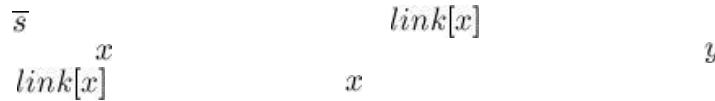
- $\text{endpos}(u) = \text{endpos}(w)$ in row s
- $\text{firstpos}(\bar{u}) = \text{firstpos}(\bar{w})$ in row \bar{s}
- \bar{u} and \bar{w} lie on the same path from the root to the suffix tree $ST(\bar{s})$.

Proof it's pretty obvious that if the beginning of occurrences of two strings are the same, then one string is a prefix of another, which means that one line lies in the suffix tree in the way of another string.

Proof of Theorem 1.

Suffix automaton states correspond to the vertices suffix tree.

Consider an arbitrary suffix link $y = \text{link}(x)$. According to the definition suffix reference $\text{longest}(y)$ is a suffix $\text{longest}(x)$, And all of such y choose the one with whom $\text{len}(y)$ maximum.



QED.

(x, y, c) $\boxed{\text{DAWG}(s)}$
 $\bar{s} \ y$ it is a condition which corresponds to the substring from which (in text \bar{s}) Coincides with $\text{longest}(x) + c$.
 firstpos substring from $c + \text{longest}(x)$.

It just means that:

$$\overline{\text{longest}(y)} = \text{ext}[c, \overline{\text{longest}(x)}].$$

The first part of the theorem, it remains to prove the second part: that all solid transitions in the machine correspond to the links in the suffix tree. Continuous transition is different from those of non-exhaustive,

that $\text{length}(y) = \text{length}(x) + 1$ ie after attributing characters C we were in a state with line, the maximum of the equivalence class of this state. This means that in the calculation corresponding spreading links $\text{ext}[c, \overline{\text{longest}(x)}]$ We immediately hit the top of the tree, and did not go down down to the nearest tree tops. Thus, assigning a single character in the beginning, we were top of the tree to another - so if it is inverted suffix link in the tree.

The theorem is proved.

Theorem 3. Having suffix automaton $\text{DAWG}(s)$ Can be a time $O(n)$ construct suffix tree $ST(\bar{s})$.

Theorem 4. With suffix tree $ST(\bar{s})$ Can be a time $O(n)$ build Suffix automaton $\text{DAWG}(s)$.

Proof of Theorem 3.

Suffix tree $ST(\bar{s})$ will contain the same number of vertices as the states in $\text{DAWG}(s)$, Wherein

top of the tree, resulting from the state v automaton corresponds to a string of length $\text{len}(v)$.

According to Theorem 1, the edges in the tree formed as inverted suffix links, and arc labels can be found on the basis of the difference between the states, and further knowing each state machine any one element of his set endpos (This is one element of endpos can be maintained in the construction of the machine).

Suffix links in the tree, we can build according to Theorem 2: it is enough to see all the solid transitions in the machine, and for each such transition (x, y) Add a link $\text{link}(y) = x$.

Thus, during the time $O(n)$ we can construct a suffix tree with a suffix link in it.

(If we consider the size of k alphabet is not constant, then all will take time to rebuild $O(n \log k)$.)

Proof of Theorem 4.

Suffix automaton $\text{DAWG}(s)$ will contain the same number of states as vertices $ST(\bar{s})$ Do each state its longest string $\text{longest}(v)$ will correspond to an inverted way of root of the tree to the top.

According to Theorem 2, to build all the transitions in the suffix automaton, we need to find all the links extend $\text{ext}[c, v]$.

First, note that some of these links extend obtained directly from the links in the suffix tree. In fact, if for any vertex x We consider it a suffix link, it means $y = \text{link}(x)$ that it is necessary to hold the link extends from y in x the first character string corresponding to the vertex x .

However, since we do not find all the links extend. Additionally, it is necessary to go through the suffix tree from the leaves to the root, and for each vertex v see all her sons, each son see all extend

references $[c, w]$ And copy this link to the top, if this symbol from the top link is not yet v was found:

This process will work during $O(n)$ If we assume a constant size alphabet.

Finally, it remains to construct suffix links in the machine, however, according to Theorem 1, this suffix links are obtained simply as a suffix tree edges $ST(\bar{s})$.

Thus, the algorithm builds a time machine $O(n)$ suffix suffix tree for the inverted row.

(If, however, we believe that the size of the alphabet - as a variable, then the asymptotic behavior will increase to $O(n \log k)$.)

Use in solving problems

We'll see what can be done with the help of suffix automaton.

For simplicity, we will assume a constant alphabet size, which will allow us to consider the asymptotic behavior of constructing suffix automaton and pass on it constant.

Checking entry

Condition. Given a text T And receives requests in the form: given a string P Required to verify whether or not the line is included P in text T as a substring.

Asymptotics. Preprocessing $O(\text{length}(T))$ and $O(\text{length}(P))$ one request.

Decision. Suffix construct automatic text T during $O(\text{length}(T))$.

How now to respond to one request. Let the current state - is a variable v It is initially equal to the initial state. Will be going for a character string P Appropriately making the transition from the current state v a new state. If at any time it happened that the transition from the current state to the desired character was not - the answer to the query "no." If we could treat the entire string P , The response to "yes."

It is understood that this will work during $O(\text{length}(P))$ Moreover, the algorithm is actually looking length of the longest prefix P Encountered in the text - and if the input samples are such that these lengths are small, then the algorithm will run much faster, without processing the entire string as a whole.

The number of different substrings

Condition. Given a string S . I want to know the number of its various substrings.

Asymptotics. $O(\text{length}(S))$.

Decision. Construct suffix automaton in line S .

In any machine suffix substring matches any path in the machine. Since the duplicate rows in the machine can not be, then the answer is - it **number of different ways** in the machine, starting at the initial vertex t_0 .

Given that suffix automaton is an acyclic graph, a number of different ways it can be considered by using dynamic programming.

Namely, let - how many different ways, starting with the state v (Including the path length of zero).

Then we have:

$$d[v] = 1 + \sum_{\substack{w : \\ (v, w, c) \in D \text{ AWG}}} d[w],$$

ie $d[v]$ can be expressed as the sum of responses to all possible transitions from state v .

Answer to the problem is the value of $d[t_0] - 1$ (Unit is subtracted to ignore empty substring).

The total length of different substrings

Condition. Given a string S . Want to know the total length of all its various substrings.

Asymptotics. $O(\text{length}(S))$.

Decision. Solution of the problem similar to the previous, Only now it is necessary to consider the dynamics of two values: the number of different substrings $d[v]$ and the total length $ans[v]$.

How to count $d[v]$, Described in the previous task, and the value of $ans[v]$ can be calculated as follows:

ie we take the answer for each vertex, and add to it $d[w]$ Thus like to start attributing each row one character.

The smallest cyclic shift

Condition. Given a string S . Required to find the lexicographically minimal cyclic shift her.

Asymptotics. $O(\text{length}(S))$.

Decision. Construct suffix automaton for the string $S + S$. Then this machine will contain both ways all cyclic shifts of the string S .

Consequently, the problem reduces to that found in the machine to the lexicographically minimal path length $\text{length}(S)$. That is done in a trivial way: we start in the initial state and each time act greedily, passing the transition with minimal character.

Number of occurrences

Condition. Given a text T And receives requests in the form: given a string P , Want to know how many times the line P text included in the T as a substring (entry may overlap).

Asymptotics. Preprocessing $O(\text{length}(T))$ and $O(\text{length}(P))$ one request.

Decision. Suffix construct automatic text T machine to count the number $cnt[v]$. Next we need to do this preprocessing: for each state v equal to the size of the $endpos(v)$. In fact, all the rows corresponding to one and the same T As are included in the T the same number of times equal to the number of positions in the set $endpos$.

However, explicit support set $endpos$ for all states, we can not, so learn to read only Sizes cnt .

To do this, proceed as follows. For each state, if it was not obtained by cloning (initial state and we also do not count), initially assign $cnt = 1$. Then we will go on

all states in order of their length len Mapped and the current value $cnt[v]$ suffix on the following link:

$$cnt[link(v)] += cnt[v].$$

It is alleged that in the end we did calculate for each state the correct values cnt .

Why is this true? Total states obtained by cloning is not, exactly $\text{length}(S)$ And of them appeared when we added the first i characters. Consequently, each of these states, we associate this position at which processing it appeared. So initially each such condition $\text{cnt} = 1$ And all other states $\text{cnt} = 0$.

Then we perform for each v such an operation: $\text{cnt}[link(v)] += \text{cnt}[v]$. The point of this is to that if the string corresponding to the state v , Met $\text{cnt}[v]$ times, then all of its suffixes will meet the same.

Why thus we do not take into account the same position a few times? Because each state of its value "Mapped" only once, so there could well be that one of his state to "icing" to some other state twice, in two different ways.

So we learned to count these values cnt for all states of the automaton.

After that prompted trivial - just return $\text{cnt}[t]$ Wherein t - The state corresponding to the model P .

Position of the first occurrence

Condition. Given a text T And receives requests in the form: given a string P , Want to know the position of the beginning of the first occurrence of the string P .

Asymptotics. Preprocessing $O(\text{length}(T))$ and $O(\text{length}(P))$ one request.

Decision. Suffix construct automatic text T

To solve the problem we must also add to the preprocessing finding positions firstpos all states of the automaton, ie for each state, we want to find a position $\text{firstpos}[v]$ end of the first occurrences. In other words, we want to find the minimum element in advance of each of the sets $\text{endpos}(v)$ (Because obviously support all sets endpos we can not).

Support these positions are easiest to right during the build machine: when we create new state cur when the function $\text{sa_extend}()$ Then expose him

(If we are working in 0-Indexing).

When cloning vertices q clone we put:

(As another embodiment, only one value of - is $\text{firstpos}(cur)$, Which is clearly more).

Thus, the response to the request - is simply $\text{firstpos}(t) - \text{length}(P) + 1$ Where - state corresponding to the sample P

The positions of all occurrences

Condition. Given a text T And receives requests in the form: given a string, you want to display the positions of all its occurrences in a row (entry can overlap).

Asymptotics. Preprocessing $O(\text{length}(T))$. Answer to a request for $O(\text{length}(P) + \text{answer}(P))$ Where $\text{answer}(P)$ - Is the size of the response, ie we will solve the problem for time order of the input and output.

Decision. Suffix construct automatic text T . Similarly to the previous problem, we can calculate in the process of constructing an automaton for each state position firstpos the end of the first occurrence.

Suppose now received a request - string P . Find, which state t it corresponds.

It is understood that $\text{firstpos}(t)$ should definitely go back. What other positions have to find? We took into account the state of machine containing the string P But did not consider other states, which correspond to lines like that P is their suffix.

In other words, we need to find all the states from which **achievable by suffix links** state t .

Therefore, to solve the problem we need to keep for each state suffix list of links leading to it. Reply to the request then it will be to make **traversal depth / across** on these inverted suffix links From the state t .

This tour will run during $O(\text{answer}(P))$ As we did not visit the same state twice (because of the condition of each suffix link goes only one, so no two routes leading to the same state).

However, we must remember that the two states of ~~their values may coincide~~: if one state was obtained by cloning the other. However, this does not worsen the asymptotic behavior as each non-cloned tops can be a maximum of one clone.

Moreover, you can easily get rid of duplicate output positions, if we do not add to the response from the states *firstpos* clones. In fact, in any state-clone is a suffix link from the initial state, which is a state of cloned. Thus, if we remember each state flag *is_clon* And will not add to the response from the states for *firstpos* which *is_clon* = *true*, We thus obtain all required *answer(P)* positions without repeats. We give an outline of implementation:

```

struct state {
    ...
    bool is_clon;
    int first_pos;
    vector <int> Inv_link;
};

After building a machine ... ...
for (int v = 1; v < sz; ++ v)
    st[st[v]. Link]. Inv_link.push_back (v);
...

/ / Response to the request - the withdrawal of all occurrences
(possibly with repetitions) void output_all_occurrences (int v,
int P_length) {
    if (! st[v]. Is_clon)
        cout << St[v]. First_pos - P_length + 1 <<
    Endl; for (size_t i = 0; i < st[v]. Inv_link.size(); ++
    I)
        output_all_occurrences (st[v]. Inv_link[i], P_length);
}

```

Find the shortest line that is not in this

Condition. Given a string *S*, and set to a specific alphabet. Required to find a string of minimum length that it does not occur in *S* as a substring.

Asymptotics. Solution for $O(\text{length}(S))$.

Decision. Will decide on dynamic programming automaton built for the string *S*.

Let $d[v]$ - This is the answer for the top *v*le we have already typed in a part of the substring, being able to *v*And want find the smallest number of characters that must add, to go beyond the machine, finding a nonexistent transition.

Considered $d[v]$ to be very simple. If, *v* no transition at least one character from the alphabet, then $d[v] = 1$ attribute such a symbol to go beyond the machine, thereby obtaining a search string.

Otherwise, a single character do not work, so we must take a minimum of answers for all sorts of characters:

$$d[v] = 1 + \min_{\substack{w: \\ (v,w,c) \in DAWG}} d[w].$$

Answer to the problem will be $d[0]$, equal, and the line itself can be restored by restoring the manner in which the dynamics turned this minimum.

Of the longest common substring of two strings

Condition. Given two strings *S* and *T*. Required to find them of the longest common substring ie this line *A*, that and it is a substring *S*And *T*.

Asymptotics. Solution for $O(\text{length}(S) + \text{length}(T))$.

Decision. Construct suffix automaton in line *S*.

We will now go on line *T*And for each prefix of the longest suffix search this prefix is occurring. In other words, we have for each position in the string we want to find the total of the longest substring *S* and *T*Ending it in this position.

For this purpose, we maintain two variables: **the current state of v** and **current length l** . These

two variables will describe the current matching part: its length, and a state that corresponds to her (Without storage length can not be avoided, as one of several can match strings of different lengths).

Initially $p = t_0, l = 0$ ie Match empty.

Now let us consider the symbol $T[i]$ and want to recalculate the answer for him.

- If the state of v in the machine there is a transition from the symbol, we simply accomplish this transition and increase l unit.
- If the state of v not have the required transition, we should try to shorten the current matching part, which should go to suffix link:

$$v = \text{link}(v).$$

In this case it is necessary to shorten the current length, but leave as possible. Obviously, this should be assigned to $v = \text{len}(v)$, because after a rush down the suffix link us to satisfy any substring length corresponding to that of:

$$l = \text{len}(v).$$

If from the new state will not go back to the desired character, then again, we should go on and reduce the suffix link, and so on, until we find the transition (then go to step 1) or we do not get

a fictitious state -1 (Which means that symbol $T[i]$ does not occur in so S assign $v = l = 0$ and proceed to the next \emptyset).

Answer to the problem will be a maximum of the values for all the rounds.

Asymptotics of this passage is $O(\text{length}(T))$. Because in one move, we can either increase by unit, or to make multiple passes over the suffix link, each of which will severely reduce the value l . Consequently, reductions could not be more $\text{length}(T)$, Which means that the linear asymptotic behavior.

Implementation:

```
string lcs (string s, string t) {
    sa_init();
    for (int i = 0; i < (int)s.length(); + + i)
        + I) sa_extend (s[i]);
    int v = 0, L = 0,
        best = 0, Bestpos = 0;
    for (int i = 0; i < (int)t.length(); + + i) {
        while (v &&! st[v]. Next.count(t[i])) {
            v = st[v]. Link;
            l = st[v]. Length;
        }
        if (st[v]. Next.count(t[i]))
            { v = st[v]. Next[t[i]]; + + L;
        }
        if (l > best)
            best = l, bestpos = i;
    }
    return t.substr (bestpos-best +1, Best);
}
```

Most common substring multiple rows.

Condition. Dana K rows. Required to find them of the longest common substring, ie this line that it is a substring of all S_i

Asymptotics. Solution for $O(\sum \text{length}(S_i) \cdot K)$.

Decision. Merge all lines T in one line after each line attributing their own delimiter character D_i entering K extra special. characters: P_i

Construct a string T suffix automaton.

Now we need to find this line in the machine, which contains all the lines, and this will help us to find the characters. Note that if a substring occurs in some string S_j Then

suffix automaton in a substring of this there is a path that contains the symbol D_j And containing no other character $D_1, \dots, D_{j-1}, D_{j+1}, \dots, D_k$.

Thus, we need to calculate the reachability: for each state of the automaton and each character D_i yes if the path contains a separator D_i And not containing other separators. It's easy to make a circuit in depth / width or lazy dynamics. After that, the answer to the problem is the string $\text{longest}(v)$ for the state, v from which the path was found for all symbols.

Literature

We first give a list of the first works related to suffix automata:

- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, R. McConnell. **Linear Size Finite Automata for the Set of All Subwords of a Word. An Outline of Results** [1983]
- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler. **The Smallest Automaton Recognizing the Subwords of a Text** [1984]
- Maxime Crochemore. **Optimal Factor Transducers** [1985]
- Maxime Crochemore. **Transducers and Repetitions** [1986]
- A. Nerode. **Linear automaton transformations** [1958]

In addition, more contemporary sources, this issue is addressed in many books on string algorithms:

- Maxime Crochemore, Wojciech Rytter. **Jewels of Stringology** [2002]
- Bill Smyth. **Computing Patterns in Strings** [2003]
- Bill Smith. **Methods and algorithms for computing row** [2006]

Finding all subpalindromes

Statement of the Problem

Given a string s length n . You want to find all the pairs where $i \leq j$ That substring is a palindrome (ie reads the same from left to right and right to left). $s[i \dots j]$

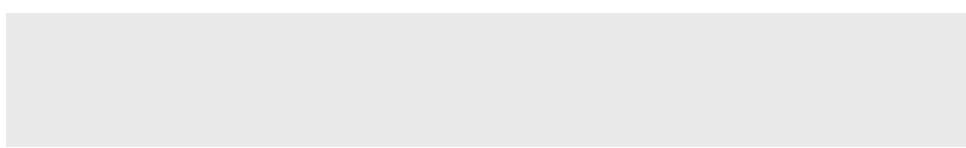
Clarification of statement

Clearly, in the worst case, such substring-palindromes can be that algorithm with linear asymptotic behavior can not exist. $O(n^2)$, And at first glance it seems

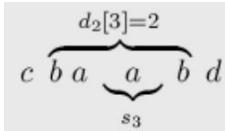
However, information can be found palindromes return more densely: For each

position $i = 0 \dots n - 1$ we find the values $d_1[i]$ and $d_2[i]$ Denoting the number of palindromes respectively odd and even length with center position i .

For example, in row $s = ababab$ There are three odd length palindrome with center symbol $s[3] = b$ ie value $d_1[3] = 3$:
value $d_1[3] = 3$:



In line $s = cbaabd$ There are two palindrome of even length with center symbol $s[3] = a$ ie value $d_2[3] = 2$:



The idea is that if there is a palindrome length with center i at any position, there is also subpalindromes length $l - 2$ $l - 4$ Etc. with centers. Therefore, two such arrays $d_1[i]$ and $d_2[i]$ enough to store information about all subpalindromes this line.

Rather unexpected fact is that there is a fairly simple algorithm that calculates these "arrays palindrome" $d_1[]$ and $d_2[]$ in linear time. This algorithm is described in this article.

Decision

Generally speaking, the problem has several well-known solutions: using hashing technique it can be solved for $O(n \log n)$ And using suffix trees and fast algorithm LCA this problem can be solved $O(n)$.

However, the article described in this method is much simpler and has a lower latent constants asymptotics of time and memory. This algorithm was discovered **Glenn Manacher (Glenn Manacher)** in 1975

Trivial algorithm

To avoid ambiguities in the following description, we agree that there is a "trivial algorithm."

This algorithm, which is to find an answer at the position i repeatedly tries to increase response by one each time comparing a pair of corresponding symbols.

Such an algorithm is too slow, the whole answer as he may deem just in time

$O(n^2)$. Give visibility to its implementation:

```
vector <int> D1 (n), D2 (n); for (int i = 0; i <n; + + I) { d1[i] = 1;
```

```
while (i-d1[i] >= 0 && I + d1[i] <N && s[i-d1[i]] == s[i +  
d1[i]]) ++ D1[i];
```

```

d2[i] = 0;
while (i-d2[i]-1 >= 0 && i + d2[i] < N && s[i-d2[i]-1] == s[i +
    d2[i]]) ++ D2[i];
}

```

Algorithm Manakera

First learn to find all subpalindromes odd length, ie calculate the array; solution for palindromes of even length (ie, finding the array $d_1[]$) A slight modification of this.

For a quick calculation will support **border** the right-most of the detected subpalindromes (ie, with the highest value subpalindromes r). Initially, we can put $l = 0, r = -1$.

Thus, suppose we want to calculate the $d_1[i]$ value for the next i . All previous values $d_1[]$ already counted.

- If i is not within the current subpalindromes, i.e. $i > r$ Then simply run a trivial algorithm.

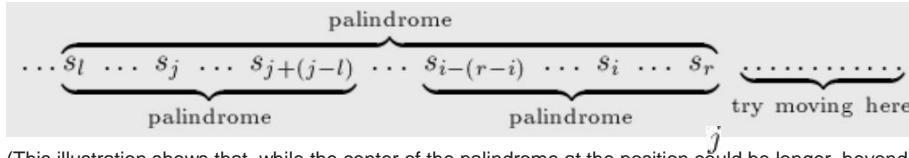
Ie will consistently increase the value s And check every time - true if the current substring $d_1[i]$ is a palindrome. When we find the first difference, or when we reach the borders line - stay we finally counted value $d_1[i]$. After that, we must do not forget to update the values (l, r)

- Consider now the case when $i \leq r$. Try to extract the information from the already calculated values $d_1[]$. Namely, the position will reflect inside subpalindromes (l, r) Ie obtain position $j = l + (r - i)$ of $d_1[j]$ And consider the meaning i . Since j - A position symmetrical position i . Then **almost always** we can simply assign $d_1[i] = d_1[j]$. An illustration of this reflection (palindrome around actually "copied" in palindrome around i)

However, there is **subtlety** which must be processed correctly when "internal palindrome" reaches external borders or climbs over it, ie, $j - d_1[j] + 1 \leq l$ (Or, equivalently, $i + d_1[j] - 1 \geq r$). As for the external borders of the palindrome is no symmetry can not be guaranteed, then simply assign $d_1[i] = d_1[j]$ is already correctly: we have enough information to say that in position i palindrome has the same length.

In fact, to properly handle these situations, it is necessary to "trim" subpalindromes length, ie assign. After that you should let a trivial algorithm that will try to increase the value of $d_1[i]$ As long as possible.

An illustration of this case (the one palindrome with center shows already "clipped" to such a length that it is placed right next to the outer palindrome)



(This illustration shows that, while the center of the palindrome at the position i could be longer, beyond the outer limits of the palindrome, - but in a position i we can use only the part of it that actually fit into the outer palindrome. But the answer to the position i may be more than this part, so then we have to run a trivial search, which will try to push it beyond external palindrome, ie to "try moving here".)

At the end of the description of the algorithm just happened to recall that we must not forget to update the values (l, r) after calculating the next value $d_1[i]$.

Also we should mention that we have described above reasoning to calculate the array of odd palindromes $d_1[]$; for array even palindromes $d_2[]$ All the arguments are similar.

Score algorithm asymptotics Manakera

At first glance it is not obvious that this algorithm has a linear asymptotic form for calculating the response for a particular position in it often runs a trivial algorithm for finding palindromes.

But a closer analysis shows that the algorithm is still linear. (It should refer to the known

algorithm for constructing the Z-line functions, internally which strongly resembles the algorithm, and also works in linear time.)

In fact, it is easy to trace algorithm, each iteration produced a trivial finding leads to an increase of one border R . The decrease in R in the course of the algorithm can not occur. Consequently, a trivial algorithm to make a sum $O(n)$ action.

Given that, except for trivial searches, all other parts of the algorithm Manakera obviously work in linear time, we obtain the asymptotic behavior of the final: $O(n)$.

Implementation of the algorithm Manakera

For the case subpalindromes odd length, ie for computing an array d_1 , We obtain the following code:

```
vector <int> D1
(n); int l = 0, R = -1;
for (int i = 0; i <n; + + I) {
    int k = (i > r? 0 : Min (d1[l + r-i], R-i)) + 1;
    while (i + k <n && i-k >= 0 && S[i + k] == S[i-k]) +
        && K;
    d1[i] = K -;
    if (i + k > r)
        l = i-k, r = i + k;
}
```

For subpalindromes even length, ie for computing an array d_2 , Only slightly changed arithmetic expressions:

```
vector <int> D2
(n); l = 0, R = -1;
for (int i = 0; i <n; + + I) {
    int k = (i > r? 0 : Min (d2[l + r-i +1], R-i +1)) + 1;
    while (i + k-1 <n && i-k >= 0 && S[i + k-1] == S[i-k]) +
        + K; d2[i] = - K;
    if (i + k-1 > R)
        l = i-k, r = i + k-1;
}
```

Problem in online judges

List of tasks that can be taken using this algorithm:

- UVA # 11475 "Extend to Palindrome" [Difficulty: Easy]

Lyndon decomposition. Duval algorithm. Finding the smallest cyclic shift

Lyndon decomposition concept

Define the concept of **Lyndon decomposition** (Lyndon decomposition).

Line is called **simple** if it is strictly less any of its own **suffix**. Examples of simple lines: a, ab, abc . It can be shown that the line is simple, and then only when it is strictly less all its nontrivial **cyclic shifts**.

Further, suppose given a string s . Then **Lyndon decomposition** line s called its expansion $s = w_1 w_2 \dots w_k$ Where line w_i simple, and wherein $w_1 \geq w_2 \geq \dots \geq w_k$.

Can show that for any row s this decomposition exists and is unique.

Duval algorithm

Duval algorithm (Duval's algorithm) finds a given string length n Lyndon decomposition during $O(n)$ using $O(1)$ additional memory. Will work with

strings in the 0-indexed.

We introduce the auxiliary notion predprostoy line. Line t called **predprostoy** If it has the form

$t = w w \dots w \bar{w}$ Wherein w - Some simple string and \bar{w} - Prefix of some w .

Duval algorithm is greedy. At any time during the work string S is actually divided into three rows $s = s_1 s_2 s_3$ Where line s_1 Lyndon decomposition has already been found and s_1 no longer used algorithm; line s_2 - It predprostaya string (length and simple lines inside it, we also remember); line s_3 - It is not part of the line treated s . Every time algorithm Duval takes the first character s_3 and tries to add it to the string s_2 . At the same time, perhaps, for some prefix string s_2 Lyndon decomposition becomes known, and this part of the proceeds to line s_1 .

We now describe the algorithm **formally**. First, it will be supported by a pointer to the beginning of the line.

The outer loop of the algorithm is executed until

Le until the entire string does not go into a string. Inside this cycle are two pointers: a pointer at the beginning of the line s (Actually a pointer to the next symbol candidate) and a pointer j at the current character in the string with which to compare. Then we in a loop trying to add character $s[j]$ to line s_2 , Which must make the comparison with the symbol $s[k]$. Here we're having three different cases:

- If $s[j] = s[k]$ Then we can finish symbol $s[j]$ to the line without breaking it "predprostoy." Consequently, in this case we simply increment pointers and j . It is obvious that the $s_2 + s[j]$ unit.
- If $s[j] > s[k]$ line $s_2 + s[j]$ will be easy. Then we increase by one, and k we move back to i . To the next character compared with the first character.
- If $s[j] < s[k]$, The string $s_2 + s[j]$ can no longer be predprostoy. So we split the string predprostoyu s_2 on simple lines plus "balance" (a simple string prefix, possibly empty); add a simple string in response (ie, derive their position, simultaneously moving a pointer), and the "balance" with the symbol translate back to $s[j]$ line s_3 And stop the execution of the inner loop. Thus we have the next iteration of the outer loop to re-process residue, knowing that he could not form a line with previous predprostoyu simple strings. It remains only to note that the derivation of simple lines of products we need to know their length; but it is obviously $j - k$.

Implementation

We present implementation of the algorithm Duval that will output the desired decomposition Lyndon line s :

```
string s; // input string
int n = (int) s.length();
int i = 0;
while (i < n) {
```

```

int j = i +1, K = i;
while (j <n && s[k] <= s[j]) {
    if (s[k] <s[j])
        k = i;
    else
        + + K;
    + + J;
}
while (i <= k) {
    cout << S.substr (i, j-k) << ' ';
    i += j - k;
}
}

```

Asymptotics

Immediately, we note that the algorithm requires Duval $O(1)$ Memory, namely three pointer i, j, k . We now estimate **operation time** algorithm.

While the outer loop makes no more n iteration, since the end of each iteration, it displays at least one symbol (total symbols are output rather obviously exactly n).

We now estimate the number of iterations **first nested loop while**. For this we consider the second nested loop while - every time he's run a number of outputs $r \geq 1$ copies of the simple string of some length $p = j - k$. Note that the line s_2 predprostoy is, with its simple strings have a length of just p ple its length does not exceed $rp + p - 1$. Since the length of the string equal $j - i$ And the pointer j is increased by one at each iteration of the inner loop of the first while, then this loop will execute no more $+ p - 2$ iterations. The worst case is that $r = 1$ And we get that while the first nested loop executes every time no more $2p - 2$ iterations. Recalling that the total output n characters, we find that for output n requires no more characters $2n - 2$ iterations of the first nested while-a.

Consequently, **Duval algorithm runs in $O(n)$** .

Easy to estimate the number of character comparisons performed by the algorithm Duval. Since each iteration of the inner loop while the first two comparisons produces character, and one comparison is made after the last iteration (to understand that the cycle has to stop), the total **number character comparisons** does not exceed $4n - 3$.

Finding the smallest cyclic shift

Suppose given a string S . Construct a string $S + S$ Lyndon decomposition (we can do it in time and space $O(n)$). Find predprostoy block that begins in a position at n (ie, in the first instance of a string S) And ends at a position greater than or equal to n (ie, in the second instance). It is alleged that **start position** this unit will be the beginning of the desired cyclic shift (this is easily seen, using the definition of Lyndon decomposition).

Starting block predprostogo find easy - just notice that the pointer i at the beginning of each iteration of the outer loop, while points to the beginning of the current block predprostogo.

Overall, we obtain the following **implementation** (To simplify the code it uses memory, explicitly way appending string to himself):

```

string min_cyclic_shift (string s) {
    s += s;
    int n = (int) s.length();
    int i = 0, Ans = 0;
    while (i <n /2) {
        ans = i;
        int j = i +1, K = i;
        while (j <n && s[k] <= s[j]) {
            if (s[k] <s[j])
                k = i;
            else
                + + K;
            + + J;
        }
        while (i <= k) i += j - k;
    }
}

```

```
    return s.substr (ans, n /2);  
}
```

Problem in online judges

List of tasks that can be solved using the algorithm Duval:

- UVA # 719 "Glass Beads" [Difficulty: low]

Algorithm Aho-Korasik

Suppose we are given a set of strings over the alphabet size of the total length m . Aho-Korasik algorithm builds a set of rows for this data structure "Bor", and then on the forest $O(m)$ builds automatic, for all time and memory. The resulting machine can already be used in various problems, the simplest of them - is to find all occurrences of each line of a given set in some text in linear time.

This algorithm was proposed by Canadian scientists Alfred Aho (Alfred Vaino Aho) and scientists Korasik Margaret (Margaret John Corasick) in 1975

Bor. Construction of boron

Formally, **boron** - A tree rooted at a vertex, and each edge of a tree by a letter signed. If we consider the list of edges from a given vertex (except ribs, leading to an ancestor), then all edges must have different labels.

Consider in boron any path from the root; write out the row labels of edges of this path. As a result, we get some string, which corresponds to this path. If we consider any vertex boron, she assign the string corresponding to the path from the root to this vertex.

leaf

Each vertex of boron also has a flag that is true if the top ends of a string of any given set.

Accordingly, **build boron** on the set of strings - means to build such a boron that each - leaf top will match any row from the set, and vice versa, each row of the set will be correspond to some leaf vertex.

We now describe **how to build a boron** for a given set of strings in linear time with respect to their total length.

We introduce a structure corresponding to the vertices of boron:

```
struct vertex {
    int next[K];
    bool leaf;
};

vertex t[NMAX
+1]; int sz;
```

Ie We will store boron as an array (number of elements in the array - it sz) structures

t

Wherei

vertex

Structure vertex contains a flag leaf [And an array of ribs next[] in next[i] - Pointer to the top, in which an edge on the symbol ? Or -1 If no such edge.

Initially, boron consists of only one vertex - root (agree that the root is always in the array t index 0).

So **initialization** Boron is:

```
memset (t[0]. Next, 255, sizeof t[0].
Next); sz = 1;
```

Now implement a function that will add boron given string s. Implementation is very simple:

we get to the root of boron, see if there from the root of the transition in the

letter

If the transition is, just go

s[0]

Then

on it in the other vertex, otherwise create a new vertex and add a transition to this vertex by letter

s[1] Etc. After completing the mark

we are standing in some vertex, repeat the process for the

letter

s[1]

last visited the top of the flag

leaf = true .

```
void add_string (const string & s) {
    int v = 0;
    for (size_t i = 0; i < s.length(); ++i) {
        char c = s[i] - 'A'; // depending on the
        alphabet if (t[v]. Next[c] == -1) {
            memset (t[sz]. Next, 255, sizeof t[sz].
Next); t[v]. Next[c] = Sz++;
        }
        v = t[v]. Next[c];
    }
}
```

```

    } t[v].Leaf = true;
}

```

Linear time work, as well as a linear number of vertices in boron obvious. Since each node have

$O(k)$ memory, the memory usage has $O(nk)$.

Memory consumption can be reduced to a linear (), but due to the increase to the asymptotic behavior of $O(n \log k)$. It's enough to keep the transitions `next` not an array, and the mapping `map < char,int >`.

Construction machine

Suppose we have constructed a forest for a given set of strings. Look at it now, some on the other side. If we consider any vertex, the line that corresponds to it, is a prefix of one or more rows from the set; ie each vertex of boron can be understood as a position in one or more rows from the set.

In fact, the top of boron can be understood as the state **deterministic finite automaton**. While in some state, we are under the influence of any input letters go to

another state - ie to another position in the rowset. For example, if boron is only string "abc" we are in the state (which corresponds to the line "ab"), Then exposed to the letters "c" we proceed in condition 3

Ie we can understand the edges of boron as transitions in the machine on the corresponding letter. However, only some edges of boron can not be limited. If we try to make the transition to a letter, and the corresponding edge in boron not, we still need to go to some state.

More precisely, let we are able to p , Which corresponds to some string t And want to jump on the symbol c . If the boron from the top p there is a transition in the letter c We just go along this edge and get to the top, which corresponds to the line tc . If no such edge, then we must find a state corresponding to the line of the longest proper suffix t (Of the longest available in boron), and try to jump on the letter c thereof.

For example, let Bor built in rows "ab" and "bc" And we exposed line "ab" moved to a state that is a leaf. Then, under the influence of the letters "c" we are forced to go to state corresponding to the line "b" And only from there jump to the letter "c".

Suffix link for each vertex p - Is the pinnacle, which ends of the longest proper suffix of the row corresponding to the top p . The only special case - the root boron; for the convenience of the suffix link from it to spend for themselves. Now we can reformulate the statement about the transitions in the machine as follows: while the current top of the boron is no transition on the corresponding letter (or until we come to the root of boron), we have to move on suffix link.

Thus, we have reduced the problem of constructing an automaton to the problem of finding suffix links for all vertices of boron. However, to build these links, we will suffice, oddly enough, on the contrary, with the help of built in automatic transitions.

Note that if we want to know the suffix link for some vertex v , Then we can go to the ancestor p current node (let c - A letter, of which at p there is a transition in v), Then go to its suffix link, and then jump out of it in the machine, according to the letter c .

Thus, the problem of finding the transition is reduced to the problem of finding suffix links, and the task of finding links suffix - suffix to the task of finding links and go, but for closer to the root vertices. We got a recursive relationship, but not infinite, and, furthermore, that can solve in linear time.

We now turn to **implementation**. Note that we now need to store each vertex of its ancestor p As well as the character to the ancestor of a transition in our top. Also at each vertex will store `int link`- Suffix link (or -1If it is not calculated), and an array `int go[k]`- Transitions in the machine for each of the characters (again, if the array element is equal -1, He has not yet calculated). We now present the full realization of all necessary functions:

```

struct vertex {
    int next[K];
    bool leaf;
    int p;
    char pch;
    int link;
    int go[K];
};

vertex t[NMAX
+1];int sz;

```

```

void init() {
    t[0]. P = t[0]. Link = -1;
    memset (t[0]. Next, 255, sizeof t[0]. Next);
    memset (t[0]. Go, 255, sizeof t[0]. Go);
    sz = 1;
}

void add_string (const string & s) {
    int v = 0;
    for (size_t i = 0; i < s.length(); ++ i) {
        char c = s[i] - 'A';
        if (t[v]. Next[c] == -1) {
            memset (t[sz]. Next, 255, sizeof t[sz]. Next);
            memset (t[sz]. Go, 255, sizeof t[sz]. Go);
            t[sz]. Link = -1;
            t[sz]. P = v; t[sz]. Pch = c; t[v]. Next[c] = Sz++;
        }
        v = t[v]. Next[c];
    }
    t[v]. Leaf = true;
}

int go (int v, char c);

int get_link (int v) {
    if (t[v]. Link == -1)
        if (v == 0 || T[v]. P == 0) t[v]. Link = 0;
    else t[v]. Link = go (get_link (t[v]. P), T[v]. Pch);
    return t[v]. Link;
}

int go (int v, char c) {
    if (t[v]. Go[c] == -1)
        if (t[v]. Next[c] == -1) t[v]. Go[c] = T[v]. Next[c];
    else t[v]. Go[c] = v == 0 ? 0 : Go (get_link (v), C);
    return t[v]. Go[c];
}

```

It is easy to understand that, by remembering found suffix links and transitions, the total time for finding all the suffix links and transitions is linear.

Applications

Search all the rows from a given set of text

Given a set of strings, and text data. You want to display all occurrences of a set of lines in the text during $O(\text{Len} + \text{Ans})$. Wherein Len - Length of the text Ans - Size of the response.

Construct from a given set of rows boron. We now handle text, one letter, moving properly on the tree, actually - to states of the automaton. Initially, we are at the root of the tree. Let us at the next step we are able to v And the next letter of the text C . Then make the transition to the state, thereby increasing or at 1 length of the current matching substring,

$$\text{go}(v, c)$$

or decreasing it, passing through the suffix link.

Now know as the current state v . There is a match with any of a set of strings? First, it is clear that if we are in the marked vertex (), then there is a match with that pattern, which ends at the top of the boron v . However, this is not the only case of coincidence achieve if we moving on the suffix links, we can achieve one or more marked vertices, the agreement will also, but for samples ending in these states. A simple example of such a situation

- When a set of strings - it {"dabce", "abc", "bc"} And the text - it "dabc".

Thus, if each marked vertex store the sample number that ends in it (or a list of numbers, if allowed duplicate samples), we can for the current state for a Room

$O(n)$

All samples for which reached a coincidence, just passing on suffix links from the current node to the root. However, this is not enough effective solution because the asymptotic behavior of the sum will.

$O(n \cdot \text{Len})$

However, it can be seen that the motion suffix links can optimizirovat previously counted for each vertex of the nearest labeled vertices reachable by suffix links (called "output function"). This value can be considered lazy dynamics in linear time. Then for

current vertex we can for

$O(1)$

find the following in the path suffix marked vertex, ie

action and the

$O(1)$ amount of

the next match. Thus, for each match will be spent

$O(\text{Len} + \text{Ans})$.

In the simplest case, when you have not find themselves entering, but only their number, you can substitute the output function to count the number of labeled lazy dynamics vertices reachable from the current node v

by suffix links. This value can be calculated for

$O(n)$

in the sum, and then for the current state v

we can for $O(1)$ find the number of occurrences of all the samples in the text, ending at the current position. Topics

Thus, the problem of finding the total number of occurrences can be solved for us $O(\text{Len})$.

Finding the lexicographically smallest string of a given length that does not contain none of these samples

Given a set of samples, and given the length of L . Want to find a string of length L . Not containing none of the samples, and all of these lines derive lexicographically smallest.

Construct from a given set of rows boron. Now recall that the vertices of which suffix links can be achieved marked vertices (vertices and such can be found in, for example, a lazy dynamics)

$O(n)$

can be perceived as an entry in the rows of the specified text. Since in this problem we need to avoid occurrences, it can be understood as the fact that such vertices we can not go. On the other hand, all the other vertices we can go.

Thus, we remove all of the automaton

"Bad" vertices, and in the remaining graph of the automaton is required to find the lexicographically minimal path length L . This problem can be solved already $O(L)$ Eg [dfs](#).

Finding the shortest string containing all occurrences of both samples

Again we use the same idea. For each node will store the mask indicating the samples for which the occurrence took place in the top. Then the problem can be reformulated as follows: initially in a state is

required to reach a state where $\text{Msk} = 0$

$(v, \text{Msk} = 2^n - 1)$

number of samples. Transitions from one state will be adding one letter to the text, ie, the transition along the edge of the top of the machine to another with a corresponding change in the mask. Running [wide detour](#) on

such a graph, we will find the way to the state $(v, \text{Msk} = 2^n - 1)$ minimum length that we just needed.

Finding the lexicographically smallest string length L Containing data samples in total k time

As in the previous problems, we can calculate for each vertex of the number of times that corresponds to it (ie, the number of marked vertices reachable from it by suffix links). Reformulate the problem

thus: current state is defined by three numbers

$(\text{Whe}, \text{Len}, \text{Cnt})$

And requires from

state $(\text{Root}, 0, 0)$ come to the state (v, L, k) rein v - Any vertex. Transitions between states

- It's just on the edges of the machine transitions from the current node. Thus, simply find [bypass depth](#) the path

between these two states (If the bypass will be in depth view of their letters natural order, then found the path will automatically lexicographically smallest).

Problem in online judges

Tasks that can be solved using boron or algorithm Aho-Korasik: •

[UVA # 11590 "Prefix Lookup"](#) [Difficulty: low]

• [UVA # 11171 "SMS"](#) [Difficulty: Medium]

- UVA # 10679 "I Love Strings!!!" [Difficulty: Medium]

Suffix tree. Ukkonen's algorithm

This article - a temporary cap, and does not contain any descriptions.

Ukkonen algorithm description can be found, for example, in the book Gasilda "strings, trees and sequences in the algorithms."

Implementation of the algorithm Ukkonen

This algorithm builds a suffix tree for a given string length n during $O(n \log k)$ Where - size alphabet (if it be assumed constant, then the asymptotic behavior is obtained $O(n)$).

The input data for the algorithm are the string s and the length n that are transmitted in the form of global variables.

The main function - `build_tree` She builds a suffix tree. Tree is stored as an array of structures `node`, where `node[0]` - Root suffix tree.

For simplicity, code is stored in the edge of the same structures: for each node in its structure recorded data on the edge that came out of her ancestor. Overall, each `node` stored: (l, r) Defining label $s[l..r - 1]$ ribs ancestor `par` - Top of ancestor `link` - Suffix link `next` - List outgoing edges.

```
string s;
int n;

struct node {
    int l, r, par, link;
    map <char,int> Next;
    node (int l = 0,int r = 0,int par = -1)
        : L(l), R(r), Par(par), Link(-1) {}
    int len() { return r - l; }
    int & Get (char c) {
        if (! Next.count(c)) next[c] = -1;
        return next[c];
    }
};

node t[MAXN];
int sz;
struct state {
    int v, pos;
    state (int v, int pos) : V(v), Pos(pos) {}
};

state ptr (0,0);

state go (state st, int l, int r) {
    while (l < r) {
        if (st.pos == t[st.v].Len()) {
            st = state (t[st.v].Get( s[l]),0);
            if (st.v == -1) return st;
        } else {
            if (s[ t[st.v].L + st.pos ] != s[l])
                return state (-1,-1);
            if (r-l <t[st.v].Len() - St.pos)
                return state (st.v, st.pos + r-l);
            l += t[st.v].Len() - St.pos;
            st.pos = t[st.v].Len();
        }
    }
    return st;
}

int split (state st) {
    if (st.pos == t[st.v].Len())
        return st.v;
```

```

        if (st.pos == 0)
            return t[st.v];
    Par; node v = t[st.v];
    int id = sz + ;
    t[id] = Node (v.l, v.l + st.pos,
    v.par); t[v.par]. Get( s[v.l]) = Id;
    t[id]. Get( s[v.l + st.pos]) = St.v;
    t[st.v]. Par = id;
    t[st.v]. L += st.pos; return id;
}

int get_link (int v) {
    if (t[v]. Link! = -1) return t[v]. Link;
    if (t[v]. Par == -1) return 0;
    int to = get_link (t[v]. Par);
    return t[v]. Link = split (go (state(to, t[to]. Len()), T[v]. L
+ (t [v]. Par ==0), T[v]. R));
}

void tree_extend (int pos) {
    for(;;) {
        state nptr = go (ptr, pos, pos
+1);if (nptr.v! = -1) {
            ptr = nptr;
            return;
        }

        int mid = split (ptr);
        int leaf = sz + ;
        t[leaf] = Node (pos, n, mid);
        t[mid]. Get( s[pos]) = Leaf;
        ptr.v = get_link (mid);
        ptr.pos = t[ptr.v]. Len();if (! Mid) break;
    }
}

void build_tree() {
    sz = 1;
    for (int i =0; i <n; + +
    I) tree_extend
    (i);
}

```

Compressed implementation

We also give the following compact realization algorithm Ukkonen proposed [freopen](#):

```

const int N =1000000, INF
=1000000000; string a;
int t[N] [26], l[N], r[N], p[N], s[N], tv, tp, ts =2;

void ukkadd (int c) {
    suff:;
    if (r[tv]<Tp) {
        if (t[tv] [c]== -1) { t[tv] [c]= Ts; l[ts]= A.size()-1; r[ts]=
            INF; p[ts + ]= Tv; tv = s[tv]; tp = r[tv]+1;goto suff;
            }
        tv = t[tv] [c]; tp = l[tv];
    }
    if (tp == -1 || c == a[tp]) tp + +; else { l[ts +1]=
        A.size()-1; r[ts +1]= INF; p[ts +1]= Ts;
        l[ts]= l[tv]; r[ts]= Tp-1; p[ts]= P[tv]; t[ts] [c]= Ts +1;
        t[ts]
[a[tp]]= Tv;

```

```
l[tv]=Tp; p[tv]=Ts; t[p[ts]] [a[l[ts]]]= Ts; ts +
=2; tv = s[p[ts-2]]; tp = l[ts-2];
while (tp <= r[ts-2]) { tv = t[tv] [a[tp]]; tp += r[tv]-
L[tv]+1;}
```

```

        if (tp == r[ts-2]+1) s[ts-2]= Tv; else s[ts-2]=
        Ts; tp = r[tv]-(tp-r[ts-2])+2;goto suff;
    }

void build() { fill(r, r
    + N, INF);
s[0]=1; l[0]= -
1; r[0]= -1;
l[1]= -1; r[1]=
-1;

memset (t, -1, sizeof t);
fill(t[1], T[1]+26,0);
for (int i =0; i <a.size(); +
    + I) ukkadd (a[i]-
'A');

}

```

The same code, commented:

```

const int N =1000000,      / / Maximum number of vertices in the suffix tree
INF =10000000000; // constant "Infinity"
                input string, for which it is necessary to construct a
string a;          / /tree
int t[N] [26],   / /array transitions (state letter)
    l[N],       / /left
                and right boundaries of the substring of a,
    r[N],// / corresponding to the edge,
entering the vertex
    p[N],       / /ancestor vertices
    s[N],// / suffix link
    tv =0,      / /peak current suffix (if we middle rib
then the lower vertex of
the edge)
                position on the line corresponding to the location on
    tp =0,      / /the edge (from l
[Tv] to r [tv] inclusive)
    ts =2;      / /the number of vertices

void ukkadd(int c) { / / to add to the tree symbol c
    suff:;      / / Will come here after each transition to
suffix (and re-enhance character)
    if (r[tv]<Tp) { / / check we did not come out beyond the current edge
        // if you got out, we find the following edge. If it is not
- create a leaf from a tree and trailers
        if (t[tv] [c]== -1) {t[tv] [c]= Ts; l[ts]= La-1; P[ts +
+]= Tv; tv = s [tv]; Tp = r[tv]+1;goto suff;}
        tv = t[tv] [c]; Tp = l[tv];/ / otherwise just go to the
next edge
    }
    if (tp == -1 || c == a[tp]) tp ++; else { / / if the letter is on the
edge coincides
c c then go to the edge, but otherwise
        // share an edge by two. Middle - top ts l[ts]=
        L[tv]; R[ts]= Tp-1; P[ts]= P[tv]; T[ts] [a[tp]]=
        Tv;
        // sheet set ts +1. It corresponds to the transition
        to c. t[ts] [c]= Ts +1; L[ts +1]= La-1; P[ts +1]=
        Ts;
        // update the parameters of the current vertex. Do not forget
transition from an ancestor to the tv ts. l[tv]= Tp; p[tv]=
        Ts; t[p[ts]] [a[l[ts]]]= Ts; ts + =2;
        / / Prepare for the descent: up to the edge and
went on suffix link.
        // tp will celebrate where we are in the current
        suffix. tv = s[p[ts-2]]; Tp = l[ts-2];
        // until the current suffix is not over, stamped down
        while (tp <= r[ts-2]) {tv = t[tv] [a[tp]]; Tp += r[tv]-
        L[tv]+1;}

```

```
    / / if we come to the top, then put it
suffix link otherwise assign ts
    / / (After all, on the trail. Iteration we create ts).
    if (tp == r[ts-2]+1) s[ts-2]= Tv; else s[ts-2]= Ts;
    / / tp set on a new edge and go to add the letter
the suffix.
    tp = r[tv]-(tp-r[ts-2])+2; goto suff;
}
}
```

```

void build() { fill(r, r
+ N, INF);
// initialize data for the root of the
tree s[0]=1;
l[0]= -
1; r[0]=
-1;
l[1]= -
1; r[1]=
-1;
memset (t, -1, sizeof t);
fill(t[1], T[1]+26, 0);
// add text in a tree, one letter for (int
i =0; i <a.size(); ++ I)
ukkadd (a[i]-'A');
}

```

Problem in online judges

Problems that can be solved by using a suffix tree:

[UVA # 10679 "I Love Strings!!!" \[Difficulty: average\]](#) char,

```

int> next;
node (int l = 0, int r = 0, int par = -1): l (l), r (r), par (par), link (-1) {} int len () {return r - l ; }
Int

```

Search all tandem repeats in a row.

Algorithm Maine-Lorentz

Given a string s length n .

Tandem repeats (Tandem repeat) it called two occurrences of a substring in a row. In other words, a pair of tandem repeats is described indices $i < j$ such that substring - is two identical rows recorded in a row.

$s[i \dots j]$

The challenge is to **find all tandem repeats**. Simplified versions of this problem: find **any** tandem repeat or find **longest** tandem repeat.

Note. To avoid confusion, all rows in the article, we will assume 0-indexed, ie the first character has an index of 0.

Algorithm described here was published in 1982 and Maine Lorenz (see References).

Example

Consider the example of tandem repeats some simple string, for example:

"acababaaee"

This line contains the following tandem repeats:

- $[2; 5] = "abab"$
- $[3; 6] = "baba"$
- $[7; 8] = "ee"$

Another example:

"abaaba"

There are only two tandem repeats:

- $[0; 5] = "abaaba"$
- $[2; 3] = "aa"$

The number of tandem repeats

Generally speaking, the tandem repeats in a string of length n may be of the order $O(n^2)$.

An obvious example is a string composed of n identical letters - in the row tandem repetition is any substring of even length, which roughly. In general, any string is periodic short period will contain a lot of tandem repeats

On the other hand, by itself this fact does not preclude the existence of an algorithm with the asymptotic $O(n \log n)$, because the algorithm can produce tandem repeats in some compressed form groups several pieces at once.

Moreover, there is the notion **series** - Quadruple numbers that describe a whole group of periodic substrings. It has been proved that the number of runs in each row with respect to the linear length of the line.

However, the algorithm described below does not use the concept of the series, so we will not detail this concept.

We present here some other interesting results related to the number of tandem repeats:

- It is known that if we consider only the primitive tandem repeats (ie, those halves which are not multiples of rows), their number in any row - $O(n \log n)$.
- If encode tandem repeats triples of numbers (called triples Krochemora (Crochemore))
(Wherein
 i - Starting position, p - The length of the repeating substring r - The number of repeats), all tandem repeats Any string can be displayed with $O(n \log n)$ such triples. (This is the result obtained by Krochemora output of the algorithm for finding all tandem repeats.) •

(i, p, r)

Fibostrings defined as follows:

$$t_0 = b,$$

$$\begin{aligned}t_1 &= a, \\t_i &= t_{i-1} + t_{i-2},\end{aligned}$$

are "strongly" periodical.

The number of tandem repeats in i th row Fibonacci length even compressed with triples

Krochemora is $O(f_n \log f_n)$.

The number of primitive tandem repeats in the Fibonacci lines - is also of order $O(f_n \log f_n)$.

Algorithm Maine-Lorentz

Idea of the algorithm Maine-Lorentz is fairly standard: an algorithm "**Divide-and-conquer**".

Briefly it is that the original string is split in half, the solution starts from each of the two halves separately (thus we find all tandem repeats, which are located only in the first or only in the second half). Next comes the hardest part - is finding tandem repeats, beginning in the first half and the second ending (we call these tandem repeats for convenience **crossing**). How it's done - and is the very essence of the algorithm Maine-Lorentz; that we will describe below.

Asymptotics algorithm "divide-and-conquer" well researched. In particular, it is important for us that if we learn to look for crossing the tandem repeats in a string length for n $O(n)^{\text{all}}$ algorithm will $O(n \log n)$. The final asymptotic behavior of

Search crossing tandem repeats

Thus, the algorithm Maine-Lorentz came down to that for a given line s learn how to search for all tandem repeats crossing, ie those that begin in the first half of the line, and an end - in the second.

We denote u and v line halves s :

$$s = u + v$$

(Length approximately equal to their length string s Divided in half).

Right and left of tandem repeats

Consider an arbitrary tandem repeat and look at its middle symbol (more on that symbol, which begins the second half of the tandem, ie if the tandem repeat - is the substring, then

$$s[i \dots j]$$

average symbol will $(i + j + 1)/2$.

Then called tandem repeat **left or right** depending on where this symbol -
↳ line u or string v . (You can say so: tandem repeat is called the left if most of it is
↳ left half of the line s ; otherwise - tandem repeat is called the right.)

Learn to look **all left tandem repeats**; right for all will be similar.

Central position tandem repeat

$cntr$

We denote the length of the desired cross tandem repeat through (ie the length of each half of the tandem repeat - this). Consider the first character of tandem repeat entering the line (he stands in a row in

position k). It coincides with the character standing v s
position $length(u)$ on k positions ahead of him; denote this position

through $cntr$.

View all tandem repeats, we will, turning this position $cntr$: ie
we first find all tandem repeats with one value $cntr$, Then another value, etc. - Going through all the
possible values $cntr$ from 0 to $length(u) - 1$.

For example, consider the following line:

(Pipe character separates the two halves

$$u \mid v$$

Tandem repeat **"caca"** Contained in this row will be found when we view
value $cntr = 1$ - Because it is in a position 1 Cost character 'a', coinciding with the first character
tandem repeat themselves in half v .

Criterion for the presence of tandem repeat with a given center $cntr$

So, we must learn to fixed values $cntr$ quickly search for all tandem repeats,
corresponding to it.

Obtain such a scheme (for an abstract line, which contains a tandem repeat "abcabc")

Here and $cntr$ we denote the length of the two pieces of tandem repeat:
through $l_1 l_2$ An - The length of the tandem repeat
to position $cntr - 1$ l_2 - The length of the tandem repeat of l_1 until the end of the halves of the tandem
Repeat. Thus, $l_1 + l_2 + l_1 + l_2$ - The length of tandem repeat.

Looking at this picture, we can see that **necessary and sufficient** the condition that the center at position $cntr$ is the length of the tandem repeat $2l = 2(l_1 + l_2) = 2(\text{length}(u) - cntr)$ is the following condition:

- Let k_1 - This is the largest number such that k_1 characters before your $cntr$ coincide with the latest k_1 character string u :

characters starting from the

- Let k_2 - This is the largest number such that k_2 position $cntr$ coincide with the first k_2 character string v :

- Then it must be true:

This criterion can be **reformulate** thus. Fix a specific value $cntr$. Then:

- All tandem repeats, which we now discover, will have a length $2l = 2(\text{length}(u) - cntr)$. However, these tandem repeats can be **several** It all depends on the choice of the lengths of the pieces l_1 and $l_2 = l - l_1$.
- Find k_1 and k_2 As described above.
- Then will be suitable tandem repeats, for which the lengths of the pieces l_1 and l_2 satisfy the following conditions:

$$\begin{cases} l_1 + l_2 = l = \text{length}(u) - cntr, \\ l_1 \leq k_1, \\ l_2 \leq k_2. \end{cases}$$

Algorithm for finding the lengths k_1 and k_2

So the whole problem is reduced to the rapid calculation of the lengths k_1 and k_2

for each value of $cntr$. Recall their definitions:

- k_1 - Maximal non-negative integer for which holds:

$$u[cntr - k_1 \dots cntr - 1] == u[\text{length}(u) - k_1 \dots \text{length}(u) - 1].$$

- k_2 - Maximal non-negative integer for which holds:

Both of these can be responsible for query

$O(1)$ Using **algorithm for finding Z-function**:

- To quickly find the values k_1 pre-calculate Z-function string \bar{u} (ie lines written out in the reverse order).

Then the value for a particular $cntr$ is just equal to the corresponding value of the array Z-function.

- To quickly find the values k_2 pre-calculate Z-function string $v + \# + u$ (ie line u Attributed to the line through the separator character).

Again, the value of k_2 specific $cntr$ will have to just take the corresponding element of Z-function.

Find the right tandem repeats

Up to this point we have only worked with the left tandem repeats.

To look for the right tandem repeats, act the same way: we define the center $cntr$ as symbol corresponding to the last character of a tandem repeat, who got into the first row.

Then the length will be defined as the greatest number of characters to the position inclusive, coinciding with the last character. Length is defined as the maximum number of characters starting with . Coinciding with the first character.

$cntr + 1$ an it will be necessary to calculate in advance Z-function for a_n . Thus, you can quickly find k_1 strings v respectively. Then, turning over a specific value $cntr$ We at the same criterion will finding all the right tandem repeats.

Asymptotics

Asymptotika algorithm Maine-Lorentz be so : Since this algorithm is an algorithm of "divide-and-conquer", each recursive invocations which works in time, linear in the length of the string: for four lines in linear time sought their Z-function And then value moves and displays all groups detected tandem repeats.

Tandem repeats detected algorithm Maine-Lorentz form of original **groups**: Such

quadrupes $(cntr, l, k_1, k_2)$, Each of which represents a group of tandem repeats of a length centered with all sorts of lengths pieces l_1 and l_2 Satisfying the conditions:

$$\begin{cases} l_1 + l_2 = l, \\ l_1 \leq k_1, \\ l_2 \leq k_2. \end{cases}$$

Implementation

We present implementation of the algorithm Maine-Lorentz, during which finds all tandem repeats this line in a compressed form (as a group described by four numbers).

In order to demonstrate the tandem repeats found during $O(n^2)$ "Decompress" and displayed on separately. This conclusion is in solving real-world problems will be easily replaced by some other, more effective actions, for example, search of the longest tandem repeat, or count the number of tandem repeats.

```

vector<int> Z_function (const string & s)
{
    int n = (int) s.length(); vector<int> Z (n);
    for (int i = 1, L = 0, R = 0; i < n; +
        + I) { if (i <= r)
        z[i] = Min (r-i + 1, z[i-1]);
        while (i + z[i] < N && s[z[i]] == s[i +
            z[i]]) ++ Z[i];
        if (i + z[i]-1 > R)
            l = i, r = i + z[i]-1;
    }
    return z;
}

void output_tandem (const string & s, int shift, bool left, int cntr, int
l, int l1, int l2) {
    int pos;
    if (left)
        pos = cntr-l1;
    else
        pos = cntr-l1-l2+l1 +1;
    cout << "[" << Shift + pos << ".." << Shift + pos +2* l-1 << "] =" << s.substr (pos, 2* l) << endl;
}

void output_tandems (const string & s, int shift, bool left, int cntr, int
l, int k1, int k2) {
    for (int l1 =1; l1 <= l; + + L1) {
        if (left && l1 == l) break;
        if (l1 <= k1 && l-l1 <= k2)
            output_tandem (s, shift, left, cntr, l, l1, l-l1);
    }
}

```



```

inline int get_z (const vector <int> & z, int i) {
    return 0<= i && i <(int)z.size() ? z[i] :
    0;
}

void find_tandems (string s, int shift = 0) {
    int n = (int) s.length();
    if (n == 1) return;
    int nu = n /2, Nv = n-nu;
    string u = s.substr (0, Nu),
           v = s.substr (nu);
    string ru = string (u.rbegin(), U.rend()),
           Rv = string (v.rbegin(), V.rend());
    find_tandems (u, shift);
    find_tandems (v, shift + nu);
    vector <int> Z1 = z_function (ru),
            z2 = z_function (v + '#' + U),
            Z3 = z_function (ru + '#' + Rv),
            Z4 = z_function (v);
    for (int cntr =0; cntr <n; ++ Cntr) {
        int l, k1, k2;
        if (cntr <nu) {
            l = nu - cntr;
            k1 = get_z (z1, nu-cntr);
            k2 = get_z (z2, nv +l+ Cntr);
        }
        else {
            l = cntr - nu + 1;
            k1 = get_z (z3, nu +1 + Nv-1-(cntr-nu));
            k2 = get_z (z4, (cntr-nu)+1);
        }
        if (k1 + k2> = l)
            output_tandems (s, shift, cntr <nu, cntr, l, k1, k2);
    }
}

```

Literature

- Michael Main, Richard J. Lorentz. **An O (n log n) Algorithm for Finding All Repetitions in a String** [1982]
- Bill Smyth. **Computing Patterns in Strings** [2003]
- Bill Smith. **Methods and algorithms for computing row** [2006]

Search substring in a string using a Z-or-prefix function

Two strings S and T. required to find all occurrences of the string in the text of S T for O (N), where N - the total length of strings S and T.

Algorithm

Form the string **S \$ T** Where \$ - a separator which is not found in S, or T. in

We compute for the resulting string **prefix function** P for O (N). Walk through the array P, and consider all the elements that are equal to | S | (length S). By definition prefix certain functions, which means that in this place ending substring that matched | S |, ie the desired entry. Thus, we find all occurrences. This algorithm is called an algorithm **ILC (Knuth-Morris-Pratt)**.

Now solve the same problem using **Z-function**. Construct in O (N) array Z - Z-line function S \$ T. Go through its elements, and consider those that are equal to | S |. By definition, in this place starts the substring that matched S. Thus, we find all occurrences.

Solution of the problem "row compression" for O (N)

S. A string is required to find such a string T, the string S is obtained by rote T. Of all the possible T to choose the smallest length.

This task is very simple to solve for O (N) using the prefix function.

So, let the array P - prefix-row function S, which can be computed in O (N).

Now consider the last element P: P [N-1]. If N is divisible by (N - P [N-1]), then the answer is there, and it is N - P [N-1] the first letters of the string S. If not divisible, then there is no answer.

The correctness of this method is easy to understand. P [N-1] is equal to the length of the longest line of its own suffix S, coincides with the prefix S. If there is a response, it is evident that the initial bit string S length (N - P [N-1]) is the response, and therefore, N is divided into (N - P [N-1]). If there is no answer, the (N - P [N-1]) is equal to some inexplicable value to which N will not share (otherwise the answer existed).

Implementation

```
int n = (int) s.length
(); vector <int> p (n);
// ... Here prefix computation function ...
int l = n - p [n-
1]; if (n% l == 0)
    cout << s.substr (l);
else
    cout << "No Solution";
```



Sqrt-decomposition

Sqrt-decomposition - a method, or a data structure that allows you to perform some common operations

(summation subarray elements, finding the minimum / maximum, etc.) for that

$$O(\sqrt{n})$$

considerably faster than $O(n)$ for the trivial algorithm.

First we describe the data structure for one of the simplest applications of this idea, then show how to generalize it to solve

some other problems, and finally, consider a slightly different application of this idea:

splitting the input requests sqrt-blocks.

The data structure on the basis of decomposition sqrt-

We pose the problem. Given an array $a[0 : n - 1]$. Required to implement such a data structure, which be able to find the sum of elements $a[l \dots r]$ for arbitrary l and r for $O(\sqrt{n})$ operations.

Description

The basic idea sqrt-decomposition is that we will do next **predposchet**: Divide the a into blocks of length approximately

\sqrt{n} . And in each block i predposchitaem advance amount $b[i]$ elements therein.

We can assume that the length of one block and the number of blocks equal to the same number - the square root of n . Rounded up:

$$s = \lceil \sqrt{n} \rceil,$$

then the array $a[]$ roughly divided into blocks as follows:

$$\underbrace{a[0] \ a[1] \ \dots \ a[s-1]}_{b[0]} \ \underbrace{a[s] \ a[s+1] \ \dots \ a[2 \cdot s-1]}_{b[1]} \ \dots \ \underbrace{a[(s-1) \cdot s] \ \dots \ a[n]}_{b[s-1]}.$$

Although the last block may be less than s , Components (if n not divided into s) - It does not matter.

Thus, for each block k we know the amount of it $b[k]$.

So, let these values b_k previously counted (to do this, obviously, $O(n)$ operations). That they can to the calculation of response to another request (l, r) ? Note that if the segment $[l; r]$ long, then it will contain several blocks entirely, and these blocks we can find out the amount to them in a single operation. As a result, the total length of two blocks will only entering it only partially, and these lumps us will produce a trivial summation algorithm.

Illustration (here k denotes block number, which is l And through p - Number of a block in which is r)

This figure shows that in order to calculate the amount of segment $[l \dots r]$, It is necessary to sum items only two "tails": $[l \dots (k+1) \cdot s - 1]$ and $[p \cdot s \dots r]$ And sum the values $b[i]$ in all blocks, starting with $+1$ and ending $p-1$:

(Note: this formula is incorrect when: $p < k$, In this case, some items are summed twice, in which case you just have to sum elements l by r)

Thus we have a significant amount of ekonomim operations. Indeed, the size of each of the "tails" are obviously not exceed the length of the block s And also the number of blocks does not exceed s . Since $s \approx \sqrt{n}$. Then

to calculate the sum of all the segment $[l \dots r]$ we need only $O(\sqrt{n})$ operations.

Implementation

We first present a simple implementation:

```
// input int n;
vector <int> A
(n);
// predposchet
int len = (int) sqrt (n + .0) + 1; // and a block size and the number of
blocks vector <int> B (len);
for (int i = 0; i <n; ++ I)
    b[i / len] += 
        A[i];
// Response to
requests for (,;) {
    int l, r; // reads input - another request int sum = 0;
    for (int i = l; i <= r; )
        if (i % len == 0 && I + len - 1 <= R) {
            // If i indicates the beginning of the block lying entirely
in [l; r]
            sum += b[i /
                len]; i += len;
        }
        else {
            sum += 
                a[i]; ++ I;
        }
}
```

The disadvantage of this implementation is that it unreasonably long division operations (which are known to run significantly slower than the other operations). Instead, you can count the number of blocks c_l and c_r , which lie in the boundary and r respectively, and then make a loop with the blocks on, $c_l + 1 \dots c_r - 1$ separately processed "tails" in the blocks c_l and c_r . Furthermore, in case of this implementation $c_l = c_r$ becomes specific and requires a separate processing:

```
int sum = 0;
int c_l = l / len, c_r = r / len;
if (c_l == c_r)
    for (int i = l; i <= r;
        ++ I) sum += 
            a[i];
else {
    for (int i = l, end = (c_l +1)* Len-1; i <=
        end; ++ I) sum += a[i];
    for (int i = c_l +1; i <= c_r-1;
        ++ I) sum += b[i];
    for (int i = c_r * len; i <=
        r; ++ I) sum += 
            a[i];
}
```

Other problems

We have examined the problem of finding the sum of array elements in some of its subsegments. This task can expand a bit: also solve **change** individual array elements. Indeed, if

changing an element, it is sufficient to update the value of $b[k]$ in the block in which this element is ($k = i/\text{len}$)

On the other hand, instead of the amount of similar problems can be solved the problem of **minimum**, **maximum** elements in the interval. If these problems to allow changes of individual elements, that too will need to recalculate the value of the unit, which owns variable element, but recalculate already fully pass on all elements of the unit for $O(\text{len}) = O(\sqrt{n})$ operations.

Similarly sqrt-decomposition can be applied to the set **other** similar problems:

finding the number of zero elements, the first non-zero element, counting the number of certain elements, etc.

There are a class of problems occur when **change elements on the whole subsegments**: addition or appropriation of elements of the array at some subsegments A .

For example, you must perform the following two types of queries added to all elements of a certain length

value, and recognizing the value of an individual item . Then, as Put that amount b_k which must be added to all members of the second block (for example, initially all b_k); then when the query is "addition" will need to perform the addition of all elements $b_k = 0$ Tails", and a_i . And the

then perform the addition of all elements b_i for blocks lying entirely in the interval $[l \dots r]$ answer the second database is likely to be a $a_i + b_k$ Where i / len . Thus, adding the interval will be performed for $O(\sqrt{n})$ And the request is a separate element - for $O(1)$.

Finally, you can use both types of tasks: changing elements on the interval and response to requests for the same interval. Both types of operations will be carried out for. To do this already will have to do two "block" array b and c :

one - to secure changes in the interval, the other - to answer queries.

Can give an example, and other tasks to which you can apply sqrt-decomposition. For example, we can solve the problem of **maintaining a set of numbers** with the ability to add / remove numbers, check the numbers on the accessory set, search the order number. To solve this problem it is necessary to store numbers in sorted order, separated by a few blocks of numbers in each. When adding or

removing the need to produce a number of "rebalancing" blocks, throwing the number of start / end of some units in the beginning / end of the neighboring blocks.

Sqrt-decomposition of the input query

Consider now a completely different ideas about the use of sqrt-decomposition.

Suppose that we have some problem in which we are given some input data, and then enter commands / queries, each of which we need to give treat and give the answer. We consider the case

when requests are requesting as (do not change the system state, but only requesting some information) and modifying (ie affecting the state of the system originally specified input data).

Specific task can be quite daunting, and "honor" of her decision (which reads one request, processes it, changing the state of the system, and returns the response) can be technically challenging or even be unaffordable for the final. On the other hand, the decision "off-line" embodiment problem, i.e. when there are no modifying operations, and there are only requesting queries - often

much easier. Suppose we **know how to solve "Offline" option** problem, i.e. build some time some data structure that can answer queries, but can not process modifying queries.

Then **divide the input requests for blocks** (How long - is not specified, we denote this length by s). At the beginning of each processing unit will be in a data structure to build "Off-line" version of the problem as the data at the beginning of this block.

Now we take turns taking requests from the current block and handle each of them. If the current request

- Modifying, then skip it. If the current request - requesting, then turn to the data structure for the offline version of the problem, but the pre- **taking into account all modifying queries in the current block**. Incorporation of such a modifying queries is not always possible, and

it should be fast enough - during $O(s)$ or a little worse; denote this time via $Q(s)$.

Thus, if we just requests that need to process them time.

Value m should be selected based on the specific type of functions and . For example, if $B(m) = O(m)$ and $Q(s) = O(s)$, The best choice would be $s \approx \sqrt{m}$ And turn $B(m) = O(m\sqrt{m})$

Since the above arguments are too abstract, we give some examples of problems to which this applies sqrt-decomposition.

Example task: Adding the interval

Condition of the problem: given an array of numbers $a[1 \dots n]$ meaning in element of the array, and add a number for each array element in an interval $a[l \dots r]$.

Although this problem can be solved without this technique with the division requests for blocks, we present it here - as a simple and intuitive application of this method.

So, we divide the input into blocks of questions (where m - Number of requests). At the beginning of the first block any request to build structures is not necessary, just keep an array. Come on now at the request of the first block. If the current request - request addition, it is still missing it. If the current request - the request to read the value

in some position i . Then just take the first as a response value $a[i]$. Then go through the all missed in this block requests the addition, and for those of them, which gets i , Apply them to the current increase in response.

Thus, we have learned to respond to requests for requesting time $O(\sqrt{m})$.

It remains only to note that at the end of each block requests, we need to apply all modifying requests for this unit to an array $a[]$. But it's easy to make for $O(n)$ - Enough for each request adding (l, r, x) noted in the sub-array at the point number l , and at the point $r + 1$ - Number x . And then go through this array, adding the current amount to an array $a[]$.

Thus, the final asymptotic solution will be $O(\sqrt{m}(n + m))$.

Example problem: disjoint-set-union division

Undirected graph with n peaks m ribs. Receives requests three types: add an edge (x_i, y_i) , Remove rib (x_i, y_i) And check whether or not connected vertices x_i and y_i means.

If removal requests absent, the solution of the problem would have been well-known data structure [disjoint-set-union](#) (the system of disjoint sets). However, if a remote task is much more complicated.

Made in the following manner. At the beginning of each block queries see which edges in this block will be removed and immediately **remove** Count them out. Now we construct a system of disjoint sets (dsu) on the resulting graph.

As we now have to respond to another request from the current block? Our system of disjoint sets "knows" all the edges, except those that are added / removed in the current block. However, removal of dsu we do not need - we have removed all such advance edges of the graph. Thus, all that may be - is more, add the ribs, which can be a maximum \sqrt{m} pieces.

Therefore, in response to the current request requesting we can just keep the bypass wide at the connected components dsu, that will work for, as we are only in the consideration $O(\sqrt{m})$ Ribs.

Off-line tasks to requests for subsegments and versatile array sqrt-heuristics for them

Consider another interesting variation ideas sqrt-decomposition.

Suppose we have some problem in which there is an array of numbers, and there are requesting queries of the form (l, r) to learn something about subsegments. We believe that the request is modified, and are known us in advance, ie task - the offline.

Finally, we introduce the latest **restriction**. We believe that we can quickly recalculate prompted when the left or right border of the unit. Ie if we knew the answer to the request, then quickly be able to calculate the answer to a query $(l + 1, r)$ or $(l - 1, r)$ or $(l, r + 1)$ or $(l, r - 1)$. We now describe **universal heuristics** for all such tasks. Sort the requests

pair: $(l \text{ div } \sqrt{n}, r)$. Ie we sorted requests to the number sqrt-block in which lies the left end and being equal - on the right end.

Consider now the group queries with the same value $l \text{ div } \sqrt{n}$ and see how we can handle it. All such requests we ordered at the right margin, which means that we can just start with a blank portion $[l; l - 1]$. And repeatedly incrementing the right border - in the end to answer all these questions.

Good **example** for this heuristic is such a task: find the number of different numbers in the segment array $[l; r]$. This problem is difficult to solve by classical methods.

Slightly more complicated version of this problem is [problem with one of the rounds Codeforces](#).

Fenwick tree

Fenwick tree - a data structure, tree on the array, having the following properties:

- 1) allows to calculate the value of a reversible operation G on any interval [L; R] during **O (log N)**;
- 2) allows you to change the value of any element in **O (log N)**;
- 3) requires **O (N) Memory** or, more precisely, exactly the same as and array of N elements;
- 4) easily generalized to the case of multidimensional arrays.

The most common use of wood Fenwick - to calculate the amount of the interval, ie function $G(X_1, \dots, X_k) = X_1 + \dots + X_k$.

Fenwick tree was first described in the article "A new data structure for cumulative frequency tables" (Peter M. Fenwick, 1994).

Description

For ease of description, we assume that the operation of G, on which we build the tree - it **amount**.

Suppose we are given an array A [0 .. N-1]. Wood Fenwick - array T[0 .. N-1], in which each element is stored programming of certain elements of the array A:

T_i = sum A_j all F (i) <= j <= i,

where F (i) - a function that we will define later.

Now we can write **pseudocode** function for computing the sum of the interval [0; R] for the function of the cell changes:

```
int sum (int r)
{
    int result = 0;
    while (r >= 0) {
        result += t[r];
        r = f(r) - 1;
    }
    return result;
}

void inc (int i, int delta)
{
    for all j, for which F (j) <= i <= j
    {
        t[j] += delta;
    }
}
```

The sum function works as follows. Instead of going over all the elements of the array A, it moves the array of T, making the "jumping" through the segments where possible. First, it adds to the response value of the sum on the interval [F (R); R], then takes the sum on the interval [F (F (R) -1); F (R) -1], and so on, until it reaches zero.

Function inc moves in the opposite direction - in the direction of increasing indices, updating sum value T_j only for those products for which it is necessary, ie for all j, for which F (j) <= i <= j.

Obviously, the choice of the function F is dependent on both the speed of the operations. Now we consider the function that will achieve logarithmic performance in both cases.

Determine the value of F (X) follows. Consider the binary representation of the number and look at the its least significant bit. If it is zero, then F (X) = X. Otherwise, the binary representation of X ends with group one or more units. Replace all the units of the group for the zeros, and assign the resulting number value of the function F (X).

This corresponds to a rather complicated description of a very simple formula:

F (X) = X & (X +1),

where k - is the bitwise logical "AND".

It is easy to see that this formula corresponds to the verbal description function given above.

We just need to learn how to quickly find such number j , for which $F(j) \leq i \leq j$.

However, it is not difficult to make sure that all such numbers are obtained from i j successive replacements of the right (the youngest) zero in the binary representation. For example, for $i = 10$, we find that $j = 11, 15, 31, 63$, etc. Oddly enough, this operation (replacement youngest zero to one) also corresponds to a very simple formula:

$H(x) = x | (x + 1)$,

where $|$ - is the bitwise logical "OR".

Implementing tree Fenwick for the sum-dimensional case

```
vector <int>
t; int n;
void init (int nn)
{
    n = nn; t.assign
    (n, 0);
}

int sum (int r)
{
    int result = 0;
    for (; r >= 0; r = (r & (r + 1)) -
        1) result += t [r];
    return result;
}

void inc (int i, int delta)
{
    for (; i < n; i = (i | (i + 1)))
        t [i] += delta;
}

int sum (int l, int r)
{
    return sum (r) - sum (l - 1);
}

void init (vector <int> a)
{
    init ((int) a.size ());
    for (unsigned i = 0; i < a.size (); i +
        +) inc (i, a [i]);
}
```

Implementing tree Fenwick for a minimum one-dimensional case

It should immediately be noted that, as the tree Fenwick allows you to find the function value at an arbitrary interval $[0; R]$, then we will not be able to find at least on the interval $[L; R]$, where $L > 0$. Further, all values change should take place only in the direction of decreasing (again, does not work because the pay function min).

This is a significant limitation.

```
vector <int>
t; int n;
const int INF = 1000 * 1000 * 1000;
void init (int nn)
```

```

{
    n = nn;
    t.assign (n, INF);
}

int getmin (int r)
{
    int result = INF;
    for (; r >= 0; r = (r & (r +1)) - 1)
        result = min (result, t
                      [r]);
    return result;
}

void update (int i, int new_val)
{
    for (; i <n; i = (i | (i +1)))
        t [i] = min (t [i], new_val);
}

void init (vector <int> a)
{
    init ((int) a.size ());
    for (unsigned i = 0; i <a.size (); i +
        +) update (i, a [i]);
}

```

Implementing tree Fenwick for the sum for the two-dimensional case

As already noted, Fenwick tree is easily generalized to the multidimensional case.

```

vector <vector <int>> t;
int n, m;
int sum (int x, int y)
{
    int result = 0;
    for (int i = x; i >= 0; i = (i & (i +1)) - 1)
        for (int j = y; j >= 0; j = (j & (j +1)) -
            1) result += t [i] [j];
    return result;
}

void inc (int x, int y, int delta)
{
    for (int i = x; i <n; i = (i | (i +1)))
        for (int j = y; j <m; j = (j | (j +1)))
            t [i] [j] += delta;
}

```

System of disjoint sets

This article examines the data structure "A system of disjoint sets" (English "Disjoint-set-union", , or simply "DSU").

This data structure provides the following features. Initially, there are several elements, each of which is located in a separate (its own set). In one operation can **combine any two sets**, and can also **request, in which the set** now is specified item. Also, in the classic version, introduced another operation - creating a new item, which is placed in a separate set.

Thus, the base interface of the data structure consists of only three operations:

- **make_set(x)** - adds new item x , Placing it in a new set consisting of one him.
- **union_sets(x, y)** - combines these two sets (the set containing the element x and set, in which the element is used). y
- **find_set(x)** - returns in which a set of is specified item. Actually x This is returned when one of the plurality of elements (referred to as **representative** or **leader** (In English literature "leader")). This representative is selected in each set of the very structure of the data (And may change over time, namely, after the call **union_sets()**).

For example, if a call **find_set()** for any two elements of one and returned to the same value, it means that these elements are in one and the same set, and otherwise - to the different sets.

With the following data structure allows each of these operations in nearly average $O(1)$ (More details on the asymptotics below after the description of the algorithm).

Also, in one of the subsections of the article describes an alternative embodiment of the DSU, allowing obtain the asymptotic behavior $O(\log n)$ an average of one request when $m \geq n$; when $m \gg n$ (i.e., m considerably more n) - And at the time $O(1)$ averaging the request (see "Storage DSU as an explicit list sets").

Building an efficient data structure

We first define the form in which we will store all the information.

Set of elements we will store in the form of **trees**: One tree corresponds to one set. Root of the tree - is representative (leader) of the set. parent

When implemented, this means that we lead an array in which each element we store a reference to its parent in the tree. For the roots of trees, we assume that their ancestor - they (ie link loops in this place).

Naive implementation

We can already write the first implementation of a system of disjoint sets. It will be quite inefficient, but then we will improve it with two receptions, receiving a result of almost constant work.

So, all the information about the sets of elements stored with us using the array **parent**.

To create a new item (operation **make_set(v)**), We simply create a tree rooted at the top, v . Noting that her father - it herself.

To combine the two sets (operation **union_sets(a, b)**), We first find the set of leaders in which is a And set in which is located. If leaders coincide, then do not do anything - it means that the sets already been merged. Otherwise, you can simply indicate that the ancestor of a vertex is a (Or vice versa) - thereby attaching one tree to another.

Finally, the implementation of the search operation **leader (v)** Is simple: we ascend from the top of the ancestors v , until we reach the root, ie while the reference to the ancestor is not a. This operation is easier to implement recursively (especially it will be convenient later, in connection with optimizations being added).

```
void make_set (int v) {
    parent[v] = v;
}

int find_set (int v) {
    if (v == parent[v])
```

```

        return v;
    return find_set (parent[v]);
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set
    (b); if (a! = b)
        parent[b] = A;
}

```

However, such an implementation of disjoint sets very **ineffective**. It is easy to construct an example, when, after several unions of sets get a situation that many - this tree, degenerated into a long chain. As a result, each call will work in this assay

`find_set()`

time order of the depth of the tree, ie for $O(n)$.

This is very far from that of the asymptotics, which we were going to get (constant time).

Therefore, we consider two optimizations that allow (even applied separately) significantly improved performance.

Heuristics compression path

This heuristic is designed to speed up `find_set()`.

It lies in the fact that when we find after calling the desired leader p set, then remember that at the top v and all passed on the way peaks - this is the leader. This is easiest

make forwarding them `parent[]` this peak p .

Thus, the array of ancestors `parent[]` meaning is somewhat different: it is now **compressed array Ancestors**, ie for each vertex there can be stored for immediate ancestor, and ancestor ancestor ancestor parent parent, etc.

On the other hand, it is clear that it is impossible to have these indexes at step `parent` would update the leaders in

always pointed to the leader:
otherwise
 $O(n)$ elements.
partially

Thus, the array `parent[]` it should be treated as an array of ancestors may compressed.

The new implementation of operations

`find_set()` as follows:

```

int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
}

```

This simple implementation does all that meant: first, by the recursive call is the leader of the set, and then, during stack unwinding, this leader is assigned to all the links traversed elements.

Implement this operation and can be non-recursive, but then have to perform two passes on a tree: first find the desired leader, the second - it would put all the vertices of the path. However, in practice, a non-recursive implementation of no significant payoff.

Rating asymptotics when applying heuristics compression path

We show that the use of a heuristic path compression achieves logarithmic

asymptotics $O(\log n)$ per request, on average.

Note that, because the operation is a two call operations and more operations, then we can focus only $O(1)$ on the proof of assessment Time $O(m)$ operations `find_set()`.

We call **weight** vertices v the number of children of this vertex (including herself). Vertex weights, obviously, can only increase during the algorithm.

We call **scale edges** the difference between the weights of all edges $|w[a] - w[b]|$ (Apparently tops ancestor weight is always greater than that of the top of the descendant). It can be noted that the scope of a fixed rib (a, b) can only increase during the algorithm.

In addition, we split the rib **classes**: We say that the edge has class k if it belongs to the segment sweep $[2^k; 2^{k+1} - 1]$. Thus, the class edges - is a number from 0 to $\lceil \log n \rceil$.

We now fix an arbitrary vertex \mathcal{C} and will monitor how the edge in its ancestor: first it

absent (until the top x is the leader), and then held an edge of x in some vertex (when the set with vertex x connected to the other set), and then may change during compression paths. It is clear that we are interested in the asymptotic behavior of only the last case (compressed paths):

Call x paths: all other cases require time on one request.

Consider the work of a call operation $O(1)$ He passes along some tree **pathErasing** all edges of this path and redirecting them to the **find_set** leader.

Consider this way and **exclude** from consideration last edge of each class (ie, no more than one edge of the class

$0, 1, \dots, \lceil \log n \rceil$. Thus, we have excluded $O(\log n)$ edges of each request.

We now consider all **other** the edges of this path. For each such edge, if it has class, it turns out that in this way there is one more edge class (otherwise we would be required to eliminate the current edge as the sole representative of the class). Thus, the compression path is replaced by the edge rib class

least $k + 1$. Given that the reduced weight of the edge can not we get that for each vertex, affected by the query **find_path**, An edge in her ancestor was either excluded or severely increased its class.

Hence, we finally obtain the asymptotic behavior of m Queries: $O((n + m) \log n)$ That (at $m \geq n$) Means a logarithmic time per request on average.

Heuristics union by rank

We consider here the other heuristics, which in itself is able to accelerate the running time, and in combination with compression heuristic ways and all capable of achieving almost constant running time per query on average.

union_sets

the number of vertices

In both cases heuristics are one and the same: when the **union_sets** will join tree lower scores to a tree with a large rank.

Show implementation **rank heuristics based on the size of trees**:

```
void make_set (int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (size[a] < size[b])
            swap (a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

Show implementation **rank heuristics based on the depth of trees**:

```
void make_set (int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[b]++;
    }
}
```

Both options are equivalent to the rank of heuristics terms asymptotic however in practice possible to use any of them.

Evaluation of the asymptotics in the application of rank heuristics

We show that the asymptotic behavior of the system of disjoint sets using only rank heuristics, heuristics without compression paths will **logarithmic** per request, on average: $O(\log n)$.

Here we show that for any of the two options rank heuristics depth of each tree will be value $O(\log n)$. That will automatically mean logarithmic asymptotics for the request `find_set` and therefore, the request `union_sets`.

Consider the **rank heuristic depth tree**. We show that if the rank of the tree is then k . The tree contains at least 2^k vertices (hence will automatically follow that the rank and, hence, depth of the tree is the value $O(\log n)$). Will prove by induction for $k = 0$ compressing ways depth can only decrease. Rank increases with tree $k - 1$ to k . When it is joined tree rank $k - 1$; applying to these two trees size $k - 1$ induction hypothesis, we get that new tree rank k really will have at least $\frac{2^k}{2^k}$ vertices, as required.

We now consider the **rank heuristic size trees**. We show that if the size of Wood is, its height is not more $\lfloor \log k \rfloor$. Will prove by induction for $k = 1$ statement is true. When compressing ways depth can only decrease, so path compression does not violate anything. Let now combined two tree sizes k_1 and k_2 ; then by the induction of their height is less than or are, respectively, $\lfloor \log k_1 \rfloor$ and $\lfloor \log k_2 \rfloor$. Without loss of generality, we assume that the first tree - more ($k_1 \geq k_2$). So after the unification of the depth of the resulting tree $k_1 + k_2$ vertices will be equal:

$$h = \max(\lfloor \log k_1 \rfloor, 1 + \lfloor \log k_2 \rfloor).$$

To complete the proof, we must show that:

$$\begin{aligned} h &\stackrel{?}{\leq} \lfloor \log(k_1 + k_2) \rfloor, \\ 2^h &= \max(2^{\lfloor \log k_1 \rfloor}, 2^{\lfloor \log k_2 \rfloor}) \stackrel{?}{\leq} 2^{\lfloor \log(k_1 + k_2) \rfloor}, \end{aligned}$$

that there is almost obvious inequality as $k_1 \leq k_1 + k_2$ and $2k_2 \leq k_1 + k_2$.

Combining heuristics: path compression plus rank heuristics

As mentioned above, the combined use of these heuristics gives best results particularly in the end reaching nearly constant running time.

We will not give here the proof of the asymptotic behavior, as it is long enough (eg, Cormen, Leiserson, Rivest, Stein "Algorithms. Construction and Analysis"). First is the proof was held Tarjanne (1975).

The final result is that the joint application of heuristics path compression and union by rank while working on one request is received, on average (where - inverse function Ackermann)

Ackermann , which grows very slowly so slowly that for all reasonable restrictions n it does not exceed 4 (For approximately $n \leq 10^{600}$).

That is why the asymptotic behavior of the system of disjoint sets appropriate to say "almost constant time."

We present here **final implementation of the system of disjoint sets**, implements both of these heuristics (Heuristics used rank relative depths of the trees)

```
void make_set (int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = Find_set (parent[v]);
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set
    (b); if (a != b) {
        if (rank[a] < rank[b])
```

```

        swap (a, b);
parent[b] = A;
if (rank[a] == Rank[b])
    + + Rank[a];
}
}

```

Applications to various problems and improve

In this section we consider some applications of data structures, "a system of disjoint sets" as trivial and some improvements using data structures.

Support component of the graph

This is one of the obvious applications of the data structure "system of disjoint sets", which, apparently, and stimulated the study of the structure.

Formally problem can be formulated as follows: initially given a blank graph, This gradually graph can add vertices and undirected edges, and receives requests - "in the same (a, b) whether the connected components are tops a and b ?".

Here directly applying the above data structure, we obtain a solution that processes a request to add a vertex / edge or request a review of the two peaks - in almost constant time on average.

Given that almost exactly the same problem is formulated using [algorithm Kruskal finding the minimum spanning tree](#), we immediately obtain [an improved version of the algorithm](#), Works with almost linear time.

Sometimes occurs in practice **inverted version of this problem**: Originally a graph with some vertices and edges, and receives requests to remove edges. If the task is given to offline, ie we can know in advance all the questions, then you can solve this problem as follows: turn the problem backwards: we assume that we have an empty graph, which can be added to the edges (first add an edge of the last request, then the penultimate, etc.). Thus, by inverting this task, we came to the usual problem whose solution is described above.

Search the connected components in the image

One of the underlying surface applications DSU is to solve the following problem: there is a picture $n \times m$ pixels. Initially, all white image, but then it draws some black pixels. Required to determine the size of each "white" on the connected components of the final image.

To solve we just iterate through all the white cells of the image, for each cell iterate its four neighbors, and if the neighbor is also white - then call these two vertices. Thus, we will have to DSU nm peaks corresponding to the pixels of the image. The resulting trees eventually DSU - are the desired connected components.

However, such a decision by the system of disjoint sets has no advantages over solution using [traversal depth](#).

Support for additional information for each set

"The system of disjoint sets" makes it easy to store any additional information related to arrays.

A simple example - it **size sets**: How to store them, it was described in the description of rank heuristics (information recorded there for the current leader of the set).

Thus, together with the leader of each set can store any additional required information in a specific problem.

Application DSU compression "jumps" over the interval. The problem of painting Subsegments offline

One common application of DSU is that if there is a set of elements, and each element comes from one edge, then we can quickly (in almost constant time) the final point, which we get if we move along the edges of a given starting point.

A good example of this application is **problem of painting subsegments**: A segment length L . Each cell of which (ie, the length of each piece 1) Has zero numbers. Receives requests form (l, r, c)

- Repaint interval in color c . Required to find the final color of each cell. Requests shall be deemed known beforehand, i.e. task - offline.

For solutions, we can make DSU-structure that for each cell will keep a reference to the nearest right unpainted cage. Thus, initially, each cell indicates itself, and after painting the first subsegments - cell before subsegments will point to the cell after the end of the subsegments.

Now, to solve the problem, we consider requests repainting **reversed** from last to first. To query every time we just use our DSU find the most left unpainted cage inside the interval, repaint it, and we move the pointer from it to the next empty cell to the right.

Thus, here we actually use heuristics DSU with compression paths, but without rank heuristics (because important for us, who will be the leader after the merger). Consequently, the total amount asymptotic $O(n \log n)$ on request (although a small compared to other data structures constant).

Implementation:

```
void init() {
    for (int i = 0; i < L; ++I)
        make_set(i);
}

void process_query (int l, int r, int c) {
    for (int v = l; ; ) {
        v = find_set(v);
        if (v >= r) break;
        answer[v] = C;
        parent[v] = v + 1;
    }
}
```

However, this solution can be implemented **with rank heuristics**: We keep for each set from an array, where the set ends (ie, the right-most point). Then it will be combined two sets to one in their rank heuristics, then affixing received many new right border. Thus, we obtain a solution for $O(\alpha(n))$.

Support distances to the leader

Sometimes in specific applications of disjoint sets pops requirement to maintain the distance to the leader (ie, the path length to the edges in the tree from the current node to the root of the tree).

If there were no compression heuristic ways, then there are no problems would not arise - the distance just to the root would be equal to the number of recursive calls that did function `find_set`.

However, as a result of compression paths several edges of the path could be compressed into a single edge. Thus, with each vertex must store additional information: **length of the current edge from vertex in the ancestor**.

When implementation is convenient to represent the array `parent` and function `find_set` now return more than one number, a pair of numbers: the top leader and the distance to it:

```
void make_set (int v) {
    parent[v] = Make_pair (v, 0);
    rank[v] = 0;
}

pair <int,int> Find_set (int v) {
    if (v != parent[v].First) {
        int len = parent[v].Second;
        parent[v] = Find_set (parent[v].First);
        parent[v].Second += len;
    }
    return parent[v];
}

void union_sets (int a, int b) {
    a = find_set(a).First;
    b = find_set(b).First;
    if (a != b) {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = Make_pair (a, parent[a].Second + rank[a]);
        rank[a] = rank[b] + 1;
    }
}
```

```

        parent[b] = Make_pair (a,
1);if (rank[a] == Rank[b])
    + + Rank[a];
}

```

Support parity path length and the problem of verifying the bipartite graph online

By analogy with the path length to the leader, so it is possible to maintain the path length of the parity before. Why is this application has been allocated as a separate item?

The fact that usually demand parity storage path emerges in connection with the following **task**: Initially given a blank graph, it can be added to the edges, and do requests like "whether connected component containing a given vertex, **bipartite**?".

To solve this problem, we can introduce the system to store disjoint sets of connected components,
и store each vertex parity path length to its leader. Thus, we can quickly check whether the addition of the ribs to a violation of a bipartite graph or not: namely, if the ends of the ribs lie in the same
и the same connected component, and thus have the same parity path length to the leader, the addition of this edge will result in a cycle of odd length and the transformation of the current components in the non-bipartite.

Home **complexity**With which we are confronted with this - is that we must be careful, given the parities to produce the union of two trees in the function **union_sets**.

If we add an edge (a, b) Connecting the two components into one, when connecting a single tree to another, we need to tell the parity such that the result of vertices a and b have obtained different parity path length.

Deduce **formula**On who should receive this parity, invoiced to the leader of one set when attaching it to the leader of another set. We denote x parity path length from the top to the leader it sets through y - The parity of the path length from the vertex to its leader set, and by t - Sought parity, we must put attachable leader. If set to a vertex a joins to the set with the top, becoming a subtree, then after joining at the top of its parity will not change and remains equal y . And at the top is equal parity $x \oplus t$ (Symbol \oplus here denotes the operation XOR (symmetric difference)). We want these two parity differed, ie their XOR is unity.

le obtain an equation for t :

solving which we find:

$$t = x \oplus y \oplus 1.$$

Thus, no matter how many joins what, you should use the specified formula to set the parity edges conducted from one leader to another.

We present **implementation** DSU supporting parities. As in the previous paragraph, for purposes of convenience, we use a pair of storage ancestors and results of operations. In addition, for each set we store in the array **bipartite** Whether it is still bipartite or not.

```

void make_set (int v) {
    parent[v] = Make_pair (v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}

pair <int,int> Find_set (int v) {
    if (v != parent[v]. First) {
        int parity = parent[v]. Second;
        parent[v] = Find_set (parent[v].
First); parent[v]. Second ^= parity;
    }
    return parent[v];
}

void add_edge (int a, int b) { pair
<int,int> Pa = find_set (a); a =
pa.first;
int x = pa.second;

pair <int,int> Pb = find_set (b);

```

```

b = pb.first;
int y = pb.second;

if (a == b) {
    if (x == y)
        bipartite[a] = false;
}
else {
    if (rank[a] < Rank[b])
        swap (a, b);
    parent[b] = Make_pair (a, x ^ y ^
1); if (rank[a] == Rank[b])
        + + Rank[a];
}
}

bool is_bipartite (int v) {
    return bipartite[ find_set(v) . First ];
}

```

Algorithm for finding the RMQ (minimum interval) for $O(\alpha(n))$ average offline

Formally problem is formulated as follows: necessary to implement the data structure, which support two types of queries: adding the specified number ($i \in 1 \dots n$) And search and retrieval

current minimum number $\text{extract_min}()$. We assume that each number is added only once. Furthermore, we

assume that the entire sequence of requests is known to us in advance, ie task - offline. **The idea of solving** Next.

Instead turns to respond to each query, iterate over

number $i = 1 \dots n$ And determine what response to a request, this number should be. To do this, we need to find this number - it is easy to understand that this first unanswered request going after adding $\text{insert}(i)$ is that request, the response to which is a number.

Thus, there is obtained an idea similar to **problem of painting segments**.

Can obtain a solution for $O(\log n)$ an average of inquiry, if we abandon the rank and heuristics will easy to store in each element closest link to the right query $\text{extract_min}()$ And use compression way to maintain these links after associations.

It is also possible to obtain a solution

and $O(\alpha(n))$ If we use the rank heuristic and will store in each set position number where it ends (that in the previous version of the solution is achieved automatically by the fact that the links were always just right - now we have to be stored explicitly).

Algorithm for finding the LCA (lowest common ancestor in the tree) for $O(\alpha(n))$ average offline

Tarjan's algorithm for finding the LCA average online described in [corresponding article](#).

This algorithm compares favorably with other search algorithms LCA in its simplicity (especially when compared to optimal algorithm Farah-Colton-Bender).

Storing DSU as an explicit list of sets. Application of this idea at the confluence of different data structures

One of the alternative ways of storing DSU is to maintain each set as **explicitly stored list of its elements**. Thus, each element is also saved link the representative of the (leader) of his set.

At first glance it seems that it is an inefficient data structure: the union of two sets, we will have to add one list to the other end, as well as update the leader of all the elements of one of the two lists.

However, as it turns out, the use of **weight heuristics** Similar to that described above, can significantly reduce the asymptotic behavior of up to $O(m + n \log n)$ to perform m queries over n elements.

Under the weight heuristic means that we are always **will add the lesser of two sets in more**. Addition $\text{union_sets}()$ one set to another is easy to implement during the order of the size of the set to add and search leader $\text{find_set}()$ - During $O(1)$ when such storage method.

We prove **asymptotics** $O(m + n \log n)$ to perform m queries. Fix an arbitrary element

and examine how it affected the operations of union worked first time, we can say that the size of its new set will be at least 2. When x exposed a second time - it can be argued that it reaches the set size no less 4 (As we add more to the smaller set). And so on - we see that an element x could affect the maximum merge operations. Thus, the sum over all vertices of $\log n$. $O(n \log n)$, Plus $O(1)$ for each query - QED. Here is an example

implementation:

```

vector<int>
Lst[MAXN]; int
parent[MAXN];

void make_set (int v) {
    Lst[v] = Vector<int>(1, v);
    parent[v] = v;
}

int find_set (int v) {
    return parent[v];
}

void union_sets (int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (Lst[a].Size() < Lst[b].Size())
            swap(a, b);
        while (!Lst[b].Empty()) {
            int v = Lst[b].Back();
            Lst[b].Pop_back(); parent[v]
            = a; Lst[a].Push_back
            (v);
        }
    }
}

```

Also, the idea of adding elements to a smaller set can be used more and outside the DSU, to solve other problems.

For example, consider the following **task**: Given a tree, each leaf is attributed to any number (the same number may occur several times in different leaves). Required for each vertex see the number of different numbers in its subtree.

Applying in this problem the same idea, you can get a solution: Let **traversal depth** Wood, who will return a pointer to **set** properties - a list of all the numbers in the subtree of this vertex. Then, to get an answer to the current node (unless, of course, she did not list) - bypassing the need to call in the depth of all children of this vertex, and combine all received **set** One, the size of which will be the answer for the current node. To effectively combine multiple **set** One just apply the techniques described above: will combine two sets, simply by adding one element in a larger set of smaller. As a result, we obtain a solution

for $O(n \log^2 n)$, Because the addition of one element **set** made for $O(\log n)$.

Storing DSU preserving explicit structure trees. Perepodveshivanie. Search algorithm bridges in the graph for $O(\alpha(n))$ average online

One of the most powerful applications of the data structure "of disjoint sets" is that it allows you to store both **How to compress, and uncompressed structure trees**.

Compressed structure can be used to quickly bring together trees and checking two vertices belonging to one tree, and uncompressed - for example, to find a path between two given vertices, detours or other tree structure.

When implemented, this means that in addition to the usual array of compressed DSU ancestors array of conventional, uncompressed, **real_parent**. It is understood that the maintenance of such an array ancestors worsens asymptotics changes to it occur only when the two trees, and only one element.

On the other hand, in practical applications often need to learn how to connect two wood specified edge is not necessarily coming out of their roots. This means that we have no other choice but to **perepodvesit** one of the trees

for the specified vertex then we were able to attach a tree to another, making the root of the tree of child nodes to be added to the second end edge.

At first glance it seems that the operation perepodveshivaniya - very costly and greatly worsen

asymptotics. Indeed, in the tree top perepodveshivaniya v we must go from this vertex to root of the tree, updating all pointers `parent[]` and `real_parent[]`.

However, in reality things are not so bad: you simply perepodveshivat from the two trees, which is less than one to get the asymptotic behavior of the association, equal $O(\log n)$ avg.

More details (including proof of the asymptotics) see ability to bridge in the graph for $O(\log n)$ average online.

Historical Retrospective

The data structure "system of disjoint sets" was known relatively long time.

A method of storage of this structure in the form of **forest trees** was apparently first described by Haller and Fisher in 1964 (Galler, Fisher "An Improved Equivalence Algorithm"), but a full analysis of the asymptotic behavior was carried out much later.

Heuristics compression paths and associations ranked, seem developed McIlroy (McIlroy) and Morris (Morris), and independently from them, Tritter (Tritter).

Some time was known only estimates $O(\log^* n)$ One operation in the middle, this Hopcroft and Ullman in 1973 (Hopcroft, Ullman "Set-merging algomthms") - here **$\log^* n$ -iterated logarithm** (It is slowly increasing function, but not so slow as the inverse Ackermann function).

First assessment $O(\alpha(n))$ Wherein $\alpha(n)$ -**inverse Ackermann function** - Tarjan received in his Article 1975 (Tarjan "Efficiency of a Good But Not Linear Set Union Algorithm"). Later in 1985, he got together with Lyuvenom this time estimate for several different heuristics rank and path compression techniques (Tarjan, Leeuwen "Worst-Case Analysis of Set Union Algorithms").

Finally, Fredman and Saks in 1989 proved that the adopted model calculations **any** algorithm system of disjoint sets should work at least on average (Fredman Saks "The cell probe complexity of dynamic data structures").

However, it should also be noted that there are several articles **challenging** this time estimate and argues that a system of disjoint sets with heuristics and path compression union by rank work for an average of: Zhang "The Union-Find Problem Is Linear", Wu, Otoo "A Simpler Proof of the Average Case Complexity of Union-Find with Path Compression".

Problem in online judges

List of tasks that can be solved using a system of disjoint sets:

- TIMUS # 1671 "Spider Anansi" [Difficulty: Easy]
- CODEFORCES 25D "Roads not only in Berland" [Difficulty: Medium]
- TIMUS # 1003 "Parity" [Difficulty: Medium]
- SPOJ # 1442 "Chain" [Difficulty: Medium]

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Construction and analysis** [2005]
- Kurt Mehlhorn, Peter Sanders. **Algorithms and Data Structures: The Basic Toolbox** [2008]
- Robert Endre Tarjan. **Efficiency of a Good But Not Linear Set Union Algorithm** [1975]
- Robert Endre Tarjan, Jan van Leeuwen. **Worst-Case Analysis of Set Union Algorithms** [1985]

Segment tree

Segment tree - a data structure that allows efficient (ie the asymptotic behavior $O(\log n)$) implement the following operations: finding the amount / minimum of array elements in a given interval $a[l \dots r]$. Where input to the algorithm), the method further may change array elements: how changing the value of one element, and change elements on a whole array of subsegments (ie, all the elements are allowed to assign a value, or to add to all the elements array any number).

Generally, the segment tree - a very flexible structure and the number of problems solved her theoretically unlimited. In addition to the above types of transactions with trees segments are also possible and much more complex operations (see "Complicated version tree segments"). In particular, the segment tree is easily generalized to higher dimensions: for example, to solve the problem of finding the amount / minimum in some

podpryamougolnike this matrix (but only for the time already $O(\log^2 n)$).

An important feature of tree segments is that they consume a linear memory: standard segment tree requires about $4n$ memory elements to work on an array of size n .

Description of the tree segments in the base case

First, consider the simplest case of tree segments - segment tree for sums. If you pose the problem formally, then we have an array, and our segment tree should be able to find the sum $a[0 \dots n-1]$ c-th element $a[i]$ (this request amounts) and also handle value change of a specified element of the array actually respond to the assignment (this modification request). Again $a[i] = x$ again, the segment tree must handle both the request for time $O(\log n)$.

Tree Structure Segments

So, what is a segment tree?

We calculate and remember somewhere sum of all elements of the array, ie segment $[0 \dots n-1]$. Also calculate the amount of two halves of the array: $a[0 \dots n/2]$ and $a[n/2 + 1 \dots n-1]$. Each of these two halves, in turn, divide in half, calculate and store the sum at them, then divide in half again, and so on until it reaches the current segment length 1. In other words, we start with an interval and $[0; n-1]$ every time we divide the current segment in half (if it has not yet become a segment of unit length), then calling the same procedure on both halves; for each such segment we store the sum on it.

We can say that these segments in which we considered the sum, form a tree: the root of the tree - the segment, and each vertex has exactly two sons (except leaf vertex whose interval has a length 1). Hence the name - "segment tree" (although the implementation is usually no tree is clearly not built, but on this below in the implementation section).

Now that we have a tree structure segments. Immediately, we note that it has **linear dimension** And it comprises at least $2n$ vertices. This can be understood as follows: the first level of the tree segments contains one vertex (segment $[0 \dots n-1]$). The second layer - in the worst case, two peaks, the third level b the worst case there will be four peaks, and so forth until it reaches the number of vertices n . Thus, the number of vertices b worst case estimated by the sum $n + n/2 + n/4 + n/8 + \dots + 1 < 2n$.

It is worth noting that when n than a power of two, not all levels of the tree segments will be fully filled. For example, when $n = 3$ left son of the root is a segment $[0 \dots 1]$ Having two children, while as the right child of the root - cut $[2 \dots 2]$, Is a leaf. No special difficulties in the implementation of this is not, but nevertheless it must be borne in mind.

Height tree segments is the value - eg because the length of the segment in the root of the tree is n And when you go to one level down the length of the segments is reduced by about half.

Construction

The process of constructing the tree segments for a given array a can be done efficiently as follows, from the bottom up: first, record the values of the elements in the corresponding leaves of the tree, then on the basis of $a[i]$

They calculate values for the vertices of the previous level as the sum of the two leaves, and then a similar manner to calculate the value of another level, etc. Convenient to describe this operation recursively: we run the procedure of construction of the root segments, and the procedure of construction, if it is not caused by the sheet, calls itself from each of two sons and summarizes the calculated values, and if it is caused by the sheet - simply writes a value of the array element.

Asymptotics tree construction segments will thus $O(n)$.

Request amount

We now consider the amount of the request. Are input two numbers l and r And we have time for sum of the numbers in the interval $a[l \dots r]$

$O(\log n)$ consider

To do this, we will go down the tree segments constructed using to calculate the amount previously computed response at each node of the tree. Initially we get to the root of the tree segments. Let's see in which of the two

his sons misses cut request (Recall that the sons of the root segments - it

$[l \dots r]$). There are two possibilities: that the

segments $[0 \dots n/2]$ and $[n/2 + 1 \dots n - 1]$ segment $[l \dots r]$ misses only one son, root, and that, conversely, the segment intersects with two sons.

The first case is simple: just go to that son, in which lies our segment query and apply the algorithm described here to the current node.

In the second case, we no other options but to go first to the left and count son prompted him, and then - go to the right child, calculate the answer in it and add to our response. In other words, if the left son represented the segment, and the right - section (note that $l_2 = r_1 + 1$), Then we go to the left child request $[l \dots r_1]$ And in the right - with the request $[l_2 \dots r]$

$[l_1 \dots r_1]$ $[l_2 \dots r_2]$

that $l_2 = r_1 + 1$, Then we go to the left child request $[l \dots r_1]$ And in the right - with the request $[l_2 \dots r]$

Thus, the sum of query processing is **recursive function** Which calls itself whenever either the left child or from the right (without changing the query boundaries in both cases), or by both at once (at the same time sharing our inquiry into two corresponding subquery). However, recursive calls are not always going to do: if the current request coincided with the boundaries of the segment in the current top of the tree segments, the response will return the precomputed value of the sum in this segment, recorded in the tree segments.

In other words, the query evaluation is descending a tree segments, which extends all the necessary branches of the tree, and for quick work using already computed the amount in each segment in the tree segments.

Why **asymptotics** This algorithm will $O(\log n)$? To do this, look at each level

tree segments as maximum lengths could visit our recursive function when processing a request. It is argued that these segments could not be more than four; Then, using the estimate

$O(\log n)$

for the height of the tree, we obtain the desired asymptotic behavior of the algorithm time.

We show that the evaluation of the four segments is correct. In fact, at the zero level of the tree is affected only request top - the root of the tree. Next on the first level recursive call in the worst case is split into two recursive calls, but it is important here is that the questions in these two calls will coexist, ie request by the second recursive call is one more than the number of requests in the first recursive call. This implies that at the next level, each of these two calls could produce two more recursive calls, but in this case, half of the non-recursive queries will work, taking the desired value from the top of the tree segments. Thus, every time we will have no more than two real

operating branches recursion (we can say that one branch close to the left border of the query, and the second branch - to the right), and total number of affected segments could not exceed the height of the tree segments, multiplied by four, ie it is the number of $O(\log n)$.

Finally, you can lead an understanding of the request amount: input segment $[l \dots r]$ breaks several subsegments, the answer to each of which is calculated and stored in the tree. If you do it the right way partition, thanks to the tree structure of segments required by subsegments will always $O(\log n)$, Which gives the performance of the tree segments.

Update request

Recall that the update request receives the input index i and the value of x And rearranges the tree segments so as to conform to the new value. This request must also be performed for $u[i] = x$ time $O(\log n)$.

It is more simple compared to the query request amount counting. The fact that the element $a[i]$ involves only a relatively small number of vertices of the segments: namely, in $O(\log n)$ tops - one on each level.

Then it is clear that the update request can be implemented as a recursive function: it is passed the current node of the tree lines, and this function performs a recursive call from one of his two sons (from that which contains the position i in its segment), and after that - recalculates the value of the amount of the current vertex in the same way as we did in the construction of the tree segments (ie, the sum of the values for both sons of current node).

Implementation

Realizable main point - is that, as **store** wood segments in memory. For simplicity, we will not store a tree in an explicit form, and use this trick: we say that the root of the tree has a number 1,

his sons - the rooms 2 and 3 and their sons - rooms 4 on and so forth. Easy to understand the correctness of the following formula: if the node has room i , Then let her son left - is the pinnacle of a number $2i$ And the right - with the number $2i + 1$.

This technique greatly simplifies the programming tree segments - now we do not need to be stored in the memory tree structure of segments, but only to have a array of amounts on each segment of the tree segments.

One has only to note that the size of the array at a numbering should not put $2n$ And $4n$. The fact that this numbering does not work perfectly when n not a power of two - then there are missed calls, which do not correspond to any vertex of the tree (actually, numbering behaves just as if n be rounded up to the nearest power of two). This does not pose any difficulties in the implementation, however leads to the fact that the size of the array must be increased to $4n$.

So, we segment tree **store** just as an array $t[]$, Four times larger than the size n Input:

```
int n, t[4* MAXN];
```

Procedure **tree construction segments** for a given array is as follows: a recursive function, passing it the array, the number v the current top of the tree, and the boundaries tl and tr segment corresponding to the current top of the tree. From the main program should call this function parameters $v = 1, tl = 0, tr = n - 1$.

```
void build (int a[], int v, int tl, int tr) {
    if (tl == tr)
        t[v] = A[tl];
    else {
        int tm = (tl + tr) / 2;
        build (a, v *2, Tl, tm);
        build (a, v *2+1, tm+1, Tr);
        t[v] = T[v *2] + T[v *2+1];
    }
}
```

Further, the function for **request amount** also is a recursive function in the same way that information is transmitted to the current top of the tree (ie, the number v , Which in the main program should pass 1 , Respectively), and in addition to this - and also borders r current request. In order to simplify this code fukntsii always makes two recursive calls, even if really need one

- Just extra recursive call request be passed, which it is easy to ~~cut off~~ additional check in the beginning of the function.

```
int sum (int v, int tl, int tr, int l, int r) {
    if (l > r)
        return 0;
    if (l == tl && r == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    return sum (v *2, Tl, tm, L, min(r,tm))
        + Sum (v *2+1, tm+1, Tr, max(l,tm+1), R);
}
```

Finally, **modification request**. He just passed the information about the current top of the tree segments, and further, the index changing element, as well as its new value.

```
void update (int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr)
        t[v] = New_val;
    else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update (v *2, Tl, tm, Pos, new_val);
        else
            update (v *2+1, tm+1, Tr, pos, new_val);
        t[v] = T[v *2] + T[v *2+1];
    }
}
```

It is worth mentioning that the function of **update** easy to make a non-recursive because it tail recursion, so is branching never happens: one call can produce only one recursive call. When non-recursive implementation, speed can increase by several times.

From other **optimizations** worth mentioning is that multiplication and division by two is necessary to replace bit operations

- It also slightly improves the performance of the tree segments.

Sophisticated version of the tree segments

Segment tree - a very flexible structure, and allows you to make generalizations in many different directions. Try to classify them below.

More complex functions and queries

Improve tree segments in this direction can be as pretty obvious (as in the case of the minimum / maximum instead of the sum), and a spectacular nontrivial.

Minimum / maximum

Little to change the conditions of the problem described above: instead of requesting a sum will produce now request the minimum / maximum on the interval.

Then the tree segments for this practically does not differ from the tree segments described above. Just need to change the method of calculating the functions and as well as calculated returned response function **SUM** (Replace the summation by minimum / maximum).

[**pair**] **build** **update**

Combining two such pairs in the same worth as a separate function because this operation will need to produce and modify the query, and the query search for the maximum.

```
pair <int,int> T[4* MAXN];

pair <int,int> Combine (pair <int,int> A, pair <int,int>
    B) { if (a.first> b.first)
        return a;
    if (b.first> a.first)
        return b;
    return make_pair (a.first, a.second + b.second);
}

void build (int a[], int v, int tl, int tr) {
    if (tl == tr)
        t[v] = Make_pair (a[tl],1);
    else {
        int tm = (tl + tr) / 2;
        build (a, v *2, Tl, tm);
        build (a, v *2+1,tm+1, Tr);
        t[v] = Combine (t[v *2], T[v *2+1]);
    }
}

pair <int,int> Get_max (int v, int tl, int tr, int l, int r)
{ if (l> r)
    return make_pair (-INF, 0);
if (l == tl && r == tr)
    return t[v];
int tm = (tl + tr) / 2;
return combine (
    get_max (v *2, Tl, tm, L, min(r,tm)),
    Get_max (v *2+1,tm+1, Tr, max(l,tm+1), R)
);
}

void update (int v, int tl, int tr, int pos, int new_val) {
```

```

    if (tl == tr)
        t[v] = Make_pair (new_val, 1);
    else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update (v *2, Tl, tm, Pos, new_val);
        else
            update (v *2+1, tm+1, Tr, pos, new_val);
        t[v] = Combine (t[v *2], T[v *2+1]);
    }
}

```

Find the greatest common divisor / least common multiple

If we want to learn to look for GCD / LCM of all the numbers in a given segment of the array.

It's pretty interesting generalization tree lengths obtained in exactly the same way as for the amount of trees segments / minimum / maximum: simply stored in each node of the tree GCD / LCM of all the numbers in the corresponding segment of the array.

Count the number of zeros, search k Zero th

In this task, we want to learn how to respond to your request the number of zeros in a given segment of the array, as well as finding inquiry k Th element zero.

Again slightly modify the data stored in the tree segments: we keep the amount now in the array $t[]$ zeros occur in respective segments of the array. It is clear, how to maintain and use the data functions build, sum, update. Thus we solved the problem of the number of zeros in a given segment of the array.

Now learn how to solve the problem of finding the position of the first occurrence of zero in the array. To do this, we will go down the tree segments, starting from the root, and going every once in left or right child depending on which of the segments is required th zero. In fact, to understand what we need to move on son, just look at the value stored in the left son if it is greater than or equal to, the need to move to the left son (because it has a minimum interval k zeros), and otherwise - to move in the right child.

In the case of implementation can be cut off when the zero-th exists even when the function returned as a response, for example, -1 .

```

int find_kth (int v, int tl, int tr, int k) {
    if (k > t[v])
        return -1;
    if (tl == tr)
        return tl;
    int tm = (tl + tr) / 2;
    if (t[v *2] >= K)
        return find_kth (v *2, Tl, tm, K);
    else
        return find_kth (v *2+1, tm+1, Tr, k - t[v *2]);
}

```

Search prefix array with a given sum

Problem is this: you want for a given value x quickly find a i That the sum of the first i array elements is greater than $a[]$ or equal to x (Assuming that the array $a[]$ contains only non-negative integers).

This problem can be solved using binary search, calculating each time inside the sum on a particular prefix array, but it will lead to the solution during $O(\log^2 n)$.

Instead, you can use the same idea as in the previous paragraph, and look for the desired position of a descent on a tree, turning every once in a left or right child depending on the value of the sum in the left son. Then the answer to the search request will be one such descending tree, and, hence, will be performed for $O(\log n)$.

Search subsegments with a maximum amount

Still given to the input array $a[0 \dots n - 1]$ And receives requests (l, r) Which means: find a subsegment $a[l' \dots r']$ That $l \leq l' r' \leq r$ and the amount of this segment $a[l' \dots r']$ maximum. Requests modification of individual elements of the array are allowed. Array elements can be negative (and, for example, if all the numbers are negative, the optimal subsegments will be empty - it sum is zero).

This is a very nontrivial generalization tree segments obtained as follows. Will be stored in each node of the tree segments

four values: the amount in this segment, the maximum amount of all

prefixes of this segment, the maximum amount of all suffixes, as well as the maximum amount for subsegments it. In other words, for each segment of the tree segments the answer to it already predposchitan and additionally answer counted among all the segments that are limited in the left margin of the segment, as well as among all the segments that are limited in the right border.

How do you build tree to such data? Again we come to this point of view with recursive let for the current top four values in the left son and son in law already counted, count them now for the summit. Note that the answer is in the very top:

- or response in the left son, which means that the best in the current top subsegment can actually fit into a segment of the left son
- or answer in the right son, that means that the best in the current top subsegment can actually fit into a segment of the right son
- or maximum amount of the suffix in the left son and the maximum prefix in the right son, that means that the best subsegment is its origin in the left son, and the end - in the right.

Hence, in response to the current vertex is the maximum of these three values. Recalculate the maximum amount on the same prefix and suffix even easier. We present the implementation of functions, which will be passed two structures, inclusive of the left and right of the sons, and which returns the data in the current vertex.

```
struct data {
    int sum, pref, suff, ans;
};

data combine (data l, data r) {
    data res;
    res.sum = l.sum + r.sum;
    res.pref = max (l.pref, l.sum + r.pref);
    res.suff = max (r.suff, r.sum + l.suff);
    res.ans = max (max (l.ans, r.ans), L.suff +
    r.pref);return res;
}
```

So we learned how to build a tree segments. Is easy to obtain and implement modification request: as in the simple tree segments, we recalculate the values in the tree tops all changed

segments, which all use the same function . To calculate the values of tree leaves and the auxiliary function That returns the structure Calculated one number make_data data r val .

```
data make_data (int val) {
    data res;
    res.sum = val;
    res.pref = res.suff = res.ans = max (0,
    Val);return res;
}

void build (int a[], int v, int tl, int tr) {
    if (tl == tr)
        t[v] = Make_data (a[tl]);
    else {
        int tm = (tl + tr) / 2;
        build (a, v *2, Tl, tm);
        build (a, v *2+1, tm+1, Tr);
        t[v] = Combine (t[v *2], T[v *2+1]);
    }
}

void update (int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr)
        t[v] = Make_data (new_val);
    else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update (v *2, Tl, tm, Pos, new_val);
        else
            update (v *2+1, tm+1, Tr, pos, new_val);
        t[v] = Combine (t[v *2], T[v *2+1]);
    }
}
```

It remains to deal with the response to the request. To do this, we have also, as before, down the tree, breaking those most segment requests $a[l \dots r]$ several subsegments, coincides with the segment tree segments and They combine the responses into a single response for the whole task. Then it is clear that the work is no different from ordinary wood work segments, instead of just need a simple summation / minimum / maximum values use the **combine**. The following implementation is slightly different from the sale of Request **sum**: It does not allow for cases when the left boundary request exceeds the right boundary (otherwise any unpleasant incidents - what structure **data** I return when the query interval is empty? ..).

```
data query (int v, int tl, int tr, int l, int r) {
    if (l == tl && tr == r)
        return t[v];
    int tm = (tl + tr) / 2;
    if (r <= tm)
        return query (v * 2, Tl, tm, L,
                      r); if (l > tm)
        return query (v * 2+1, tm+1, Tr, l,
                      r); return combine (
            query (v * 2, Tl, tm, L, tm),
            query (v * 2+1, tm+1, Tr, tm+1, R)
        );
}
```

Preserve all the subarray in each node of the tree segments

This is a separate subsection standing apart from the rest because each node of the tree segments we will not store any information about this compressed subsegments (sum, minimum, maximum, etc.), and all array elements that lie in the subsegments. Thus, the root of the tree will store all segments elements of the array, the left son of the root - the first half of the array, right child of the root - the second half, and so on.

The simplest version of the application of this technique - where each node of the tree segments stored sorted list of all numbers appearing in the corresponding interval. In more complex scenarios are not stored lists, and any data structure built on these lists (**set**, **map** etc.). But all these methods have in common is that each node of the tree segments stored some data structure in memory having a size of about the length of the corresponding segment.

The first natural question is posed when considering trees segments of this class - is **memory usage**. Allegedly that if each node of the tree segments stored a list of all occurring at this interval number or any other data structure size of the same order, the sum of all segment tree will occupy memory cells. Why is this so? Because each number $O(n \log n)$ flagged $O(\log n)$ tree segments segments (not least because the height of the tree segments have $O(\log n)$).

Thus, despite the apparent extravagance of the tree lines, it consumes memory is not much longer than normal wood segments.

Below described are some typical uses such a data structure. It is worth noting immediately clear analogy trees segments of this type **two-dimensional data structures** (Actually, in a sense, this is a two-dimensional data structure, but rather with disabilities).

Find the smallest number greater than or equal to the specified in this interval.

No modification requests

Required to respond to requests from the following: $a[l \dots r]$ That is to find the minimum number in the interval $a[l \dots r]$ which is greater than or equal to x .

Construct segment tree, in which each node will store the sorted list of all numbers appearing on the corresponding interval. For example, the root will contain an array $a[]$ sorted order. How to build a segment tree as efficiently? For this approach the task, as usual, in terms of recursion: let the left and right children of the current node, these lists have already been constructed, and we need to build this list for the current node. In this formulation, the question becomes almost obvious that this can be done in linear time: we just need to merge two sorted lists into one that is done in one pass on them with two pointers. C ++ users even easier, because this merging algorithm is already included in the standard library STL:

```
vector <int> T[4 * MAXN];

void build (int a[], int v, int tl, int tr) {
    if (tl == tr)
        t[v] = Vector <int>(1, A[tl]);
    else {
        int tm = (tl + tr) / 2;
```

```

        build (a, v *2, Tl, tm);
        build (a, v *2+1, tm+1, Tr);
        merge (t[v *2]. Begin(), T[v *2]. End(), T[v *2+1]. Begin(),
T[v *2
+1]. End());
    }
}
}

```

We already know that so constructed segment tree will occupy memory. And thanks such an implementation time of its construction also has value $O(n \log n)$. Because each list is constructed for linear in the size of its time. (Incidentally, there is an obvious analogy here with the algorithm **mergesort**: Only here we store the information from all stages of the algorithm, not only the total.)

Now consider the **response to a request**. Will descend on the tree, as it makes a standard response to a request in the tree segments, breaking our segment into several subsegments (some units). Clearly, that the answer to the whole problem is the minimum of the responses to each of these subsegments. Understand now how to respond to a request for one such subsegments, coinciding with a vertex of the tree.

So, we came to the top of some tree segments and want to compute the answer to it, ie find the smallest number greater than or equal to this x . For this we just need to perform **binary search** on the list, considered in this top of the tree, and return the first number on the list, more than or equal to x .

Thus, in response to a request occurs subsegments $O(\log n)$ And the entire request is being processed for time $O(\log^2 n)$.

```

int query (int v, int tl, int tr, int l, int r, int x) {
    if (l > r)
        return INF;
    if (l == tl && tr == r) {
        vector <int> :: Iterator pos = lower_bound (t[v]. Begin(),
T[v].
end(), x);
        if (pos != t[v].
End())
            return
            * pos;
        return INF;
    }
    int tm = (tl + tr) / 2;
    return min (
        query (v *2, Tl, tm, l, min(r,tm), x),
        query (v *2+1, tm+1, Tr, max(l,tm+1), r, x)
    );
}

```

Constant **INF** is equal to some large number, certainly more than any number in the array. It carries the meaning of "response in a given interval does not exist."

Find the smallest number greater than or equal to the specified in this interval.

Allowed modification requests

Problem is similar to the previous one, only now allowed modification requests: assignment process $a[i] = y$.

The solution is also similar to the preceding problem, but instead of a simple list in each node of the tree segments we store a balanced list that allows you to quickly search for the desired number, remove it and insert a new number. Given that the general number of the input array can be repeated, the best choice is the structure of STL data **multiset**.

Construction segments of the tree occurs in about the same as in the previous problem, just now

should unite not sorted lists, That will lead to the fact that construction of the multiset asymptotics deteriorate to $n \log^2 n$ (Although, apparently, red-black trees allow to merge two trees in linear time, but the STL does not guarantee it).

The answer to **search request** general practically equivalent to the code above, only now **lower_bound** need to call on $t[v]$.

Finally, **modification request**. To handle it, we have to go down the tree, making changes to all lists containing affected items. We simply remove the old value of this element ($O(\log n)$) and insert the new value. (Do not forget that we do not need to remove them all together with the number of repetitions) and insert the new value.

```
void update (int v, int tl, int tr, int pos, int new_val) {  
    t[v].Erase (t[v].Find (a[pos]));  
    t[v].Insert (new_val);
```

```

        if (tl! = tr) {
            int tm = (tl + tr) / 2;
            if (pos <= tm)
                update (v *2, Tl, tm, Pos, new_val);
            else
                update (v *2+1,tm+1, Tr, pos, new_val);
        }
    else
        a[pos] = New_val;
}

```

Process this request also occurs during $O(\log^2 n)$.

Find the smallest number greater than or equal to the specified in this interval.

Acceleration using the technique of "partial cascading"

Improve response time to find time $O(\log n)$ by applying art
"Partial cascading" ("Fractional cascading").

Partial cascading - is a simple technique that helps to improve the work of several binary searches being conducted by the same value. In fact, the response to the search request is that we divide our task into several subtasks, each of which is then solved for the number of binary search x . Partial cascading allows to replace all of these binary searches on one.

The simplest and most obvious example is the partial cascading **the next challenge**: There are several sorted lists of numbers, and we have in each list to find the first number is greater than or equal to the specified value.

If we solved the problem of "head", it would have to run a binary search on each of these lists,

, Then the What if a lot of these lists, it becomes a very important factor: if all lists will $O(k \log(n/k))$ n Where - the total size of all lists (such asymptotics, because the worst in case - where all of the lists are approximately equal in length to each other, i.e. equal n/k).

Instead, we could combine all these lists into one sorted list, where each number will keep a list of positions: first position in n_i the list of the first number is greater than or equal, n_i a similar position in the second list, and so on. In other words, for each number we keep occurring at the same time the number of binary search results on it in each of the lists. In this case,

asymptotic behavior of the response to the request is received $O(\log n + k)$, Which is significantly better, but we forced to pay a large memory consumption: namely, we need $O(nk)$ memory cells.

Tech partial cascading going on in this task and achieves memory consumption $O(n)$ at the same time the answer to the inquiry $O(\log n + k)$. (To do this, we keep not one big list length n , and back again to the lists, but with each list is stored every second element from the following list; we again have the number with each record its position in both lists (current and next), but it will continue to effectively respond to a request: we do a binary search on the first list, and then go on these lists in order, moving each time The following list of predposchitannyy using pointers and taking one step to the left, thereby considering that half of the next list number has not been taken into account).

But we in our application to the tree segments **not needed** full power of this technology. The fact that the list contains the current node to all the numbers which can occur in the left and right sons. Therefore, to avoid a binary search through the list of his son, it is sufficient for each list in the tree segments for each count of the number of its position in the list of the left and right sons (more precisely, the position of the first number is less than or equal to the current).

Thus, instead of a simple list of all the numbers we store a list of triples: the number itself, the position in the list left child, the right position in the list of his son. This will allow us to determine the position in the list left or right son, instead of doing a binary list on it.

The easiest way to apply this technique to the problem when no modification requests - then these positions are just numbers, and count them in the construction of the tree very easily inside the algorithm merge two sorted sequences.

In the case of modification requests are allowed, all a bit more complicated: these positions now be stored in the form of iterators inside, and when you request an update - the right to reduce / increase for those elements for which it is required.

Anyway, the problem is reduced to net realizable subtleties, but the basic idea - replacing a binary search $O(\log n)$ search on the list in the root of the tree - fully described.

Other possible

Note that this technique implies a whole class of possible applications - everything is determined by the structure of the data selected for storage in each node of the tree. We have examined the application

using `vector` and `multiset`. While in general you can use any other compact data structure: other tree segments (about this little discussed below in the section on multivariate trees segments) [Fenwick tree](#), [Cartesian tree](#) etc.

Update on the interval

We have considered only the problem when a modification request affects only element of the array. In fact, the segment tree allows queries that apply to entire segments of contiguous elements, and on these requests in the same time $O(\log n)$.

Addition to the interval

We begin our consideration of trees such segments with the simplest case: a modification request is the addition of all the numbers in a certain number of subsegments x . Read request - $a[l \dots r]$ still reading a certain number of values $a[i]$.

To make a request for adding effectively will be stored in each node of the tree segments as necessary to add all the numbers of this segment as a whole. For example, if a request comes in "add to all array by $a[0 \dots n-1]$ " in the root of the tree by 2. Thus, we can process the request adding any subsegments efficiently, rather than changing all $O(n)$ values.

Now, if a request comes in to read the value of a number, it is sufficient to go down the tree, summing all encountered on the way the values written in the tree tops.

```
void build (int a[], int v, int tl, int tr) {
    if (tl == tr)
        t[v] = A[tl];
    else {
        int tm = (tl + tr) / 2;
        build (a, v *2, Tl, tm);
        build (a, v *2+1, tm+1, Tr);
    }
}

void update (int v, int tl, int tr, int l, int r, int add) {
    if (l > r)
        return;
    if (l == tl && tr == r)
        t[v] += Add;
    else {
        int tm = (tl + tr) / 2;
        update (v *2, Tl, tm, L, min(r,tm), Add);
        update (v *2+1, tm+1, Tr, max(l,tm+1), R, add);
    }
}

int get (int v, int tl, int tr, int pos) {
    if (tl == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return t[v] + Get (v *2, Tl, tm, Pos);
    else
        return t[v] + Get (v *2+1, tm+1, Tr, pos);
}
```

Assigning the interval

Suppose now that a modification request is assigned to all the elements of a certain length $a[l \dots r]$ some value p . As a second request will be considered read array values $a[i]$.

To make a modification on the whole segment have in each node of the tree segments store, painted this segment entirely in any number or not (and if painted, the store itself is a number). This will allow us to do "**Retarded**" **update** tree segments: the modification request, we instead change the values in the set of vertices of the tree segments will change only some of them, leaving the flags "painted" for other segments, which means that the entire segment with their subsegments should be painted in this color.

So, after making a request modifications segment tree becomes, generally speaking, irrelevant - it remained Shortfall some modifications.

For example, if the request came modification "to assign the entire array $n - 1$ a number "then tree segments we will only change - will mark a tree root that he painted entirely in that number. The rest of the top of the tree remain unchanged, although in fact the entire tree must be painted in the same number.

Suppose now that in the same tree segments came second modification request - to paint the first half of the array at any other number. To handle such a request, we have

paint the entire left child of the root in this new color, but before we do that, we must deal with the root of the tree. The subtlety here is that the tree should be preserved, that the right half is painted in old color, but at the moment no information in the tree for the right half was not saved.

Output is: produce **pushing** information from the root, ie if the root of the tree was painted in any number, then paint in the number of its right and left his son, and from the root to remove this mark. After that, we can safely paint left child of the root, without losing any necessary information.

Summarizing, we obtain for any queries with tree (modification request or reading) while descending the tree, we should always do push information from the current node in both of her sons. You can understand it so that when descending the tree, we use the retarded modification, but only as much as needed (not to worsen with the asymptotic behavior $O(\log n)$).

When implemented, this means that we need to make a function that will be transferred to the vertex tree lines, and it will produce pushing information from this vertex in both her sons. Should call this function at the beginning of request processing functions (but not call it from the leaves, because of the push plate information is not necessary, and nowhere).

```

void push (int v) {
    if (t[v] , = -1) {
        t[v *2] = T[v *2+1] =
        T[v]; t[v] = -1;
    }
}

void update (int v, int tl, int tr, int l, int r, int color) {
    if (l> r)
        return;
    if (l == tl && tr == r)
        t[v] = Color;
    else {
        push (v);
        int tm = (tl + tr) / 2;
        update (v *2, Tl, tm, min(r,tm), Color);
        update (v *2+1,tm+1, Tr, max(l,tm+1), R, color);
    }
}

int get (int v, int tl, int tr, int pos) {
    if (tl == tr)
        return t[v];
    push (v);
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return get (v *2, Tl, tm, Pos);
    else
        return get (v *2+1, tm+1, Tr, pos);
}

```

Function `get` could be implemented in another way: do it delayed updates, and immediately the response, as soon as it hits the top of the tree segments, entirely painted in a particular color.

The addition of the interval, the maximum request

Suppose now that the modification request will again request the addition of all subsegments including some of the same number and a request is to find the maximum reading in some subsegments.

Then at each node of the tree segments will need to additionally store a maximum of all these subsegments. But subtlety here is how to recalculate these values.

For example, suppose there was a request "added to the entire first half i.e. $n/2$ Number 2 ". Then tree it will affect the recording number 2 in the left child of the root. How now calculate the new value of the maximum in the left son and the root? Here it becomes important not to get confused - what the maximum is stored in the tree top: maximum without adding on top of all this, or considering it. You can choose any of these approaches, but the main

- Consistently use it everywhere. For example, the first approach is fundamentally maximum will be obtained

as the maximum of two numbers, a maximum in the left son, plus the addition of the left son, and the son of a maximum in the right plus an addition to it. In the second approach, the same maximum in the root will be obtained as an addition to the root plus a maximum of maxima in the left and right sons.

Other Destinations

Here were considered only basic application segments trees in problems with the modifications on the segment. Other objects are obtained based on the same ideas as described herein.

It is only important to be very careful when dealing with pending modifications: it should be remembered that even if the current vertex we have "pushed" the pending revision, the left and right sons likely have not yet done. Therefore it is often necessary to call `is also on the push` left and right sons the current node, or as carefully consider pending modifications in them.

Generalization to higher dimensions

Segment tree generalized quite naturally on the two-dimensional and multi-dimensional case at all. If the one-dimensional case we broke array indexes on segments, the two-dimensional case will now first break all the indices in the first, and for each segment for the first index - build common tree segments for the second index. Thus, the basic idea of the solution - it is inserting trees segments on the second index into the tree segments for the first index.

Let us explain this idea for an example of the problem.

Two-dimensional segment tree in the simplest case

Dana rectangular matrix $a[0 \dots n - 1, 0 \dots m - 1]$ And enter search queries sum (or minimum / maximum) at some podryamougolnikah $a[x_1 \dots x_2, y_1 \dots y_2]$ And requests modification of individual elements of the matrix (ie, queries of the form $a[x][y] = p$).

So, we will build a two-dimensional tree segments: first segment tree in the first coordinate (x), Then - on the second (y).

That the process of constructing more understandable, it is possible to forget that the original array was two-dimensional, and leave only the first coordinate. We will construct the usual one-dimensional segment tree, working only with first coordinate. But as the value of each segment, we will not record a number, as in dimensional case, and the whole tree segments: ie at this point we are reminded that we still have the second and coordinate; but since at this point is recorded that the first coordinate is an interval $[l \dots r]$

we actually work with the band $a[l \dots r, 0 \dots m - 1]$ And for her to build a tree segments.

We present the implementation of operations for constructing a two-dimensional tree. It actually consists of two separate units: the construction of the tree segments coordinate `build_x()` And the coordinate `build_y()`. If

The first function is almost indistinguishable from the conventional one-dimensional tree, the latter is forced to deal

separately two cases: when the current segment in the first coordinate ($[tlx \dots trx]$) Has unit length, and when - length greater than one. In the first case, we simply take the required value from the matrix $a[]$ second - Combine the values of two tree lengths of the left and right son son coordinate x .

```

void build_y (int vx, int lx, int rx, int vy, int ly, int ry) {
    if (ly == ry)
        if (lx == rx)
            t[vx][vy] = A[lx][ly];
        else
            t[vx][vy] = T[vx * 2][vy] + T[vx * 2+1][vy];
    else {
        int my = (ly + ry) / 2;
        build_y (vx, lx, rx, vy * 2, Ly, my);
        build_y (vx, lx, rx, vy * 2+1, My +1,
Ry); t[vx][vy] = T[vx][vy * 2] + T[vx]
[vy * 2+1];
    }
}

void build_x (int vx, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        build_x (vx * 2, Lx, mx);
        build_x (vx * 2+1, Mx +1,
Rx);
    }
    build_y (vx, lx, rx, 1, 0, M-1);
}

```

This segment tree takes still linear memory, but more constant: $16nm$

memory cells. Clearly, it is constructed that the process described above `build_x` also in linear time.

We now turn to **query processing**. Respond to the two-dimensional inquiry will on the same principle: first break request to the first coordinate, and then when we got to the top of the tree some segments in the first coordinate - initiate a request from the relevant sections of the tree to the second coordinate.

```
int sum_y (int vx, int vy, int tly, int try_, int ly, int ry) {
    if (ly > ry)
        return 0;
    if (ly == tly && try_ == ry)
        return t[vx][vy];
    int tmy = (tly + try_) / 2;
    return sum_y (vx, vy * 2, Tly, tmy, ly, min(ry, tmy))
        + Sum_y (vx, vy * 2 + 1, Tmy + 1, Try_, max(ly, tmy + 1), Ry);
}

int sum_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
    if (lx > rx)
        return 0;
    if (lx == tlx && trx == rx)
        return sum_y (vx, 1, 0, M-1, Ly, ry);
    int tmx = (tlx + trx) / 2;
    return sum_x (vx * 2, Tlx, tmx, lx, min(rx, tmx), Ly, ry)
        + Sum_x (vx * 2 + 1, Tmx + 1, Trx, max(lx, tmx + 1), Rx, ly, ry);
}
```

This function works in the time since it first down the tree in the first coordinate, and for each vertex passed this tree - makes a request for a normal tree segments to the second coordinate.

Finally, we consider **modification request**. We want to learn how to modify the segment tree in accordance with a change in the value of an element $[x][y] = p$. It is clear that changes will occur only in those segments of the first tree tops that cover coordinate x (And there will be $\log n$), And for tree segments, corresponding to it - changes are only those vertices that cover coordinate y (And there is).

Therefore, the implementation of the modification request will not differ greatly from dimensional case, but now we first down in the first coordinate, and then - on the second.

```
void update_y (int vx, int lx, int rx, int vy, int ly, int ry, int x, int y, int new_val) {
    if (ly == ry) {
        if (lx == rx)
            t[vx][vy] = New_val;
        else
            t[vx][vy] = T[vx * 2][vy] + T[vx * 2 + 1][vy];
    }
    else {
        int my = (ly + ry) / 2;
        if (y <= my)
            update_y (vx, lx, rx, vy * 2, Ly, my, x, y, new_val);
        else
            update_y (vx, lx, rx, vy * 2 + 1, My + 1, Ry, x, y, new_val);
        t[vx][vy] = T[vx][vy * 2] + T[vx][vy * 2 + 1];
    }
}

void update_x (int vx, int lx, int rx, int x, int y, int new_val) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        if (x <= mx)
            update_x (vx * 2, Lx, mx, x, y, new_val);
        else
            update_x (vx * 2 + 1, Mx + 1, Rx, x, y, new_val);
    }
    update_y (vx, lx, rx, 1, 0, M-1, x, y, new_val);
}
```

Compression of two-dimensional tree segments

Suppose that the problem is the following: there n points in the plane defined by its coordinates (x_i, y_i) And receives requests like "count the number of points lying in a rectangle $((x_1, y_1), (x_2, y_2))$ ". It is understood that in the case of

this task becomes unnecessarily wasteful to build a two-dimensional tree to elements. Much of this memory will be wasted because each separate point can be reached only in the segments of the tree segments in the first coordinate, and therefore, the total "useful" $O(n^2)$

$O(\log n)$

the size of all segments of the trees to the second coordinate is the value $O(n \log n)$.

Then proceed as follows: each node of the tree segments in the first coordinate will be stored

segment tree built only on those second coordinates, which are found in the current interval
first coordinates. In other words, the construction of the tree segments within some vertex numbered
boundaries tx, trx we will consider only those points that fall in this segment $x \in [tx; trx]$,
and build a segment tree just above them.

Thus we ensure that every segment tree to the second coordinate will take as much

memory as it should. As a result, total **memory** reduced to $O(n \log n)$. **Respond**

on inquiry We will continue over $O(\log^2 n)$ Just now when you call the query from a tree on the segments second coordinate, we'll have to do a binary search on the second coordinate, but it will not worsen the asymptotic behavior.

But the payback would be the impossibility of making an arbitrary **modification request**: In fact, if a new point, it will lead to what we would have in any tree segments to the second coordinate add a new element in the middle that can not be done effectively.

In conclusion, we note that a concise manner described two-dimensional segment tree becomes almost **equivalent** above modifications dimensional tree segments (see "Saving all the subarray in each node of the tree segments"). In particular, it turns out that what is described here

two-dimensional segment tree - it's just a special case of the conservation of the subarray in each node of the tree where the subarray itself is stored as a tree segments. It follows that if you have to abandon the two-dimensional tree segments due to inability to perform one or another query, it makes sense to try to replace the embedded tree segments to any more powerful data structure, for example, **Cartesian tree**.

Tree to preserving the history of its values (improvement to persistent-data structure)

Persistent-data structure called a data structure such that at each modification remembers its previous state. This allows to apply to any version that we are interested in the data structure and execute its request.

Segment tree is one of those data structures that can be converted into a persistent-data structure (of course, we consider the persistent-efficient structure, and not one that copies the entire himself entirely before each update).

In fact, any request for change in the tree segments leads to a change in data $O(\log n)$ peaks, and along the way, starting from the root. So, if we keep the segment tree on signs (ie is a pointer to the left and right sons do pointers stored in the top), then the update query instead we should just change the existing vertices to create new vertices of which are direct links to the old top. Thus, when requesting an update will be created new peaks, including

$O(\log n)$

will create a new tree root segments, and all previous version of the tree, hanging over the old root, will remain unchanged.

Here is an example implementation for the simplest tree segments: when there is only a request for calculating the amount of subsegments and modification request singular.

```
struct vertex {
    vertex * l, * r;
    int sum;
    vertex (int val)
        : L(NULL), R(NULL), Sum(val)
    {}
    vertex (vertex * l, vertex * r)
        : L(l), R(r), Sum(0)
    {
        if (l) sum += l->
        sum; if (r) sum += r-
        > sum;
    }
};

vertex * build (int a[], int tl, int tr) {
    if (tl == tr)
        return new vertex (a[tl]);
    int tm = (tl + tr) / 2;
    return new vertex (
```

```

        build (a, tl, tm),
        Build (a, tm+1, Tr)
    );
}

int get_sum (vertex * t, int tl, int tr, int l, int r) {
    if (l> r)
        return 0;
    if (l == tl && tr == r)
        return t-> sum;
    int tm = (tl + tr) / 2;
    (t-> l, tl, tm, L,
    return get_summin(r,tm))
        + Get_sum (t-> r, tm+1, Tr, max(l,tm+1), R);
}

vertex * update (vertex * t, int tl, int tr, int pos, int new_val) {
    if (tl == tr)
        return new vertex (new_val);
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return new vertex (
            update (t-> l, tl, tm, Pos,
            new_val), T-> r
        );
    else
        return new vertex ( t-
            > l,
            update (t-> r, tm+1, Tr, pos, new_val)
        );
}

```

By using this approach can be turned into persistent-data structure virtually any segment tree.

Cartesian tree (treap, deramida)

Cartesian tree - a data structure that combines a binary search tree and binary heap (hence its name, and the second: treap (tree + heap) and deramida (wood + pyramid)).

More strictly, it is a data structure that stores a pair (X, Y) in the form of a binary tree, so that it is a binary search tree for x and binary pyramid by y. Assuming that all X and Y all are different, we see that if an element of the tree contains (X0, Y0), then all elements in left subtree $X < X_0$, all the elements in the right subtree $X > X_0$, and also in the left and right subtree, we have: $Y < Y_0$.

Deramidy been proposed sitting (Siedel) and Aragon (Aragon) in 1996

The advantages of such data organization

B that the application, which we consider (we consider deramidy as Cartesian tree - it is actually a more general data structure), Xs are the keys (and also the values stored in the data structure), and Y's - called **priorities**. If priority was, it would be

the usual binary search tree on X, and a given set of X's could fit a lot of trees, some of which are degenerate (eg, in the form of a chain), and therefore extremely slow (basic operations would be carried out in $O(N)$).

B Meanwhile, **priorities** allow **uniquely** specify the tree, which will be built (of course, does not depend on the order of addition of elements) (this is proved the corresponding theorem). Now it is obvious that if **randomly choose priorities**, Then we will build it

nondegenerate trees average case ensure that the asymptotic behavior $O(\log N)$ avg. Hence clear another name of this data structure - **randomized binary search tree**.

Operations

Thus, treap provides the following operations:

- **Insert (X, Y)** - for $O(\log N)$ on the average Performs the addition of a new element in the tree.

The variant in which the priority value Y is not passed to the function, and is chosen randomly (but bear in mind that it should not coincide with any other Y in the tree).

- **Search (X)** - for $O(\log N)$ on the average

Searches for an item with the specified key value X. implemented exactly the same as for a binary search tree.

- **Erase (X)** - for $O(\log N)$ on the average

Searches for an item and removes it from the tree.

- **Build (X1, ..., XN)** - for $O(N)$

Builds a tree from a list of values. This operation can be implemented in linear time (assuming that the values of X1, ..., XN sorted), but here, this implementation will not be discussed.

Here we will use only the simplest implementation - in the form of successive calls Insert, ie for the $O(N \log N)$.

- **Union (T1, T2)** - for $O(M \log(N/M))$ on the average

Combines two trees, on the assumption that all elements are distinct (though this operation can be implemented with the same asymptotic behavior, if the union need to remove duplicate items).

- **Intersect (T1, T2)** - for $O(M \log(N/M))$ on the average

Finds the intersection of two trees (ie their common elements). Here the implementation of this operation will not be considered.

In addition, due to the fact that the Cartesian tree is a binary search tree, and their meanings, to applicable thereto operations such as finding the K-th largest element and, conversely, the element number determination.

Description of the implementation

From an implementation standpoint, each cell contains X, Y, and pointers to the left L and right R son. To implement the operations need to implement two auxiliary operations: Split and Merge.

Split (T, X) - shared tree T two wood L and R (which is the return value) so manner that L contains all elements smaller key X, and R contains all the elements are large X. This operation is performed in $O(\log N)$. Its implementation is quite simple - the obvious recursion.

Merge (T1, T2) - combines two subtrees T1 and T2, and returns the new tree. This operation also implemented in $O(\log N)$. It works on the assumption that T1 and T2 have an appropriate order (all

X values in the first than in the second X values). Thus, we need to combine them so as not to upset the order of priorities for Y. To do this, simply select it as the root of a tree where Y is fundamentally more and recursively call itself from another tree and the corresponding son selected tree.

Now obvious implementation **Insert (X, Y)**. First down on a tree (as in a normal binary search tree on X), but stay on the first element, in which the priority value was less than Y. We found the position where we insert our element. Now call Split (X) of the found element (the element with all its subtree), and return it write L and R as the left and right child element to add.

Well understood and implementation **Erase (X)**. Go down the tree (as in a normal binary search tree on X), looking for the item to be removed. Find the items we simply call it Merge from the left and right sons, and its return value put in place the item to remove.

Operation **Build** implement in O (N log N) simply by successive calls Insert.

Finally, the operation **Union (T1, T2)**. Theoretically its asymptotic O (M log (N / M)), but in practice it works very well, probably with a very small hidden constant. Suppose, without loss of generality, T1-> Y > T2-> Y, ie, T1 is the root of the result. To get the result, we need to combine the trees T1-> L, T1-> R and T2 in two such tree, so that you can make sons T1. For this call Split (T2, T1-> X), thereby

we partition the T2 into two halves L and R, which are then recursively combine sons T1: Union (T1-> L, L) and Union (T1-> R, R), thus we construct the left and right subtrees result.

Implementation

Implement all the operations described above. Introduced here for the convenience of other designations - priority is indicated prior, values - key.

```
struct item {
    int key, prior;
    item * l, * r;
    item () {}
    item (int key, int prior): key (key), prior (prior), l (NULL), r (NULL) {}
};

typedef item * pitem;

void split (pitem t, int key, pitem & l, pitem & r)
{
    if (! t)
        l = r = NULL;
    else if (key <t-> key)
        split (t-> l, key, l, t-> l), r = t;
    else
        split (t-> r, key, t-> r, r), l = t;
}

void insert (pitem & t, pitem it)
{
    if (! t)
        t = it;
    else if (it-> prior > t-> prior)
        split (t, it-> key, it-> l, it-> r), t = it;
    else
        insert (it-> key <t-> key? t-> l: t-> r, it);
}

void merge (pitem & t, pitem l, pitem r)
{
    if (! l || ! r)
        t = l? l: r;
    else if (l-> prior > r-> prior)
        merge (l-> r, l-> r, r), t = l;
    else
        merge (r-> l, l, r-> l), t = r;
}

void erase (pitem & t, int key)
{
    if (t-> key == key)
        merge (t, t-> l, t-> r);
    else
        erase (key <t-> key? t-> l: t-> r, key);
}

pitem unite (pitem l, pitem r) {
```

```

if (!l || !r) return l? l: r;
if (l->prior <r->prior) swap (l, r);
pitem lt, rt;
split (r, l->key, lt,
rt); l->l = unite (l->l,
lt); l->r = unite (l->r,
rt); return l;
}

```

Support sizes of subtrees

To extend the functionality of the Cartesian tree, it is often necessary for each node to store the number of vertices in the subtree - int cnt certain field in the structure item. For example, it can easily

be found in O ($\log N$) K-th smallest element of a tree, or, conversely, for the same asymptotic know the item number in the sorted list (implementation of these operations will not differ from their sale to ordinary binary search trees).

If you change the tree (add or remove items, etc.) must vary accordingly and cnt some vertices. Define two functions - cnt () will return the current value of cnt or 0

if the node does not exist and function upd_cnt () will update the value of cnt for the specified vertex, provided that for her sons l and r these cnt already correctly updated. Then, of course, you can add function calls upd_cnt () at the end of each function, insert, erase, split, merge, to constantly maintain the correct values cnt.

```

int cnt (pitem t) {
    return t? t->cnt: 0;
}

void upd_cnt (pitem t)
{
    if (t)
        t->cnt = 1 + cnt (t->l) + cnt (t->r);
}

```

Construction of Cartesian tree in O (N) in the offline

TODO

Implicit Cartesian trees

Implicit Cartesian tree - it is a simple modification of the usual Cartesian tree, which, nevertheless, is a very powerful data structure. In fact, implicit Cartesian tree can be thought of as an array, which can be implemented over the following (all in O ($\log N$) online)

- Insert element in the array in any position
- Deleting an arbitrary element
- Sum, Min / Max on an arbitrary interval, etc.
- Addition, painting on a segment
- Coup (permutation of the elements in reverse order) on the interval

The key idea is that the key as keys to be used **Indexes** elements in the array. However, these values are clearly key store, we will not (otherwise, for example, when you insert an item would change the key in O (N) vertices of the tree).

Note that actually in the case of some key vertices - the number of vertices, it smaller. It should be noted that the vertices of the smaller, are not only in its left subtree, but possibly in the left subtree of her ancestors. More precisely, **implicit key** a vertex t equals the number of vertices cnt (t->l) in the left subtree of this node plus a value similar cnt (p->l) +1 for each ancestor of the vertex p, with the proviso that t is in the right subtree for p.

Clearly, as I quickly calculated for the current vertex its implicit key. As in all operations, we arrive at the top in any way down the tree, we can simply accumulate this amount, transferring its functions. If

we go to the left subtree - accumulated amount is not changed, and if we go to the right - increased by

cnt (t->l) +1. We present new implementations of functions split and merge:

```

void merge (pitem &t, pitem l, pitem r)
{
    if (!l || !r)
        t = l? l: r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
}

```

```

    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0)
{
    if (! t)
        return void (l = r = 0);
    int cur_key = add + cnt (t->l); // Calculate the
    implicit key if (key <= cur_key)
        split (t->l, l, t->l, key, add), r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt (t->l)), l =
    t; upd_cnt (t);
}

```

We now turn to the implementation of various miscellaneous operations on implicit Cartesian trees:

- **Inset element.**

Suppose we need to insert an element in the position pos. We divide the Cartesian tree into two halves: the proper array [0 .. pos-1] and array [pos .. sz]; it is enough to cause a split (t, t_1, t_2, pos). Then we can combine the tree t_1 with a new vertex; it is enough to cause a merge (t_1, t_1, new_item) (easy to see that all the preconditions to merge met). Finally, combine the two trees t_1 and t_2 back into a tree t -

calling merge (t, t_1, t_2).

- **Removal element.**

It is even easier: just find item to be removed, and then perform the merge for his sons l and r, and put in place the result of combining the top of t . In fact, the removal of the implicit Cartesian tree is different from the usual Cartesian removal of the tree.

- **Sum / minimum etc. on the interval.**

First, for each vertex will create an additional field f in the structure item, which will be stored in the objective function value for the subtree of this vertex. Such a field is easy to maintain, this should be analogous support sizes cnt (create a function that computes the value of the field, using his values for the sons and insert calls to this function at the end of all the functions that change the tree). Secondly, we need to learn to respond to a request for an arbitrary interval $[A; B]$. Learned to distinguish from a tree

part corresponding to the segment $[A; B]$. It is easy to understand that it is sufficient to cause at first split (t, t_1, t_2, A), and then split ($t_2, t_2, t_3, B-A+1$). As a result of the tree and t_2 will consist of all the elements in the interval $[A; B]$, and only them. Consequently, the answer to the query will be in the top of the field f t_2 . After answering the query tree should restore calls merge (t, t_1, t_2) and merge (t, t, t_3).

- **Addition / Paint on the interval.**

Here we proceed as in the previous paragraph, but instead of f will store field add , and which will contain the added value (or values, which paint the entire subtree of the vertex).

Before performing any operation should add this value to "push" - ie amended accordingly $t_l->add$ and $t_r->add$, and have a value add remove. Thus, we will ensure that no changes

tree information will not be lost.

- **Revolution on the interval.**

This item is almost identical to the previous - you need to enter a field bool rev , which put in a true, when want to make a revolution in the subtree of current node. "Push" field rev is that we exchange places sons current node, and set the flag for them.

Implementation. An example, the full realization of the implicit Cartesian tree with the coup on segment.

Here each vertex is stored as a field value - the actual value of the element present in the array at the current position. Also shows the implementation of the function output (), which displays an array corresponding to the current state of implicit Cartesian tree.

```

typedef struct item * pitem;
struct item {
    int prior, value, cnt;
    bool rev;
    pitem l, r;
};

int cnt (pitem it) {
    return it? it->cnt: 0;
}

void upd_cnt (pitem it)
{
    if (it)
        it->cnt = cnt (it->l) + cnt (it->r) + 1;
}

void push (pitem it) {
    if (it && it->rev) {it-
        >rev = false;
}

```

```

        swap (it-> l, it-> r);
        if (it-> l) it-> l-> rev ^ =
        true; if (it-> r) it-> r-> rev
        ^ = true;
    }

void merge (pitem & t, pitem l, pitem r)
    {push (l);
     push (r);
     if (! l | ! r)
         t = l? l: r;
     else if (l-> prior> r-> prior)
         merge (l-> r, l-> r, r), t = l;
     else
         merge (r-> l, l, r-> l), t =
     r; upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0)
    {if (! t)
        return void (l = r = 0);
     push (t);
     int cur_key = add + cnt (t->
     l); if (key <= cur_key)
         split (t-> l, l, t-> l, key, add), r = t;
     else
         split (t-> r, t-> r, r, key, add + 1 + cnt (t-> l)), l =
     t; upd_cnt (t);
}

void reverse (pitem t, int l, int r)
    {pitem t1, t2, t3;
     split (t, t1, t2, l);
     split (t2, t2, t3, r-l
     +1); t2-> rev ^ = true;
     merge (t, t1, t2);
     merge (t, t, t3);
}

void output (pitem t)
    {if (! t)
     return; push
     (t); output (t->
     l);
     printf ("% d", t->
     value); output (t-> r);
}

```

Modification of the stack and queue to retrieve the minimum in O (1)

Here we look at three things: the modification of the stack with the addition of extracting the smallest element in O (1), a similar modification of the queue, as well as their application to the problem of finding a minimum in all subsegments fixed length of this array for O (N).

Modification of the stack

Want to add the ability to extract the minimum from the stack in O (1), retaining the same asymptotic behavior of adding and removing elements from the stack.

For this will be stored in the stack is not the elements themselves, and couples: element and at least in the stack, starting with this element and below. In other words, if the stack is present as an array of vapor,

```
stack [i]. second = min {stack [j].  
    first} j = 0 .. i
```

It is clear that while finding a minimum throughout the stack will be just in taking values stack.top (). Second.

It is also obvious that when you add a new item to the stack second value will be equal to min (stack.top ()). Second, new_element). Removing an item from the stack is no different from the usual stack of removal as a removable element could not affect the second value for the remaining elements.

Implementation:

```
stack <pair <int,int>> st;
```

- Adding an element:

```
int minima = st.empty ()? new_element: min (new_element, st.top () .  
second); st.push (make_pair (new_element, minima));
```

- Removing elements:

```
int result = st.top () .  
first; st.pop ();
```

- Finding a minimum:

```
minima = st.top () . second;
```

Modification of the queue. Method 1

Here we consider a simple method for modifying the queue, but has the disadvantage that the modified turn really can only hold all the elements (ie, retrieving an item from the queue we will need to know the value of the element that we want to extract). Clearly, this is a very specific situation (usually just need a place to learn the next element, and not vice versa), but this method is attractive for its simplicity. Also, this method is applicable to the problem of finding a minimum in subsegments (see below).

The key idea is to actually be stored in all elements of the queue, and we need only to determine the minimum. Namely, let queue represents a non-decreasing sequence of numbers (ie, b head kept the smallest value), and, of course, is not arbitrary, but always containing a minimum. Then at least the entire queue will always be the first element of it. Before adding a new element

b all enough to make "Cut": while in the tail queue element is greater new element will remove the item from the queue; then add a new item to the end of the queue. Thus, on the one hand, do not disturb the order, and on the other hand, do not lose the current element, if it in any subsequent step would be the minimum. But when removing an element from the head of the queue it there, generally speaking, can not be - our modified turn could throw this element in the process of rebuilding. Therefore, when you delete an item we need to know the value of the item to retrieve - if the element with this value is in the head of the queue, then extract it; or simply do nothing.

Consider the implementation of the above transactions:

```
deque <int> q;
```

- Finding a minimum:

```
current_minimum = q.front();
```

- Adding an element:

```
while (!q.empty() && q.back() > added_element) q.pop_back();
q.push_back(added_element);
```

- Removing elements:

```
if (!q.empty() && q.front() == removed_element) q.pop_front();
```

It is clear that the average execution time of all these operations is O (1).

Modification of the queue. Method 2

Consider herein a method of modifying the other queue to extract low in O (1), which is slightly more complicated to implement, but devoid of the main drawbacks of the previous method, all elements of the actual queue stored therein and, in particular, when removing the element is not required to know the value of .

The idea is to reduce the problem to a problem on the stack, which has already been solved by us. Learn to simulate queue using two stacks.

Head of two stacks: s1 and s2; of course, refers to the stacks, modified for finding the minimum in O

(1). Add new elements will always stack s1, and retrieve items - only from the stack s2. In this case, if you try to retrieve the item from the stack s2 it was empty, just transfer all the elements of the stack to the stack s1

s2 (with the elements in the stack s2 are obtained already in the reverse order that we need to extract the elements; s1 same stack is empty). Finally, to find the minimum in the queue will actually be to find the minimum of the minimum stack s1 and minimum stack s2.

Thus, we perform all operations continue in O (1) (for the simple reason that every element in the worst case one time is added to the stack s1, 1 time transferred to the stack and s2 1 times popped s2).

Implementation:

```
stack<pair<int,int>> s1, s2;
```

- Finding a minimum:

```
if (s1.empty() || s2.empty())
    current_minimum = s1.empty() ? s2.top().second : s1.top().second;
else
    current_minimum = min(s1.top().second, s2.top().second);
```

- Adding an element:

```
int minima = s1.empty() ? new_element : min(new_element, s1.top().second);
s1.push(make_pair(new_element, minima));
```

- Removing elements:

```
if (s2.empty())
    while (!s1.empty()) {
        int element = s1.top().first;
        s1.pop();
        int minima = s2.empty() ? element : min(element, s2.top().second);
        s2.push(make_pair(element, minima));
    }
result = s2.top().first;
s2.pop();
```

The problem of finding a minimum in all subsegments fixed length of this array

Suppose we are given an array A of length N, and is given by M ≤ N. find the minimum required in each subsegments of length M of the given array, ie find:

$$\begin{array}{lllll} \min A[i], & \min A[i], & \min A[i], & \dots & \min A[i] \\ & & 2 \leq i \leq M & & \\ 0 \leq i \leq M-1 & 1 \leq i \leq M & +1 & & N-M \leq i \leq N-1 \end{array}$$

Solve this problem in linear time, ie O (N).

It's enough to make all modified for finding the minimum of O (1), which was considered by us above, and in this problem you can use any of the two methods of implementing a queue. Next, the solution is clear: to add to the queue first M of array elements, we find it and remove it at least, and then add the next item in the queue, and extract from it the first element of the array, again derive the minimum, etc. Since all operations are performed on the average queue in constant time, then the asymptotic behavior of the algorithm just get O (N).

It should be noted that the implementation of the modified queue first method is easier, however, for it would probably need to store the entire array (as on the i-th step we need to know the i-th and (iM)-th elements of array). When implementing the second stage of the method to store the array A is clearly not required - only to find another, i-th element of the array.

Randomized heap

Randomized heap (randomized heap) - it's a lot, which through the use of a random number generator allows you to perform all necessary operations in logarithmic expected time.

Heap is a binary tree, for each vertex of which is true, that the value in the top less than or equal value in all its descendants (it's a lot for a minimum; course, you can define a bunch of symmetrically to the maximum). Thus, the root is always a minimum heap.

Standard set of operations defined for the piles as follows:

- Adding an item
- Finding the minimum
- Removing the minimum (remove it from the tree and return its value)
- Merge two heaps (back pile containing elements of both heaps, duplicates are not removed)
- Deleting an arbitrary element (at a known position in the tree)

Randomized bunch lets you perform all of these operations within the expected time $O(\log n)$ at very simple implementation.

Data structure

Directly describe the data structure that describes a binary heap:

```
struct tree {
    T value;
    tree * l, * r;
};
```

In the top of the tree is stored value `value` a type for which the operator is defined comparison `operator <`. Additionally, pointers to the stored left and right Sons (which are equal to 0, if the corresponding son is missing).

Performing

It is easy to understand that all operations on a bunch reduced to a single operation: **merger** one of the two piles. Indeed, adding an element to a heap equivalent merge this heap with a heap consisting of a single element to be added. Finding the minimum does not require any action - is just the minimum root of the heap. Removing the minimum equivalent to the result of a merger pile replaced the left and right subtrees of the root. Finally, the removal of an arbitrary element similar to the removal of the minimum:

the entire subtree rooted at this vertex is replaced by the result of the merger of two sons-subtrees this vertex.

So we actually need to implement the merge operation only two piles, all other operations are reduced to trivial this operation.

Given two heaps and want to return their union. It is clear that the root of each of these piles minima are blocked so radically resultant heap will be a minimum of these two values. So, we compare the root of what is heaps less important it is placed in the root of the result, and now

we must unite sons selected vertex with the remaining bunch. If we choose some feature of one of the two sons, then we will have to simply combine the subtree root with this son of a heap. Thus, we again come to a merge operation. Sooner or later, this process will stop (it will take, of course, no more than the sum of the heights of piles).

Thus, to achieve an average logarithmic asymptotics, we need to specify the method of selecting one of the two sons, so that the average length of the path traveled by the order would be obtained from the logarithm of the number of elements in the heap. It is not difficult to guess that make this choice, we will accidentally Thus, the implementation of the merge operation is obtained as follows:

```
tree * merge (tree * t1, tree * t2) {
    if (! T1 || ! T2)
        return t1? t1: t2;
    if (t2-> value <t1-> value)
        swap (t1,
t2);if (rand() & 1)
        swap (t1-> l, t1-> r);
```

```

t1-> l = merge (t1-> l,
t2);return t1;
}

```

It first checks if at least one of the piles is empty, then no action is not necessary to produce fusion. Otherwise, we do to heap t_1 was a bunch with a smaller value in the root (which exchange the t_1 and t_2 if necessary). Finally, we believe that the second bunch t_2 will merge with the left child of the root heap t_1 . So we randomly exchange the left and right children, and then execute the merger left child and a second heap.

Asymptotics

Introduce the random variable $h(T)$ denoting **length of random paths** from root to leaf (length in number of edges). It is understood that the algorithm `merge` performed for operations. Therefore, to study the asymptotic behavior of the algorithm is necessary to investigate the random variable $h(T)$.

Expectation value

It is argued that the expectation $h(T)$ is bounded above by the logarithm of the number of n vertices in this pile:

$$Eh(T) \leq \log(n+1)$$

- Respectively the left and right subtrees of the root of the heap,
an L R and T n_L
 n_R - The number of vertices in them (of course, $n = n_L + n_R + 1$).

Then we have:

$$\begin{aligned} Eh(T) &= 1 + \frac{1}{2}(Eh(L) + Eh(R)) \leq 1 + \frac{1}{2}(\log(n_L + 1) + \log(n_R + 1)) = \\ &= 1 + \log \sqrt{(n_L + 1)(n_R + 1)} = \log 2\sqrt{(n_L + 1)(n_R + 1)} \leq \\ &\leq \log \frac{2((n_L + 1) + (n_R + 1))}{2} = \log(n_L + n_R + 2) = \log(n + 1) \end{aligned}$$

QED.

Exceeding the expected value

We prove that the probability of exceeding the estimate obtained above is small:

$$P\{h(T) > (c+1)\log n\} < \frac{1}{n^c}$$

for any positive constant c .

We denote the set of paths from the root to the leaf pile, the length of which exceeds $(c+1)\log n$. Note that for any path length p likelihood as random path will be selected because it equal $2^{-|p|}$. Then we obtain:

QED.

Asymptotic behavior of the algorithm

Thus, the algorithm `merge` And, therefore, all other operations expressed through it, and runs in $O(\log n)$ avg.

Moreover, for any positive constant ϵ there exists a positive constant C that the likelihood that the operation would require more than operations, less (in some sense describes the worst behavior of the algorithm).

Problem RMQ (Range Minimum Query - at least on the interval)

Given an array A [1 .. N]. Receives requests form (L, R), for each query requires a minimum in the array A, starting at position L and ending at R.

Apps

In addition to direct application in a variety of tasks are the following:

- [Task LCA](#)
(lowest common ancestor)

Decision

RMQ problem is solved by data structures.

Described on the website of the data structures you can choose:

- **Sqrt-decomposition** - responds to a request for O (\sqrt{N}), preprocessing for O (N). The advantage is that it is very simple data structure. Disadvantage - asymptotics.
- **Segment tree** - responds to a request for O ($\log N$), preprocessing for O (N). Advantage - good asymptotic behavior. Disadvantage - a lot of code compared to other data structures.
- **Fenwick tree** - responds to a request for O ($\log N$), preprocessing for O ($N \log N$)
Advantage - written very quickly and works very fast too. But a significant drawback - Fenwick tree can only respond to requests from the L = 1, which is applicable to many applications.

Note. "Preprocessing" - a pre-processing of the array A, it actually build a data structure for this array.

Now suppose that the array A **may vary** during operation (ie, will also do requests for changes in values in an interval [L; R]).

Then, the resulting problem can be solved by

[Sqrt-decomposition](#) and [Tree segments](#).

Finding of the longest increasing subsequence

Condition of the problem following. An array of n Numbers: $a[0 \dots n - 1]$ locate in this sequence is strictly increasing subsequence of greatest length.

Formally it looks as follows: required to find a sequence of indices $i_1 \dots i_k$ That:

$$i_1 < i_2 < \dots < i_k, \\ a[i_1] < a[i_2] < \dots < a[i_k].$$

This article discusses the various algorithms to solve this problem, as well as some of the tasks that can be reduced to this problem.

Solution for $O(n^2)$: Dynamic programming method

Dynamic programming - it is a very common technique, allowing to solve a huge class of problems. Here we consider this technique as applied to our specific problem.

Learn to look first for **length** of the longest increasing subsequence, and recovering the subsequence Let us later.

Dynamic programming to find the length of the response

To do this, let's learn how to consider an array where $-$ is the length of the longest increasing subsequence ending exactly at index. This array (he is -)

dynamics itself) will be assumed gradually: first $d[0]$. Then $d[1]$ etc. In the end, when the array is calculated by us, the answer to the problem will be equal to the maximum in an array $d[]$.

So, let the current index - ie we want to calculate the value $d[i]$ And all previous values $d[0] \dots d[i - 1]$ already counted. Then we note that we have two choices:

- or the desired subsequence consists only of the number $a[i]$. Then before the i dynamics itself will be assumed gradually: first $d[0]$. Then $d[1]$ etc. In the end, when the array is calculated by us, the answer to the problem will be equal to the maximum in an array $d[]$.
- or $d[i] = 1$ number in the desired subsequence worth a different number. Let's iterate through this number: it can be any element $a[j]$ But such that $a[j] < a[i]$. Because the dynamics of $d[j] + 1$ for it already calculated, it turns out that this number $a[j]$ number of answers $d[j] + 1$. Thus, formula:

By combining these two one embodiment obtain a final algorithm for calculating $d[i]$.

$$d[i] = \max \left(1, \max_{\substack{j=0 \dots i-1, \\ a[j] < a[i]}} (d[j] + 1) \right).$$

This algorithm - and there are very dynamic.

Implementation

We present the implementation of the algorithm described above, which finds and displays the length of the longest increasing subsequence:

```
int d[MAXN]; // constant MAXN equal to the largest possible value of n

for (int i = 0; i < n; ++
    i) { d[i] = 1;
```

```
for (int j =0; j <i; + +
J) if (a[j]
<A[i])
    d[i] = Max (d[i],1 + D[j]);
}
```

```

int ans = d[0];
for (int i = 0; i <n; + + I)
    ans = max (ans,
d[i]);cout << Ans << endl;

```

Recovery Answer

While we only have learned to look for the long answer, but very of the longest subsequence, we can not deduce, since it does not maintain any additional information about where the maximum occurs.

To be able to restore the response, in addition to the

dynamics of $d[0 \dots n - 1]$ also need to keep an auxiliary

array $p[0 \dots n - 1]$. That, in any location peaked for each value

$p[i]$ will denote the same index for which to obtain the greatest value

dynamic programming is often referred to as "ancestors array".

Then, in order to deduce the answer, you just have to go on the element with the maximum

value

$d[i]$. In other words, the index

$d[i]$. (This array $p[]$

Then, in order to deduce the answer, you just have to go on the element with the maximum

value

long as we do not derive all subsequence, ie until we reach the element with a value of

$d = 1$.

Implementation of the recovery response

So, we will change the code of the speakers, and add the code that produces the conclusion of the longest subsequence (output indexes to a subsequence, the 0-indexed).

For convenience, we initially put indexes: for elements whose dynamics was found to be unity, this value will remain ancestor

minus one that a little bit easier when you restore an answer.

```

int d[MAXN], P[MAXN]; // constant MAXN equal to the largest possible value of
n

for (int i = 0; i <n; + +
I) { d[i] = 1;
p[i] = -1;
for (int j = 0; j <i; + +
J) if (a[j]
<A[i])
    if (1 + D[j] > D[i]) {
        d[i] = 1 + D[j];
        p[i] = J;
    }
}

int ans = d[0], Pos = 0;
for (int i = 0; i <n; + +
I)
    if (d[i] > Ans) {
        ans = d[i];
        pos = i;
    }
cout << Ans << endl;

vector <int> Path;
while (pos != -1) {
    path.push_back (pos);
    pos = p[pos];
}
reverse (path.begin(), Path.end());for
(int i = 0; i <(int)path.size(); + + I)
    cout << Path[i] << ' ';

```

An alternative way to restore response

However, as is almost always the case in dynamic programming to restore the response can not store an additional array of ancestors, and simply re-counting the current element dynamics and looking for

$p[]$ which was also the index reached the maximum.

This method leads to the implementation of a little more than a long code, but instead we get the memory savings and absolute coincidence logic program in the counting process and the dynamics of the recovery process.

Solution for $O(n \log n)$: Dynamic Programming

with binary search

To get a quick solution to the problem, we construct another option for dynamic programming,

$O(n^2)$

and then understand how this option can be accelerated to $O(n \log n)$.

Dynamics now becomes: let $d[i](i = 0 \dots n)$ - is a number which ends increasing subsequence of length i (And if some of these numbers - then the least of them). Initially, we believe $d[0] = -\infty$ And all other elements $d[i] = \infty$.

Consider this trend, we will gradually, by treating $a[0]$ Then $a[1]$ Etc. We

present the implementation of this dynamic for $O(n^2)$.

```
int d[MAXN];
d[0] = -INF;
for (int i = 1; i <= n; +
    + i) d[i] = INF;
for (int i = 0; i < n; i++) {
    for (int j = 1; j <= n; j++)
        if (d[j-1] < A[i] && A[i] < d[j])
            d[j] = A[i];
```

Now note that this dynamic is one **a very important property**:

all $i = 1 \dots n$. Another property - that every element $a[i]$ updates the maximum single cell $d[i]$ for $d[j]$.
Thus, it means that the next process $a[i]$ we can for $O(\log n)$ Making a binary search
the array. In fact, we are just looking in the array $d[]$ than $a[i]$ And
try to update this element is similar to the above realization.

Implementation for $O(n \log n)$

Using standard in the language C++ binary search algorithm (which returns the position of the first $upper_bound$ element is strictly greater part), we obtain a simple implementation:

```
int d[MAXN];
d[0] = -INF;
for (int i = 1; i <= n; +
    + i) d[i] = INF;
for (int i = 0; i < n; i++) {
    int j = int(upper_bound(d.begin(), d.end(), A[i]) - d.begin());
    if (d[j-1] < A[i] && A[i] < d[j])
        d[j] = A[i];
}
```

Recovery Answer

For such dynamics also can restore the answer, which again, in addition to speakers also need to store an array of "ancestors" - is the element with which the index ends optimal subsequence of length i . In addition, for each element of the array will have to keep it "ancestor" - ie the index of the element that must precede $a[i]$ in the optimal subsequence.

By keeping these two arrays during the dynamics calculations, at the end it will be easy to restore the desired subsequence.

(It is interesting to note that with regard to the dynamics of this response can be restored only way through arrays ancestors - and without them recover after calculating the response dynamics will be impossible. This is one of the rare cases where the dynamics of an alternative method of recovery is not applicable - no arrays ancestors).

Solution for $O(n \log n)$: Data structures

If the above method for a very beautiful, but it's not trivial ideological, then there is another way:
 $O(n \log n)$ use one of the well-known simple data structures.

In fact, let's go back to the very first dynamic, where the state is simply the current position.

The current value is calculated as the dynamics of the maximum value of all elements i , that $a[j] < a[i]$.

Consequently, if we through $t[]$ denote such array in which we write the value of the dynamics of the numbers:

$$t[a[i]] = d[i],$$

it turns out that all that we need to be able to - to look for it **maximum prefix** array

$$t[0 \dots a[i] - 1].$$

The task of finding the maximum on a prefix of the array (given the fact that the array can be changed) solved many standard data structures, for example, tree segments or Fenwick tree.

Using any such data structure, we obtain a solution for $O(n \log n)$.

This method has obvious solutions **limitations**: The length and the complexity of this path is in any case worse than described above for the dynamics. Furthermore, if the input number may be large enough, it is likely they will have to compress (ie renumbering of 0 to $n - 1$) - Without this, many data structures can not work due to the high memory consumption.

On the other hand, there is a current path **Benefits**. First, with this method of solution does not have to worry about the tricky dynamics. Secondly, this method allows us to solve our problem, some generalizations (which are explained below).

Related tasks

We present here a few problems closely related to the task of finding of the longest increasing subsequence.

Of the longest non-decreasing subsequence

In fact, it's the same problem, only now in the desired subsequence allowed the same number (ie, we must find a weakly increasing subsequence).

The solution to this problem is essentially no different from our original problem by simply changing the sign comparisons inequalities, and will have to slightly modify the binary search.

Number of the longest increasing subsequence

To solve this problem, you can use the very first performance over $O(n^2)$ either approach using data structures for solutions for $O(n \log n)$. In fact, in case all changes are only, that besides the dynamics values $[i]$ also need to keep the number of ways that value could be obtained.

Apparently, the way to solve through the dynamics for $O(n \log n)$ to this problem can not be applied.

The smallest number of non-increasing subsequences covering this sequence

Condition such. An array of n properties a requires paint it in the fewest number of colors so that for each color would be obtained non-increasing subsequence.

Decision. Allegedly the minimum number of colors is equal to the length required of the longest increasing subsequence.

Proof. In fact, we must prove **duality** this problem and the problem search of the longest increasing subsequence.

We denote x length of the longest increasing subsequence, and through y - The required minimum number of non-increasing subsequences. We must prove that $x = y$.

On the one hand, it is understandable why there can not be $y < x$ Because if we have a strictly increasing elements that no two of them could not get into a non-increasing subsequence, and hence,

$$y \geq x .$$

We now show that, on the contrary, can not be $y > x$. We prove this by contradiction: we assume that $y > x$. Then consider any optimal set of y non-increasing subsequences. Transform this set as follows: as long as there are two such subsequence that begins before the first second, but first begin with a number greater than or equal than the start of the second - to unhook it from the start by the first subsequence and the trailer to start the second. Thus, after a finite number of steps we will have subsequences, and their numbers will start to form increasing subsequence of length y . But $y > x$ we have a contradiction (in fact can not be increasing subsequence of length x).

Thus, in fact, $y = x$, As required.

Recovery Answer. Allegedly itself that the desired partition into subsequences can look eagerly, ie going from left to right and assigning the current number in that subsequence, which now ends at the smallest number greater than or equal to the current.

Problem in online judges

List of tasks that can be solved on the subject:

- MCCME # 1793 "**The largest increasing subsequence for O (n * log (n))**" [Difficulty: Easy]
- TopCoder SRM 278 "**500 IntegerSequence**" [Difficulty: Easy]
- TopCoder SRM 233 "**DIV2 1000 AutoMarket**" [Difficulty: Easy]
- Ukrainian School Olympiad in Informatics - Target F "**The Tourist**" [Difficulty: Medium]
- Codeforces Beta Round # 10 - D problem "**LCIS**" [Difficulty: Medium]
- ACM.TJU.EDU.CN 2707 "**Greatest Common Increasing Subsequence**" [Difficulty: Medium]
- SPOJ # 57 "**SUPPER. Supernumbers in a permutation**" [Difficulty: Medium]
- ACM.SGU.RU # 521 "**North-East**" [Difficulty: high]
- TopCoder Open 2004 - Round 4 - "**1000. BridgeArrangement**" [Difficulty: high]

K-th order statistic in O (N)

Suppose we are given an array A of length N, and let there be given by K. The challenge is to find in the array K-th largest number, ie K-th order statistic.

The basic idea - the idea to use quicksort. Actually, the algorithm is simple, difficult to prove that it runs in average O (N), in contrast to quicksort.

Implementation of a non-recursive function:

```
template <class T>
T order_statistics (std :: vector <T> a, unsigned n, unsigned k)
{
    using std :: swap;
    for (unsigned l = 1, r = n;;)
    {

        if (r <= l +1)
        {
            // current portion consists of 1 or 2 elements -
            // can easily find the answer
            if (r == l +1 && a [r] <a
                [l]) swap (a [l], a
                [r]);
            return a [k];
        }

        // we order a [l], a [l +1], a [r]
        unsigned mid = (l + r) >> 1;
        swap (a [mid], a [l
        +1]); if (a [l]> a
        [r])
            swap (a [l], a
        [r]); if (a [l +1]> a [r])
            swap (a [l +1], a
        [r]); if (a [l]> a [l +1])
            swap (a [l], a [l +1]);

        // perform division
        // barrier is a [l +1], ie, median of a [l], a [l +1],
a [r]
        unsigned
            i = l
            +1, j =
            r;
        const T
            cur = a [l +1];
        for (; ;)
        {
            while (a [+ + i]
<cur); while (a [- j]>
cur); if (i> j)
                break;
            swap (a [i], a
[j]);
        }

        // barrier insert a
[l +1] = a [j];
a [j] = cur;

        // continue to work in that part
        // which must contain the required element
        if (j> = k)
```

```
r = j-1;  
if (j <= k)  
    l = i;
```

}

It should be noted that in the standard C + + library, this algorithm has already been implemented - it is called nth_element.

Finding of the longest increasing subsequence

Condition of the problem following. An array of n Numbers: $a[0 \dots n - 1]$ locate in this sequence is strictly increasing subsequence of greatest length.

Formally it looks as follows: required to find a sequence of indices $i_1 \dots i_k$ That:

$$i_1 < i_2 < \dots < i_k, \\ a[i_1] < a[i_2] < \dots < a[i_k].$$

This article discusses the various algorithms to solve this problem, as well as some of the tasks that can be reduced to this problem.

Solution for $O(n^2)$: Dynamic programming method

Dynamic programming - it is a very common technique, allowing to solve a huge class of problems. Here we consider this technique as applied to our specific problem.

Learn to look first for **length** of the longest increasing subsequence, and recovering the subsequence Let us later.

Dynamic programming to find the length of the response

To do this, let's learn how to consider an array where $-$ is the length of the longest increasing subsequence ending exactly at index. This array (he is -)

dynamics itself) will be assumed gradually: first $d[0]$. Then $d[1]$ etc. In the end, when the array is calculated by us, the answer to the problem will be equal to the maximum in an array $d[]$.

So, let the current index - ie we want to calculate the value $d[i]$ And all previous values $d[0] \dots d[i - 1]$ already counted. Then we note that we have two choices:

- or ie desired subsequence consists only of the number $a[i]$. Then before the i number $a[i]$ in the desired subsequence worth a different number. Let's iterate through this number: it can be any element $a[j]$ But such that $d[i] = 1$ $a[i] > a[j]$. Because the dynamics of $d[i]$ for it already $d[i] > 1$. Let us consider some current index $a[i]$ together with the $a[j] (j = 0 \dots i - 1)$ answers $d[j] + 1$. Thus, $a[i] < a[j]$ calculated, it turns out that this number $a[j]$ number of $a[i]$ $d[j] + 1$. $d[i]$ can be considered on the following formula:

By combining these two one embodiment obtain a final algorithm for calculating $d[i]$.

$$d[i] = \max \left(1, \max_{\substack{j=0 \dots i-1, \\ a[j] < a[i]}} (d[j] + 1) \right).$$

This algorithm - and there are very dynamic.

Implementation

We present the implementation of the algorithm described above, which finds and displays the length of the longest increasing subsequence:

```
int d[MAXN]; // constant MAXN equal to the largest possible value of n

for (int i = 0; i < n; ++
    i) { d[i] = 1;
```

```
for (int j =0; j <i; + +
J) if (a[j]
<A[i])
    d[i] = Max (d[i],1 + D[j]);
}
```

```

int ans = d[0];
for (int i = 0; i <n; + + I)
    ans = max (ans,
d[i]);cout << Ans << endl;

```

Recovery Answer

While we only have learned to look for the long answer, but very of the longest subsequence, we can not deduce, since it does not maintain any additional information about where the maximum occurs.

To be able to restore the response, in addition to the

dynamics of $d[0 \dots n - 1]$ also need to keep an auxiliary

array $p[0 \dots n - 1]$. That, in any location peaked for each value

$p[i]$ will denote the same index for which to obtain the greatest value

dynamic programming is often referred to as "ancestors array".

Then, in order to deduce the answer, you just have to go on the element with the maximum

value

$d[i]$. In other words, the index

$d[i]$. (This array $p[]$

Then, in order to deduce the answer, you just have to go on the element with the maximum

value

long as we do not derive all subsequence, ie until we reach the element with a value of

$d = 1$.

Implementation of the recovery response

So, we will change the code of the speakers, and add the code that produces the conclusion of the longest subsequence (output indexes to a subsequence, the 0-indexed).

For convenience, we initially put indexes: for elements whose dynamics was found to be unity, this value will remain ancestor

minus one that a little bit easier when you restore an answer.

```

int d[MAXN], P[MAXN]; // constant MAXN equal to the largest possible value of
n

for (int i = 0; i <n; + +
I) { d[i] = 1;
p[i] = -1;
for (int j = 0; j <i; + +
J) if (a[j]
<A[i])
    if (1 + D[j] > D[i]) {
        d[i] = 1 + D[j];
        p[i] = J;
    }
}

int ans = d[0], Pos = 0;
for (int i = 0; i <n; + +
I)
    if (d[i] > Ans) {
        ans = d[i];
        pos = i;
    }
cout << Ans << endl;

vector <int> Path;
while (pos! = -1) {
    path.push_back (pos);
    pos = p[pos];
}
reverse (path.begin(), Path.end());for
(int i = 0; i <(int)path.size(); + + I)
    cout << Path[i] << ' ';

```

An alternative way to restore response

However, as is almost always the case in dynamic programming to restore the response can not store an additional array

of ancestors, and simply re-counting the current element dynamics and looking for

which was also the index reached the maximum.

This method leads to the implementation of a little more than a long code, but instead we get the memory savings and

absolute coincidence logic program in the counting process and the dynamics of the recovery process.

Solution for $O(n \log n)$: Dynamic Programming

with binary search

To get a quick solution to the problem, we construct another option for dynamic programming,

$O(n^2)$

and then understand how this option can be accelerated to $O(n \log n)$.

Dynamics now becomes: let $d[i](i = 0 \dots n)$ - is a number which ends increasing subsequence of length i (And if some of these numbers - then the least of them). Initially, we believe $d[0] = -\infty$ And all other elements $d[i] = \infty$.

Consider this trend, we will gradually, by treating $a[0]$ Then $a[1]$ Etc. We

present the implementation of this dynamic for $O(n^2)$.

```
int d[MAXN];
d[0] = -INF;
for (int i = 1; i <= n; +
    + i) d[i] = INF;
for (int i = 0; i < n; i++) {
    for (int j = 1; j <= n; j++)
        if (d[j-1] < A[i] && A[i] < d[j])
            d[j] = A[i];
```

Now note that this dynamic is one **a very important property**:

all $i = 1 \dots n$. Another property - that every element $a[i]$ updates the maximum single cell $d[i]$ for $d[j]$.
Thus, it means that the next process $a[i]$ we can for $O(\log n)$ Making a binary search
the array. In fact, we are just looking in the array $d[]$ than $a[i]$ And
try to update this element is similar to the above realization.

Implementation for $O(n \log n)$

Using standard in the language C++ binary search algorithm (which returns the position of the first $upper_bound$ element is strictly greater part), we obtain a simple implementation:

```
int d[MAXN];
d[0] = -INF;
for (int i = 1; i <= n; +
    + i) d[i] = INF;
for (int i = 0; i < n; i++) {
    int j = int(upper_bound(d.begin(), d.end(), A[i]) - d.begin());
    if (d[j-1] < A[i] && A[i] < d[j])
        d[j] = A[i];
}
```

Recovery Answer

For such dynamics also can restore the answer, which again, in addition to speakers also need to store an array of "ancestors" - is the element with which the index ends optimal subsequence of length i . In addition, for each element of the array will have to keep it "ancestor" - ie the index of the element that must precede $a[i]$ in the optimal subsequence.

By keeping these two arrays during the dynamics calculations, at the end it will be easy to restore the desired subsequence.

(It is interesting to note that with regard to the dynamics of this response can be restored only way through arrays ancestors - and without them recover after calculating the response dynamics will be impossible. This is one of the rare cases where the dynamics of an alternative method of recovery is not applicable - no arrays ancestors).

Solution for $O(n \log n)$: Data structures

If the above method for a very beautiful, but it's not trivial ideological, then there is another way:
 $O(n \log n)$ use one of the well-known simple data structures.

In fact, let's go back to the very first dynamic, where the state is simply the current position.

The current value is calculated as the dynamics of the maximum value of all elements i , that $a[j] < a[i]$.

Consequently, if we through $t[]$ denote such array in which we write the value of the dynamics of the numbers:

$$t[a[i]] = d[i],$$

it turns out that all that we need to be able to - to look for it **maximum prefix** array

$$t[0 \dots a[i] - 1].$$

The task of finding the maximum on a prefix of the array (given the fact that the array can be changed) solved many standard data structures, for example, tree segments or Fenwick tree.

Using any such data structure, we obtain a solution for $O(n \log n)$.

This method has obvious solutions **limitations**: The length and the complexity of this path is in any case worse than described above for the dynamics. Furthermore, if the input number may be large enough, it is likely they will have to compress (ie renumbering of 0 to $n - 1$) - Without this, many data structures can not work due to the high memory consumption.

On the other hand, there is a current path **Benefits**. First, with this method solutions do not have to worry about tricky dynamics. Secondly, this method allows us to solve our problem, some generalizations (which are explained below).

Related tasks

We present here a few problems closely related to the task of finding of the longest increasing subsequence.

Of the longest non-decreasing subsequence

In fact, it's the same problem, only now in the desired subsequence allowed the same number (ie, we must find a weakly increasing subsequence).

The solution to this problem is essentially no different from our original problem by simply changing the sign comparisons inequalities, and will have to slightly modify the binary search.

Number of the longest increasing subsequence

To solve this problem, you can use the very first performance over $O(n^2)$ either approach using data structures for solutions for $O(n \log n)$. In fact, in case all changes are only, that besides the dynamics value $[i]$ also need to keep the number of ways that value could be obtained.

Apparently, the way to solve through the dynamics for $O(n \log n)$ to this problem can not be applied.

The smallest number of non-increasing subsequences covering this

Condition such. An array of n properties, a requires paint it in the fewest number of colors so that for each color would be obtained non-increasing subsequence.

Decision. Allegedly the minimum number of colors is equal to the length required of the longest increasing subsequence.

Proof. In fact, we must prove **duality** this problem and the problem search of the longest increasing subsequence. We denote x length of the longest increasing subsequence, and through y - The required minimum number of non-increasing subsequences. We must prove that $x = y$.

On the one hand, it is understandable why there can not be $y < x$ Because if we have a strictly increasing elements that no two of them could not get into a non-increasing subsequence, and hence, $y \geq x$.

We now show that, on the contrary, can not be $y > x$. We prove this by contradiction: we assume that $y > x$. Then consider any optimal set of y non-increasing subsequences. Transform this set as follows: as long as there are two such subsequence that begins before the first second, but first begin with a number greater than or equal than the start of the second - to unhook it from the start by the first subsequence and the trailer to start the second. Thus, after a finite number of steps we will have subsequences, and their numbers will start to form increasing subsequence of length y . But $y > x$ we have a contradiction (in fact can not be increasing subsequence of length x).

Thus, in fact, $y = x$, As required.

Recovery Answer. Allegedly itself that the desired partition into subsequences can look eagerly, ie going from left to right and assigning the current number in that subsequence, which now ends at the smallest number greater than or equal to the current.

Dynamics of the profile. Task "parquet"

Typical problems of the dynamics of the profile are:

- find a number of ways of tiling field certain figures
- find a tiling with the fewest figures
- find a tiling with a minimum amount of unused cells
- find a tiling with a minimum number of shapes such that it is impossible to add another shape

Task "Parquet"

There is a rectangular area the size NxM. Need to find a number of ways to pave this area figures 1x2 (empty cells should not remain, figures should not overlap).

Construct a dynamics: D [I] [Mask], where I = 1 .. N, Mask = 0 .. 2 ^ M-1. I denotes the number of lines in the current field, and Mask - profile the last line in the current field. If the j-th bit in the Mask is equal to zero, then the spot profile extends at a "normal level", and if 1 - here "seizure" depth 1. answer, obviously, is D [N] [0].

Build this momentum going, just going through all the I = 1 .. N, all masks Mask = 0 .. 2 ^ M-1, and for each mask will skip forward, ie add to it a new shape in all possible ways.

Implementation:

```
int n, m;
vector <vector <long long>> d;

void calc (int x = 0, int y = 0, int mask = 0, int next_mask = 0)
{
    if (x == n)
        return;
    if (y >= m)
        d [x + 1] [next_mask] += d [x] [mask];
    else
    {
        int my_mask = 1 << y;
        if (mask & my_mask)
            calc (x, y + 1, mask, next_mask);
        else
        {
            calc (x, y + 1, mask, next_mask | my_mask);
            if (y + 1 < m && ! (mask & my_mask) && ! (mask &
                (My_mask << 1)))
                calc (x, y + 2, mask, next_mask);
        }
    }
}

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

    d.resize (n + 1, vector <long long> (1 <<
        m)); d [0] [0] = 1,
    for (int x = 0; x < n; + + x)
        for (int mask = 0; mask < (1 << m); +
            + mask) calc (x, 0, mask, 0);
    cout << d [n] [0];
}
```

Finding the greatest zero submatrix

Given a matrix a size $n \times m$. It is required to find a submatrix consisting of all zeros and all of these - having the largest area (sub-matrix - a rectangular area of the matrix).

Trivial algorithm - fingering desired submatrix - even with the implementation of good will work $O(n^2m^2)$. Below we describe an algorithm that works for $O(nm)$ in linear relative size of the matrix time.

Algorithm

To resolve ambiguities immediately observe that n equals the number of rows of the matrix a . Respectively, m - is number of columns. Elements of the matrix will be numbered in 0-Indexing, ie in the designation

$a[i][j]$

Indexes i

j run ranges $i = 0 \dots n - 1, j = 0 \dots m - 1$.

Step 1: Auxiliary speaker

Please calculate the following auxiliary dynamics: $d[i][j]$ - On top of the unit to the nearest element $a[i][j]$. Formally speaking $d[i][j]$ equal to the greatest number of the row (row of the range of -1 to i), Wherein in j -Th column, unit costs. In particular, if there is no such string, then $d[i][j]$ is assumed to be -1 (It can be understood as that the whole matrix as limited outside units).

This momentum moving easily read the matrix from the top down: let us stands i Second line, and we know the value of the dynamics for the previous line. Then simply copy these values to the dynamics of the current line, changing only those elements which are in the matrix unit. It is understood that did not even need to store the entire rectangular matrix dynamics, and need only one array size m :

```
vector<int> D (m, -1);
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j)
        if (a[i][j] == 1)
            d[j] = i;
    // D calculated for the i-th row, we can use these values here
}
```

Step 2: The solution

Now we can solve the problem in - just to sort out the current line number of the left and right columns of the desired sub-matrix and using dynamics to calculate the upper limit of zero submatrix. However, you can go ahead and significantly improve the asymptotic behavior of solutions.

It is clear that the desired zero submatrix is bounded on all four sides by some Ones (or boundaries field) - that and prevent it increase in size and improve response. Therefore, it is argued, we will not miss answer if we proceed as follows: first, let's look over the bottom line number zero submatrix, then iterate over which column we balk up zero submatrix. Using i value We immediately obtain the number of the top line zero submatrix. You're now determine $d[i][j]$ optimal left and right boundaries zero submatrix - ie maximum push this submatrix of the left and right of the first column.

What does it mean to push the far left? It means to find an index k_1 such that $d[i][k_1] > d[i][j]$, wherein k_1 - The closest such left to index. It is understood that if $k_1 + 1$ column desired zero submatrix. If there are no such index, then put $k_1 = -1$ (This means that we could extend the current zero submatrix way to the left - to the border of the entire matrix a).

for the right border: this is the closest to the right of the index such (Or, if such an index is not).
Symmetrically can identify index that Thus, the indices and, if we learn to look for them effectively, will give us all the necessary information on

$$d[i][k_2] > d[i][j] \quad m \quad k_2$$

J

$$k_1 \quad k_2$$

current zero submatrix. In particular, an area equal to $(i - d[i][j]) \cdot (k_2 - k_1 - 1)$.

How to look for these same indices and effectively at fixed *i* and? We will satisfy only the asymptotic behavior $O(1)$, At least on average.

Achieve such asymptotics can use the stack (stack) as follows. Learn to look first for the index, and preserve its value for each index j within the current line i dynamics. To do this, we will see all the columns from left to right, and the head of a stack, which will always lie only

k_1

. It is clear that the transition from the columns in which the value dynamics $d[i][j]$ more strictly $d[i][j]$ column the next column $j + 1$ want to update the contents of the stack. It is alleged that requires first put in a column stack (a stack because the "good"), and then, until the top of the stack is the wrong element (ie, whose value $d \leq d[i][j + 1]$) - Get this item. Easy to understand that removed from the stack just enough of its vertices, and from any other locations it (because the stack will be increasing by d sequence of columns).

Value $d_1[i][j]$ each j is equal to a value lying in the moment the top of stack.

Clearly, as additions to the stack on each line i occurs exactly m pieces, then deletes and also could not be more so in the sum of the asymptotic behavior is linear.

Dynamics $d_2[i][j]$ to find indices k , considered to be similar, but it is necessary to view the columns right to left.

Also note that this algorithm consume $O(m)$ memory (not counting the input data - matrix $a[0..n][0..m]$).

Implementation

This implementation of the above algorithm reads the dimensions of the matrix, then the matrix itself (as a sequence of numbers separated by spaces or line breaks), and then outputs the answer - the size of the largest zero submatrix.

Easy to improve this implementation so that it also outputs a zero submatrix itself: it is necessary at each change ans remember also row and column of the submatrix (they will be respectively $d[j] + 1$, $i, d1[j] + 1, d2[j] - 1$).

```

int n, m;
cin >> N >> m;
vector<vector<int>> A (n, vector<int>(m));
for (int i = 0; i <n; ++ I)
    for (int j = 0; j <m; ++
        + J) cin >>
            A[i] [j];

int ans = 0;
vector<int> D (m, -1), D1 (m), D2 (m);
stack<int> St;
for (int i = 0; i <n; ++ I) {
    for (int j = 0; j <m; ++
        J) if (a[i] [j]
        == 1)
            d[j] = I;
    while (! St.empty())
        st.pop(); for (int j = 0; j
        <m; ++ J) {
            while (! St.empty() && D[st.top()] <= D[j]) st.pop();
            d1[j] = St.empty() ? -1 : St.top();
            st.push (j);
        }
    while (! St.empty())
        st.pop(); for (int j = m-1;
        j >= 0; - J) {
            while (! St.empty() && D[st.top()] <= D[j]) st.pop();
            d2[j] = St.empty() ? m: st.top();
            st.push (j);
        }
    for (int j = 0; j <m; ++ J)
        ans = max (ans, (i - d[j]) * (d2[j] - D1[j] - 1));
}

```

```
cout << Ans;
```

Gauss solution of linear equations

Given a system n linear algebraic equations (SLAE) with m unknown. Required to solve this system: to determine how many solutions it has (none, one, or infinitely many), and if it has at least one solution, then find any of them.

Formally problem is formulated as follows: solve the system:

where the coefficients ($i = 1 \dots n, j = 1 \dots m$) and $x_i (i = 1 \dots n)$ known as variables $x_i (i = 1 \dots m)$ the unknown unknowns.

Convenient matrix representation of this problem:

$$Ax = b,$$

where A - Matrix $n \times m$ Composed of the coefficients a_{ij} , x and b - The height of the column vectors m .

It should be noted that the linear algebraic equation may not be over the real numbers, and over a field **modulo** any number of p , ie:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1m}x_m = b_1, & (\text{mod } p) \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2m}x_m = b_2, & (\text{mod } p) \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nm}x_m = b_n & (\text{mod } p) \end{cases}$$

- Gauss algorithm works for such systems too (but this case will be discussed below in a separate section.)

Gauss

Strictly speaking, the method described below correctly called by the "Gauss-Jordan" (Gauss-Jordan elimination), because it is a variation of the Gauss method described by Jordan surveyor William in 1887 (it is worth noting that William Jordan is not the author of any of Jordan's theorem on curves or Jordan algebra

- All three different scientists namesake; moreover, appear to be more correct spelling is "Jordan", but writing "Jordan" is rooted in the Russian literature). It is also interesting to note

that simultaneously by Jordan (according to some sources even before him) came up with this algorithm Klasen (B.-I. Clasen).

The basic scheme

Briefly, the algorithm is **sequential elimination** each of variables of the equation as long as in each equation will remain only one variable. If $n = m$, Then we can say that the Gauss-Jordan algorithm seeks to bring the matrix A system to the identity matrix - because after the matrix has become single, obvious solution to the system - the solution is unique and is given by The coefficients b_i .

When this algorithm is based on two simple transformations equivalent system: first, the two equations may be interchanged, and secondly, any equation can be replaced by a linear combination of the rows (from zero coefficient), and the other rows (of arbitrary coefficients).

In a **first step** Gauss-Jordan algorithm divides the first row by a factor a_{11} . The algorithm then adds the first row to the other rows such coefficients to the coefficients of the first column has been accessed to zero - for this purpose, obviously, when added to the first row i Second it should be multiplied by $-a_{i1}$. At each operation with the matrix A (Divide by, the addition of one other line) and the corresponding operations are carried out with the vector; in a sense, it behaves as if it were $m+1$ Th column of the matrix A .

As a result, after the first step, the first column of the matrix A will be a single (ie, unit will contain the first row and zeros in the other).

Similarly, the second step of the algorithm is done, only now considered the second column and second row: first, the second line is divided into a_{22} And then subtracted from all other rows with such coefficients,

to nullify the second column of the matrix A .

And so on, until we process all rows or all columns of the matrix A . If $n = m$, Then by the construction of the algorithm is obvious that the matrix A turn the unit that we required.

Search the support element (pivoting)

Of course, the above scheme is incomplete. It works only if each i -Th step element

a_{ii} different from zero - otherwise, we simply can not achieve zero the remaining coefficients in the current column by adding to them i -Th row.

To make the algorithm employed in such cases, just there is a process **choice**

supporting member (In English it is called a word "Pivoting"). He is

that is made rows and / or columns of the matrix to the right element a_{ii} proved non-zero number.

Note that the row permutation is realized much easier on a computer than a permutation of the columns: for the exchange of some places two columns must remember that these two variables have exchanged places, and then, in the reduction of the response to correctly recover what the answer to which the variable refers . When changing strings, no such additional actions are necessary.

Fortunately for the correctness of the method alone is sufficient exchange lines (so-called "partial pivoting", unlike "full pivoting", and when the exchange lines and columns). But what exactly is the line should be selected for the exchange? And is it true that the search for the reference element should be done only when the current element a_{ii} zero?

General answer to this question does not exist. There are a variety of heuristics, but the most effective of them (the ratio of simplicity and efficiency) is that **heuristic**: As a reference element to be taking the highest modulo element, said supporting element search and it is necessary to exchange **always** And not only when necessary (i.e. not only when $a_{ii} = 0$).

In other words, before performing i -Phase II of the Gauss-Jordan elimination with partial pivoting heuristics to be found in i -Th column among the elements with indices from i to n maximum modulus, and swap line with this element with i -Th row.

First, this heuristic will solve SLAE, even if in the course of decision would happen so that the element $a_{ii} = 0$. Second, which is very important, this heuristic improves **numerical stability** Gauss-Jordan.

Without this heuristic, even if the system is such that each second phase - Gauss-Jordan will work, but eventually accumulating error may be so great that even for matrices of size about 20 error will exceed the answer itself.

Degenerate cases

So, if you stay on the Gauss-Jordan algorithm with partial pivoting, is approved if $m = n$ and nevrozhdna system (ie has a nonzero determinant, which means that it has a unique solution), then the above algorithm will work fully and come to the identity matrix A (Proof of this, ie, that a non-zero reference element will always be, is not presented here).

We now consider **general case** - When n and m are not necessarily equal. Assume that the support element on the i -Th step has not been found. This means that i -Th column of all rows from the current contain zeros. It is alleged that in this case the i -Th variable can not be determined, and a **independent variable** (Can take an arbitrary value). To the Gauss-Jordan continued his work for all subsequent variables in such a situation should just skip the current i -Th column, without increasing the current line number (we can say that we remove virtually i -Th column of the matrix).

So, some of the variables in the process of the algorithm can be delivered independent. It is understood that when the amount of m more than the number of variables n equations, then at least $m - n$ independent variables show up.

In general, if it finds at least one independent variable, it can take an arbitrary value, while the other (dependent) variables are expressed through it. This means that when we are working in the field of real numbers, the system is potentially **infinitely many solutions** (If we consider the linear algebraic equation modulo the number of solutions is equal to this module raising the number of independent variables). However, we must be careful: it is necessary to remember that even if the independent variables were found, nevertheless SLAE **may have no solutions at all**.

This happens when the remaining untreated equations (those to which the Gauss-Jordan is not reached, ie this equation, in which there were only independent variables) have at least one non-zero free term.

However, it is easier to check the explicit substitution of the solution found: all independent variables set to zero, the dependent variables to assign values found, and substitute this solution in current Slough.

Implementation

We present here the implementation of Gauss-Jordan heuristics with partial pivoting (choice of the support member as maximum of the column).

The input function `gauss()` system passed the matrix itself \mathbf{a} . The last column of the matrix \mathbf{a} - It is in our old notation column free coefficients (as done for the convenience of programming - as in the algorithm, all operations with free coefficients b repeat the operations with the matrix A).

The function returns the number of solutions of the system (0,1 or ∞) (Infinity is indicated in the special code constant `INF`, which can be any big difference). If at least one solution exists, then it returns to the vector `ans`.

```

int gauss (vector <vector <double>> A, vector <double> & Ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector <int> Where (m, -1);
    for (int col = 0, Row = 0; col <m && row <n; ++ Col) {
        int sel = row;
        for (int i = row; i <n; ++ I)
            if (abs (a[i] [col]) > abs (a[sel] [col])) sel = i;
        if (abs (a[sel] [col]) <EPS)
            continue;
        for (int i = col; i <= m; ++ I)
            swap (a[sel] [i], A[row] [i]);
        where[col] = Row;
        for (int i = 0; i <n; + + I) {
            if (i != row) {
                double c = a[i] [col] / A[row] [col];
                for (int j = col; j <= m; + + J)
                    a[i] [j] -= A[row] [j] * c;
            }
            + + Row;
        }
    }

    ans.assign (m, 0);
    for (int i = 0; i <m; + + I)
        if (where[i] == -1)
            ans[i] = A[where[i]] [m] / A[where[i]];
    for (int i = 0; i <n; + + I) {
        double sum = 0;
        for (int j = 0; j <m; + + J)
            sum += ans[j] * A[i] [j];
        if (abs (sum - a[i] [m]) > EPS)
            return 0;
    }

    for (int i = 0; i <m; + + I)
        if (where[i] == -1)
            return INF;
    return 1;
}

```

The functions supported by two pointers - the current column `col` and current row `row`.

Starts as a vector in which each variable is recorded in what line it should happen (in other words, for each column written line number in which this column is different from zero). This vector is needed because some variables could not "determine" in the solution (ie, it

independent variables, which can be given an arbitrary value - for example, here is the implementation of zeros).

The implementation uses the technique of partial pivoting, producing a search string with a maximum modulo element and then rearranging the row position `row` (Although obvious permutation of the rows can be replaced by the exchange of two indices from an array, in practice it does not give the real payoff, because spent on exchanges $O(n^2)$ operations).

In order to ease the implementation of the current line is not divisible by the support member - with the result that at the end of the algorithm becomes the unit matrix and the diagonal (however, apparently dividing line allows the drive element to reduce some errors occur).

After finding a solution to it is inserted back into the matrix - to check whether the system has at least one solution or not. If the test is successful solutions found, the function returns 1 or ∞ -

depending on whether there is at least one independent variable or not.

Asymptotics

We estimate the asymptotic behavior of this algorithm. The algorithm consists of m phases, each of which takes place:

- search and a rearrangement of the support member - during using heuristics "partial pivoting "(maximum search in the column)
- if the reference element in the current column was found - that the addition of this equation to all other equations - during $O(nm)$

Obviously, the first paragraph has asymptotics less than a second. Note also that the second paragraph is performed no more than $\min(n, m)$ times - as much as can be dependent variables in Slough.

Thus, **final asymptotics** algorithm takes the form $O(\min(n, m) \cdot nm)$. At $n = m$ this

estimate is transformed into $O(n^3)$.

Note that when the linear algebraic equation is not seen in the field of real numbers, and in the field modulo two, the system can be solved much faster - see below under "solving the linear modulo".

A more accurate estimate of the number of actions

To simplify the calculations, we assume that $n = m$.

As we already know, the time of the entire algorithm is actually determined by the time taken to exclude the current equation of the rest.

This may occur on each of n steps with the current equation is added to all $n - 1$ others. When adding the work is only the columns from the current. Thus, the sum obtained

$n^3/2$ operations.

Additions

Acceleration algorithm: its division into direct and reverse

To double its acceleration algorithm can consider another version of it, more classical, when the algorithm is partitioned into phases forward and reverse.

In general, unlike the above-described algorithm can not lead to a diagonal matrix form, and a **triangular form** - When all the elements strictly below the main diagonal are equal to zero.

System with a triangular matrix is trivially solved - first from the last equation is the value immediately to the last variable, then the obtained value is substituted into the equation, and is the penultimate penultimate value of the variable, and so on. This process is called **reverse progress** Gauss.

Forward stroke Gauss - this algorithm similar to the algorithm described above Gauss-Jordan, for one exception: the current variable is not excluded from all equations, but only after the current equations. As a result, it does not get the diagonal and triangular matrix.

The difference is that the direct flow of work **faster** Gauss-Jordan - because on average he makes in

half additions of one equation to another. Reverse works for $O(nm)$ That, in any asymptotically faster forward stroke.

Thus, if $n = m$, Then the algorithm will have to do $n^3/4$ operations - which is twice less than Gauss-Jordan.

Solving the linear modulo

For solving the linear module can be used by the algorithm described above, it retains its validity.

Of course, now it becomes unnecessary to use any artful technique of choice supporting element - is sufficient to find any nonzero element in the current column.

If the module is a simple, no complications do not arise - occurring in the course of Gauss division does not create any problems.

Especially remarkable **module equal to two** For him, all operations can be performed with the matrix is very efficient.

For example, a single row from subtraction other modulo two - this is actually their

symmetric difference ("xor"). Thus, the entire algorithm can be greatly accelerated by squeezing the entire matrix in bit masks and their terms only. We present here a new implementation of the main part of the Gauss-Jordan using standard container C ++ "bitset":

```
int gauss (vector <bitset <N>> a, int n, int m, bitset <N> & ans) {
    vector <int> Where (m, -1);
    for (int col = 0, Row = 0; col <m && row <n; + +
        Col) { for (int i = row; i <n; + + I)
        if (a[i] [col]) {
            swap (a[i],
                  A[row]);break;
        }
        if (! a[row]
            [col]) continue;
        where[col] = Row;
        for (int i = 0; i <n; + + I)
            if (i! = row && a[i]
                [col]) a[i] ^= A[row];
        + + Row;
    }
}
```

As you can see, the implementation has become even a little shorter, though it is much faster than the old implementation

- Namely, the faster 32 times due to the bit compression. It should also be noted that the solution of systems modulo two in practice is very fast, since cases when from one row should take another, occur infrequently (for sparse matrices, this algorithm can work during most of the order of the size of a square than a cube).

If the module **arbitrary** (Not necessarily simple), then everything becomes more complicated. It is clear that using [Chinese Remainder Theorem](#), We reduce the problem with an arbitrary module to the modules only type "prime power". [Text was further obscured because is anecdotal - perhaps the wrong way to solutions]

Finally, consider the question **number of solutions of linear algebraic equation modulo**. The answer is quite simple: the number of solutions is p^k Wherein p - Module k - The number of independent variables.

A little bit about the different methods of selecting the reference element

As mentioned above, the simple answer to this question is no.

Heuristics "partial pivoting", which was to find the maximum element in the current column, actually works quite well. It also turns out that it gives almost the same result as the "full pivoting" - when the support member is sought among the elements of the entire submatrix - starting with the current row and the current column.

But it is interesting to note that both of these heuristics with the search for the maximum element, in fact, are very dependent on how the original equations were scaled. For example, if one of the equations of the system multiplied by a million, then this equation is almost certain to be selected as a presenter at the first step. This seems rather strange, so logical transition to a little more complicated heuristics - so called "**Implicit pivoting**".

Heuristics implicit pivoting is that the elements of the various lines are compared as if both lines were normalized so that the maximum modulus element therein would be unity. To implement this technique, it is necessary to maintain a current maximum in each row (each line or support so that it was at a maximum is equal to unity modulus, but this can lead to an increase in the accumulated error).

Improving the response found

Because, despite the various heuristics, the Gauss-Jordan could still lead to large errors on special order sizes matrices even 50-100.

In this regard, the resulting algorithm Gauss-Jordan response can be improved by applying it to a simple numerical method - for example, a simple iteration.

Thus, the solution turns into a two-step: First, a Gauss-Jordan, then - a numerical method that takes as initial data the solution obtained in the first step.

This method allows several to extend the set of problems solved by Gauss-Jordan algorithm with an acceptable margin of error.

Literature

- William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. **Numerical Recipes: The Art of Scientific Computing** [2007]
- Anthony Ralston, Philip Rabinowitz. **A first course in numerical analysis** [2001]

Finding the rank of a matrix

Rank of the matrix - this is the largest number of linearly independent rows / columns of the matrix.

Rank is not defined only for square matrices; let the matrix is rectangular and has a size NxM.

Also rank matrix can be defined as the maximal order minors of the matrix are different from zero.

Note that if the matrix is square and its determinant is nonzero, then the rank is equal to N (= M), or it will be less. In the general case, the rank of the matrix is not greater than min (N, M).

Algorithm

Search rank can be modified using [Gauss method](#). Will perform exactly the same operations in the solution of the system or finding its determinant, but if at any step in the i-th column of this row to the unselected no nonzero, then we skip this step, and the rank is decremented (initially set equal rank max (N, M)). Otherwise, if we have found on the i-th step of the string with a nonzero element in the i-th column, then mark the line as selected, and perform common operations subtraction of this line from the rest.

Implementation

```
const double EPS = 1E-9;

int rank = max (n, m);
vector <char> line_used
(n); for (int i = 0; i <m;
+ + i) {
    int j;
    for (j = 0; j <n; + + j)
        if (! line_used [j] && abs (a [j] [i])>
            EPS) break;
    if (j == n) -
        rank;
    else {
        line_used [j] = true;
        for (int p = i +1; p <m; +
            + p) a [j] [p] / =
            a [j] [i];
        for (int k = 0; k <n; + + k)
            if (k! = j && abs (a [k] [i])>
                EPS) for (int p = i +1; p
                <m; + + p)
                    a [k] [p] -= a [j] [p] * a [k] [i];
    }
}
```

Calculating the determinant of a matrix by Gauss

Suppose given a square matrix A of size NxN. Required to compute its determinant.

Algorithm

Use the ideas [Gauss method for solving systems of linear equations](#).

Will perform the same steps as when solving a system of linear equations, excluding only the division of the current line to a [i] [i] (more precisely, the division itself can be carried out, but implying that the number shall be made for the sign of the determinant). Then all operations that we will produce a matrix will not change the value of the determinant of the matrix, except, perhaps, a sign (we only exchange the places of two lines that changes to the opposite sign, or we add one line to the other, that does not change the value of the determinant).

But the matrix to which we arrive after Gauss, is diagonal and its determinant is the product of the elements on the diagonal. Sign, as already mentioned, will be determined by the number of exchange lines (if odd, the sign of the determinant should be changed to the opposite).

Thus, we can use Gauss to calculate the determinant of a matrix in O (N3).

It remains only to note that if at some point we do not find in the current column nonzero, then the algorithm should stop and return 0.

Implementation

```
const double EPS = 1E-9;
int n;
vector <vector <double>> a (n, vector <double> (n));
N reading ... and a ...

double det = 1;
for (int i = 0; i <n; + +
    i) {int k = i;
    for (int j = i +1; j <n; + + j)
        if (abs (a [j] [i])> abs (a [k]
            [i])) k = j;
    if (abs (a [k] [i]) <EPS)
        {det = 0;
        break;
    }
    swap (a [i], a
    [k]); if (i! = k)
        det =-det;
    det *= a [i] [i];
    for (int j = i +1; j <n; +
        + j) a [i] [j] / =
        a [i] [i];
    for (int j = 0; j <n; + + j)
        if (j! = i && abs (a [j] [i])>
            EPS) for (int k = i +1; k
            <n; + + k)
                a [j] [k] - = a [i] [k] * a [j] [i];
}
cout << det;
```

Calculating the determinant method for Kraut

O (N³)

There will be considered a modification of the method of Kraut (Crout), enables one to calculate the determinant of a matrix in O (N³).

Actually algorithm finds Kraut decomposition of the matrix A as $A = L U$ Where L - the lower and U - upper triangular matrix. Without loss of generality, we can assume that all the diagonal elements of L are equal to 1. But knowing these matrices, it is easy to calculate the determinant of A: it is the product of all the elements on the main diagonal of the matrix U.

There is a theorem, according to which every invertible matrix has LU-decomposition, and uniquely, if and only if all its principal minors are nonzero. It will be recalled that we consider only such expansions in which the diagonal L consists only of units; otherwise the same, generally speaking, the decomposition is not unique.

Let A - matrix, N - its size. We find the elements of the matrices

L and U. The algorithm consists of the following steps:

1. Let $L_{ii} i = 1, 2, \dots, N$
2. For each $j = 1, 2, \dots, N$, issue:
 1. For $i = 1, 2, \dots, j$ we find the value U_{ij}
 $j: U_{ij} = A_{ij} - \sum L_{ik} U_{kj}$,
wherein the sum over all $k = 1, 2, \dots, i-1$.
 2. Next, for $i = j+1, j+2, \dots, N$ have: $L_{ij} = (A_{ij} - \sum L_{ik} U_{kj}) / U_{jj}$,
where the sum is taken over all $k = 1, 2, \dots, j-1$.

Implementation

Code Java (using fractional arithmetic long):

```
static BigInteger det (BigDecimal a [] [], int n)
{
    try {
        for (int i = 0; i <n; i++)
        {
            boolean nonzero = false;
            for (int j = 0; j <n; j++)
            {
                if (a [i] [j]. compareTo (new BigDecimal
(BigInteger.ZERO)) > 0)
                    nonzero = true;
                if (! nonzero)
                    return BigInteger.ZERO;
            }
            BigDecimal scaling [] = new BigDecimal [n];
            for (int i = 0; i <n; i++)
            {
                BigDecimal big = new BigDecimal (BigInteger.ZERO);
                for (int j = 0; j <n; j++)
                    if (a [i] [j]. abs (). compareTo
                        (big) > 0) big = a [i] [j]. abs
                        ();
                scaling [i] = (new BigDecimal (BigInteger.ONE)).
                    divide (big, 100, BigDecimal.ROUND_HALF_EVEN);
            }
            int sign = 1;
            for (int j = 0; j <n; j++)
                if (nonzero)
                    if (a [i] [j]. compareTo (new BigDecimal
(BigInteger.ZERO)) < 0)
                        sign = -sign;
                else
                    if (a [i] [j]. compareTo (new BigDecimal
(BigInteger.ZERO)) > 0)
                        sign = -sign;
            a [i] [i] = sign;
        }
        return BigInteger.ZERO;
    }
}
```

```

    {
        for (int i = 0; i <j; i++)
        {
            BigDecimal sum = a [i]
            [j]; for (int k = 0; k
            <i; k + +)
                sum = sum.subtract (a [i] [k]. multiply (a [k]
                [j])); a [i] [j] = sum;
        }

        BigDecimal big = new BigDecimal (BigInteger.ZERO);
        int imax = -1;
        for (int i = j; i <n; i + +)
        {
            BigDecimal sum = a [i]
            [j]; for (int k = 0; k
            <j; k + +)
                sum = sum.subtract (a [i] [k]. multiply (a [k]
                [j])); a [i] [j] = sum;
            BigDecimal cur = sum.abs ();
            cur = cur.multiply (scaling
            [i]); if (cur.compareTo (big) > =
            0)
            {
                big = cur;
                imax = i;
            }
        }

        if (j! = imax)
        {

            for (int k = 0; k <n; k + +)
            {
                BigDecimal t = a [j]
                [k]; a [j] [k] = a
                [imax] [k]; a [imax]
                [k] = t;
            }

            BigDecimal t = scaling
            [imax]; scaling [imax] =
            scaling [j]; scaling [j] = t;
            sign =-sign;
        }

        if (j! = n-1)
            for (int i = j +1; i <n; i + +)
                a [i] [j] = a [i] [j].
                divide
                (A [j] [j], 100
BigDecimal.ROUND_HALF_EVEN);

    }

    BigDecimal result = new BigDecimal (1);
    if (sign == -1)
        result = result.negative
    (); for (int i = 0; i <n; i + +)
        result = result.multiply (a [i] [i]);

    return result.divide
        (BigDecimal.valueOf (1), 0,
BigDecimal. ROUND_HALF_EVEN). ToBigInteger ();
}

catch (Exception e)
{

```

```
    return BigInteger.ZERO;  
}
```

Integration by Simpson's formula

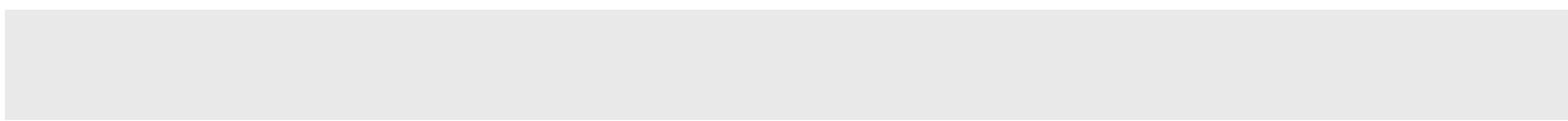
Required to calculate the value of the definite integral:

$$\int_a^b f(x)dx$$

The solution described here, was published in one of theses **Thomas Simpson** (Thomas Simpson) in 1743

Simpson's formula

Let n - A natural number. Divide the interval of integration $[a; b]$ on $2n$ Equal parts:



Now calculate the integral separately on each of the segments $[x_{2i-2}, x_{2i}], i = 1 \dots n$. And then add all the values.

So, let us consider the next segment $[x_{2i-2}, x_{2i}], i = 1 \dots n$. Replace the function $f(x)$ on

It parabola passing through the three points $(x_{2i-2}, x_{2i-1}, x_{2i})$. This parabola always exists and is unique.

It can be found analytically, then left only to integrate an expression for it, and we finally obtain

$$\int_{x_{2i-2}}^{x_{2i}} f(x)dx = (f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})) \frac{h}{3}.$$

Adding these values in all segments, we obtain the final **Simpson formula**:

$$\int_a^b f(x)dx = (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \dots + 4f(x_{2N-1}) + f(x_{2N})) \frac{h}{3}.$$

Error

The error is given by Eq Simpson, do not exceed the quantities:

$$\frac{1}{180} h^4 (b-a) \max_{a \leq x \leq b} |f^{(4)}(x)|.$$

Thus, the error is of the order of reduction as $O(n^4)$.

Implementation

Here $f(x)$ - Some user-defined function.

```
double a, b; // input
const int N = 1000*1000; // number of steps (Already multiplied
2)
double s = 0;
double h = (b - a) / N;
for (int i = 0; i <= N; ++
I) {
    double x = a + h * i;
    s += f(x) * ((i == 0 || I == N) ? 1 : ((i & 1) == 0) ? 2 : 4);
}
s *= h / 3;
```

Newton's method (tangent) to find the roots

This iterative method invented **Isaac Newton** (Isaak Newton) about 1664, however, this method is sometimes referred to by the Newton-Raphson (Raphson), as Raphson invented the same algorithm a few years later Newton, but his article was published much earlier. The task

is as follows. Given the equation:

$$f(x) = 0.$$

Required to solve this equation precisely find one of its roots (assuming the root exists). It is assumed that

$f(x)$ continuous and differentiable on the interval $[a; b]$

Algorithm

Input parameter of the algorithm, except for the function is also **the initial approximation** - Some x_0 , From which the algorithm starts to go.

Suppose that we have calculated x_i . Calculate x_{i+1} follows. Draw a tangent to the graph of the function in point $x = x_i$ And find the point of intersection of this tangent with the horizontal axis. x_{i+1} is set equal to the found point, and repeat the whole process from the beginning.

It is easy to obtain the following formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Intuitively, if the function is quite "good" (smooth) and x_i is sufficiently close to the root, then x_{i+1} will be even closer to the required root.

The rate of convergence is **quadratic**. That, relatively speaking, means that the number of accurate digits in the approximate value x_i doubles with each iteration.

Application to calculate the square root

Consider the example of Newton's method to compute the square root. If we substitute

$f(x) = \sqrt{x}$. Then after simplifying expressions obtain:

$$x_{i+1} = \frac{x_i + \frac{n}{x_i}}{2}.$$

The first exemplary embodiment of the problem - when given a fractional number n And the need to calculate its root with some accuracy **EPS**:

```
double n;
cin >> N;
const double EPS = 1E-15;
double x = 1;
for (;;) {
    double nx = (x + n / x) / 2; if
        (abs (x - nx) < EPS) break; x =
            nx;
}
printf ("% .15 Lf", x);
```

Another common problem - when you want to calculate the integral root (part n find the greatest x such that). Here we have a little change the stop condition of the algorithm, since it may happen that x starts "jumping" near response. Therefore, we add the condition that if the value x decreased in the previous step and the current step is trying to increase, the algorithm must be stopped.

```
int n;
cin >> N;
```

```

int x = 1;
bool decreased = false;
for (,;)
{
    int nx = (x + n / x) >> 1;
    if (x == nx || nx > x && decreased) break;
    decreased = nx <x;
    x = nx;
}
cout << x;

```

Finally, we have a third option - in the case of long arithmetic. Since the number of n can be quite large, it makes sense to pay attention to the initial approximation. Obviously, the closer to the root of it, the faster the results. Quite simple and efficient to take as

The number of initial approximations $\lfloor \frac{n}{2} \rfloor^2$, where bits - The number of bits including n . Here is the code in the language Java, demonstrates this option:

```

BigInteger n; // input
BigInteger a = BigInteger.ONE.shiftLeft (n.bitLength() / 2);
boolean p_dec = false;
for (,;)
{
    BigInteger b = n.divide(a).Add(a).
    ShiftRight(1);
    if (a.compareTo(b) == 0 || A.compareTo(b) < 0 && P_dec) break;
    p_dec = a.compareTo(b) > 0;
    a = b;
}

```

For example, this embodiment is performed for the code number 10^{1000} in 60 milliseconds, and if you remove the improved selection of the initial approximation (just start with 1) Will be executed approximately 120 milliseconds.

Ternary search

Statement of the Problem

Let there be given a function $f(x)$ unimodal on some interval. Under the unimodal mean one of the two options. First: first function is strictly increasing, then reaches a maximum (at one point or the whole interval), then strictly decreasing. The second option is symmetrical: the function decreases first decreases, reaches a minimum, increases. In the future, we will consider the first option, the second will be absolutely symmetrical him.

Required find the maximum function $f(x)$ on the interval $[l; r]$.

Algorithm

Take any two points m_1 and m_2 in this segment: $l < m_1 < m_2 < r$. Calculate the value of $f(m_1)$ the function and $f(m_2)$. Next we get three options:

- If you find that $f(m_1) < f(m_2)$ the required maximum can not be located on the left side, ie in parts $[l; m_1]$. This is easily seen if the left point function is smaller than the right, then either of these two points are in the "lift" function, or only the left point is there. In any case, it means that maximum further makes sense to look only in the interval $[m_1; r]$.
- If, conversely, $f(m_1) > f(m_2)$, The situation is similar to the previous up to the symmetry. Now desired maximum may not be in the right portion, i.e. in parts $[m_2; r]$. So go to the segment $[l; m_2]$.
- If the $f(m_1) = f(m_2)$ Then either of these two points are at the maximum, or a point located in the left increasing field, and right - in descending order (essentially used here is that the increase / strict decrease). Thus, in a further search must be performed in the interval $[m_1; m_2]$ But (To simplify code), this case can be attributed to any of the previous two.

Thus, according to the comparison result of the function at two internal points instead of the current segment we find a new search segment. Now repeat all the actions for this new segment, again obtain a new, strictly smaller segment, etc.

Sooner or later the length of the segment will be a little smaller than a predetermined constant-precision, and the process can be stopped. This method is numerical, so after stopping the algorithm can assume approximately that at all points $[l; r]$ the maximum is reached; as a response can take, for example, point l .

It remains to note that we did not impose any restrictions on the choice of points m_1 and m_2 . By this method, it is understood to depend on the speed of convergence (but error occurs). The most common way

- Choose points so that the segment $[l; r]$ shared them into 3 equal parts:

$$m_1 = l + \frac{r - l}{3}$$
$$m_2 = r - \frac{r - l}{3}$$

However, a different choice when m_1 and m_2 closer together, convergence rate will increase slightly.

Case of an integer argument

If the argument to an integer, then the segment also becomes discrete, however, as we
a
n
d
There are no restrictions on the choice of points an d , The correctness of the algorithm is not affected.
You can still choose an d so that they shared segment 3 parts, but equal
only approximately.

Wherein the second point - the stopping criterion of the algorithm. In this case it is necessary ternary search

will stop when it gets an d , Because in this case will be impossible to select points
so that there are distinct and different from and, and this can lead to infinite loops. After

$$r - l < 3$$

$$m_1 \quad m_2$$

Ternary search algorithm stops and becomes r , Several of the remaining points,
it is necessary to select a point with the maximum value of the
candidates $(l, l + 1, \dots, r)$ function.

Implementation

Implementation for the continuous case (ie, the function f has the form: double f (double x)

```
double l = ..., r = ..., EPS = ...; // input while
(r - l > EPS) {
    double m1 = l + (r - l) / 3,
    m2 = r - (r - l) / 3;
    if (f (m1) < F (m2))
        l = m1;
    else
        r = m2;
}
```

EPS

Here - in fact, **absolute error** Answer (not counting errors due to inaccurate calculation functions).

Instead of the criterion "while ($r - l > EPS$)" and you can choose a stopping criterion:

```
for (int it = 0; it < iterations; ++ It)
```

On the one hand, it is necessary to choose the constant $\frac{1}{3}$ To provide the desired precision (typically quite a few hundred to achieve maximum accuracy.) But, on the other hand, by iteration ceases to depend on the absolute values and we are actually using $\frac{1}{3^k}$ iterations Set the desired **relative error**.

Binomial coefficients

Binomial coefficient is the amount of ways to choose a set of k objects of n various items, excluding the order of these elements (ie the number of unordered collections).

Also binomial coefficients - it appears in the expansion $(a + b)^n$ (Eg, binomial theorem)

$$(a + b)^n = C_n^0 a^n + C_n^1 a^{n-1} b + C_n^2 a^{n-2} b^2 + \dots + C_n^k a^{n-k} b^k + \dots + C_n^n b^n$$

It is believed that this formula, like a triangle, to effectively find coefficients opened Blaise Pascal (Blaise Pascal), who lived in the 17th century. Nevertheless, he was known to the Chinese mathematician Yang Hui (Yang Hui), who lived in the 13th century. Perhaps it was opened by the Persian scholar Omar Khayyam (Omar Khayyam). Moreover, Indian mathematician Pingala (Pingala), still lived in the 3rd century BC, got similar results. Newton is a merit that he generalized this formula for degrees that are not natural.

Calculation

An analytic formula to compute:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

This formula can be deduced from the problem of the disordered sample (number of ways to randomly select items from n elements.)

First, calculate the number of ordered samples. Select the first element is

Ways second - $n - 1$ Third - $n - 2$ And so forth. As a result, the number of samples we get ranked formula. To disordered samples, easily go if we notice that each disordered sample corresponds exactly $k!$ ranked (since

is the number of possible permutations k elements.) As a result, by dividing $\frac{n!}{(n-k)!}$ on $k!$, We obtain the desired formula.

Recurrence formula (Which is associated with the famous "Pascal's triangle"):

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

It is easy to deduce by the previous formula.

It is worth noting particularly when $n < k$ value C_n^k always assumed to be zero.

Properties

Binomial coefficients have many different properties, present the most simple of them:

- Rule of symmetry:

$$C_n^k = C_n^{n-k}$$

- Adding-imposition:

$$C_n^k = \frac{n}{k} C_{n-1}^{k-1}$$

- Summation over k :

$$\sum_{k=0}^n C_n^k = 2^n$$

- Summation over n :

$$\sum_{m=0}^n C_m^k = C_{n+1}^{k+1}$$

- Summation over n and k :

$$\sum_{k=0}^m C_{n+k}^k = C_{n+m+1}^m$$

- Summation of squares:

$$(C_n^0)^2 + (C_n^1)^2 + \dots + (C_n^n)^2 = C_{2n}^n$$

- Weighted summation:

$$1C_n^1 + 2C_n^2 + \dots + nC_n^n = n2^{n-1}$$

- Communication with Fibonacci:

$$C_n^0 + C_{n-1}^1 + \dots + C_{n-k}^k + \dots + C_0^n = F_{n+1}$$

Calculations in the program

Direct calculations analytical formula

Calculating the first direct formula is very easy to program, but this method is subject to overflows, even at relatively low values n and (even if the answer is quite fit in any type of data, the calculation of intermediate factorials can cause an overflow). So very often, this method can only be used together with [[Long arithmetic | Long arithmetic]]:

```
int C (int n, int k) {
    int res = 1;
    for (int i = n-k+1; i <= n;
         ++i) res *= i;
    for (int i = 2; i <= k; +
         + i) res /= i;
}
```

An improved

It can be noted that the above implementation of the numerator and the denominator is the same number of factors (), each of which is less than unity. Therefore, we can replace our product at a fraction of fractions, each of which is real-valued. However, you will notice that after the multiplication of the current response to each another fraction will still receive an integer (this, for example, follows from the properties of the "make-issuance"). Thus, we obtain the following realization:

```
int C (int n, int k) {
    double res = 1;
    for (int i = 1; i <= k; ++i)
        res = res * (n-k+i) /
    I; return (int) (res + 0.01);
}
```

Here we present a fractional number carefully to the whole, given that due to accumulated errors it may be slightly less than the true value (eg. 2.99999 instead of three).

Pascal's Triangle

Using the same recurrence relation can build a table of binomial coefficients (in fact, Pascal's triangle), and take from it the result. The advantage of this method is that the intermediate results never exceed the answer and for the calculation of each new element of the table need only one addition. The disadvantage is the slow operation for large N and K, if in fact the table is not necessary, and need a single value (because you will need to calculate the build C_n^k table for all C_i^j , $1 < i < n$, $1 < j < n$, Or at least to $1 \leq j \leq \min(i, 2k)$).

```

const int maxn = ...;
int C[maxn +1] [maxn
+1];
for (int n =0; n <= maxn; + +
N) { C[n] [0] = C[n]
[n] = 1;for (int k =1;
k <n; + + K)
C[n] [k] = C[n-1] [k-1] + C[n-1] [k];
}

```

If the entire table of values is not necessary, it is easy to see enough of it to keep only two lines (the current - n -Th row and the previous - $n-1$ -Th.).

Calculation of O (1)

Finally, in some situations is beneficial predposchitat advance the values of all the factorials, in order to subsequently assume any necessary binomial coefficient, producing only two divisions. This may be advantageous when using [Long arithmetic](#)When memory does not allow the entire predposchitat

Pascal's triangle, or when it is required to make payments for some simple module (if it is not simple, then there are difficulties in dividing the numerator by the denominator and can be overcome if the factoring module and store all the numbers in the form of vectors of the degrees of these simple; sm [section "Long arithmetic in factored form"](#)).

Catalan numbers

Catalan numbers - numerical sequence, which is found in a surprising number of combinatorial problems.

This sequence is named after Belgian mathematician Catalan (Catalan), who lived in the 19th century, although in fact it was already known to Euler (Euler), who lived a century before Catalan.

Sequence

The first few Catalan numbers C_n (Starting from zero):

1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

Catalan numbers abound problems in combinatorics. n Th Catalan number - is:

- Bracketed number of valid sequences consisting of opening and closing parentheses.
- Number of binary trees with root $n+1$ leaves (vertices are numbered).
- Number of ways to completely separate brackets $n+1$ factor.
- Number of triangulations of a convex $n+2$ -Gon (ie, the number of partitions of a polygon nonintersecting diagonals into triangles).
- The number of ways to connect $2n$ points on the circumference n disjoint chords.
- Number of non-isomorphic complete binary trees with n internal vertices (ie having at least one son).
- Number of monotone paths from point $(0,0)$ point in a square lattice size $n \times n$, not rising above the main diagonal.
- The number of permutations of length n You can sort the stack (it can be shown that the permutation is stack sortable if and only if there are no indexes that). $i < j < k \quad a_k < a_i < a_j$
- Number of partitions of a set of continuous n elements (ie partitions into contiguous blocks).
- Number of ways to cover the ladder $1 \dots n$ via n rectangles (referring to the figure, consisting of n column i Th which has a height of i).

Calculation

There are two formulas for the Catalan numbers: recurrent and analytical. Since we believe that all the above problems are equivalent, to prove the formulas, we will choose the task with which to do it the easiest way.

Recurrence formula

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

Recurrence formula is easily derived from the problem of the correct bracket sequence.

The leftmost opening parenthesis l corresponds to a certain closing bracket r, that breaks the formula two parts, each of which in turn is correct bracketing sequence. Therefore, if we denote for any fixed r will be exactly ways. Summing this over all admissible k , We obtain the recurrence relation for C_n .

An analytic formula

$$C_n = \frac{1}{n+1} C_{2n}^n$$

(Here C_n^k designated as generally binomial coefficient).

This formula is the easiest way to bring out the problem of monotone paths. Total number of monotone paths in the lattice size $n \times n$ power. Now count the number of monotone paths crossing diagonal. Consider any of these ways, and find the first rib, which is above the diagonal. Reflect the diagonal all the way, going after this edge. The result is a monotonic path in the lattice. On the other hand, any monotonic path in the lattice $(n-1) \times (n+1)$ necessarily intersects the diagonal thus obtained as it is in this way from time

any (and only) monotone path intersecting diagonal lattice $n \times n$. Monotone paths lattice $(n-1) \times (n+1)$ there is C_{2n}^{n-1} . As a result, we obtain the formula:

$$C_n = C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$$

Necklaces

Task "necklace" - is one of the classical combinatorial problems. Required to count the number of different necklaces n beads, each of which may be colored in one color. When comparing two necklaces they can rotate, but not to turn (ie, allowed to make a cyclic shift).

Decision

You can solve this problem using [Lemma Burnside and Polya theorem](#). [Below is a copy of the text of this article] In this task, we can immediately find a group invariant permutations. Obviously, it will consist of n permutations:

$$n_{n-1} = n \cdot 1 \cdot \dots \cdot (n-1)$$

Find an explicit formula for calculating $C(\pi)$. First, note that the permutations have the form that - oh permutation on n Second position is $\pi(1)$ taken modulo n . If it is larger). If we consider the cyclic structure of the permutation, we see that the unit goes into n passes

And so on, until we arrive at the

in $1 + 2i \cdot 1 + 2i \cdot \dots \cdot 1 + 3i$ number of $1 + kn$: for the remaining elements

performed similar calculations. From this we can understand that all cycles have the same length, equal $\text{lcm}(i, n)/\text{gcd}(i, n)$ ("Gcd" - the greatest common divisor, "lcm" - the least common fold). Then the number of cycles in π second permutation is simply equal to $\text{gcd}(i, n)$.

Substituting these values in the Polya theorem, we obtain **decision**:

$$\text{Ans} = \frac{1}{n} \sum_{i=1}^n k^{\text{gcd}(i, n)}$$

You can leave a formula in this form, but you can minimize it even more. We proceed from the sum of all i to sum only divisors n . Indeed, in our sum will be many of the same terms: if i is not a divisor n , Then there is such a divisor after calculation.

Consequently, for each divisor of its $\text{gcd}(i, n)$ $d|n$

summand $k^{\text{gcd}(d, n)} = k^d$ will take into account several times, i.e. the amount may be represented in the form:

where

re C_d - The number of such numbers i that $\text{gcd}(i, n) = d$. Find an explicit expression for this quantity.

Any such number is: $i = dj$ where $\text{gcd}(j, n/d) = 1$ (Otherwise $\text{gcd}(i, n) > d$).

Remembering [Euler's function](#), We find that the number of such - is the value of the Euler function $\phi(n/d)$.

Thus, $C_d = \phi(n/d)$ And finally obtain **formula**:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

The alignment of elephants on a chessboard

Required to find the number of ways to arrange K elephants on board size NxN.

Algorithm

Will solve the problem using **Dynamic Programming**.

Let $D[i][j]$ - The number of ways to arrange j elephants on the diagonals to the i-th inclusive, and only those diagonals that the same color as the i-th diagonal. Then, $i = 1 \dots 2N-1$, $j = 0 \dots K$.

Diagonal enumerate as follows (example for boards 5x5):

Black:	White:
1	5 9
5	2 6 8
9	7
9	6 8 4
7	3
3	8
4	4

Ie odd numbers correspond to black diagonals, even - white; diagonal enumerate in order to increase the number of elements in them.

With this numbering, we can calculate each $D[i][j]$, based only on $D[i-2][j]$ (deuce deducted that we treat the diagonal of the same color).

So, let the dynamics of the current element - $D[i][j]$. We have two transitions. First - $D[i-2][j]$, i.e. put all j elephants previous diagonal. Second transition - if we put one elephant at the current diagonal, and the remaining $j-1$ elephants - previous; note that the number of ways to put an elephant on the current diagonal equal to the number of cells in it minus the $j-1$, because elephants standing in the previous diagonals will overlap part of the directions. So manner, we have:

$$D[i][j] = D[i-2][j] + D[i-2][j-1] (\text{cells}(i) - j + 1)$$

wherein $\text{cells}(i)$ - number of cells lying in the i-th diagonal. For example, cells can be calculated as follows:

```
int cells (int i)
{
    if (i & 1)
        return i / 4 * 2 + 1;
    else
        return (i - 1) / 4 * 2 + 2;
}
```

It remains to determine the dynamics of the base, there is no difficulty: $D[0][0] = 1$, $D[1][1] = 1$.

Finally, calculating the dynamics actually find **answer** to solve the problem. Iterate number $i = 0 \dots K$ elephants standing on the black diagonals (number of the last black diagonal - $2N-1$), respectively, K_i elephants put the white diagonal (diagonal number of the last white - $2N-2$), i.e. to add the response value $D[2N-1][i] * D[2N-2][K_i]$.

Implementation

```
int n, k; // Input if (k>
2 * n-1) {
    cout << 0;
    return 0;
}

vector<vector<int>> d (n * 2, vector<int> (k
+2)); for (int i = 0; i <n * 2; + + i)
    d [i] [0] =
1; d [1] [1] = 1;
for (int i = 2; i <n * 2; + + i)
    for (int j = 1; j <= k; + + j)
        d [i] [j] = d [i-2] [j] + d [i-2] [j-1] * (cells (i) - j + 1);

int ans = 0;
```

```
for (int i = 0; i <= k; ++ i)
    ans += d [n * 2-1] [i] * d [n * 2-
2] [k-i]; cout << ans;
```

Right parenthesis sequence

Correct bracketing sequence is a string consisting only of characters "brackets" (often considered only parentheses, but here will be considered and the general case of several types of brackets), where each closing parenthesis there corresponding opening (with the same type).

Here we consider the classic problem parenthesis on the right of the sequence (hereinafter, for brevity, simply "sequence") for checking the accuracy, the number of sequences, all sequences generated by finding the lexicographically next sequence^{*} finding - the second sequence in a sorted list of all sequences, and, conversely, determination sequence numbers. Each of the problems addressed in two cases - when parenthesis are allowed only one type, and when several types.

Check on the correctness

Suppose first parenthesis are allowed only one type, then check the correctness of the sequence can be very simple algorithm. Let depth - This is the current number of open parentheses. Initially $\text{depth} = 0$. Will move from left to right on the line if the current parenthesis, then increase depth unit, otherwise reduce. If this once turned negative, or at the end of the algorithm depth different from zero, this string is not correct bracketing sequence is different. If several types of braces are allowed, then the algorithm to change. Instead, you must create counter stack depth which will put the opening parenthesis upon receipt. If the current character string - opening bracket, then put it on the stack, and if closing - then check that the stack is not empty, and that its top bracket is the same type as the current one, and then we obtain this bracket from the stack. If any of the conditions are not met, or end of the stack algorithm was not empty, then the sequence is not correct bracket, is otherwise.

Thus, both of these tasks, we learned to solve during $O(n)$.

The number of sequences

Formula

The number of correct sequences bracketed with one type of brackets can be calculated as [Catalan number](#). Ie if there n pairs of brackets (line length $2n$), The amount is equal to:

$$\frac{1}{n+1} \cdot C_{2n}^n.$$

Suppose now that there is not one, and types of brackets. Then each pair of brackets independently of the others can take one of k types, and therefore we obtain the following formula:

$$\frac{1}{n+1} \cdot C_{2n}^n \cdot k^n.$$

Dynamic programming

On the other hand, this problem can be approached from the standpoint of **Dynamic Programming**. Let - number of correct sequences of bracketed n steam parentheses. Note that in the first position will always stand opening bracket. It is clear that within this set of parentheses is worth some proper bracket sequence; similarly, after the pair of brackets is also worth correct bracket sequence. Now to count, move, how many pairs of parentheses $d[n]$ will stand inside the first pair, then, accordingly, $n - 1 - i$ pair of brackets will stand after the first pair. Therefore, the formula for $d[n]$ has the form:

$$d[n] = \sum_{i=0}^{n-1} d[i] \cdot d[n - 1 - i].$$

The initial value for this recursion formula - it $d[0] = 1$.

Finding all sequences

Sometimes you want to find and put all the right parenthesis sequence specified length n (In this case n - The length of the line).

To do this, you can start with the lexicographically first sequence and then finding each time lexicographically following sequence using the algorithm described in the next section.

But if the restrictions are not very big (n to $10 - 12$), Then we can proceed much easier. Find all the possible permutations of these brackets (for this useful function `next_permutation()`), they will and every check on the correctness of the above algorithm, and if correct, shows the current sequence.

$$C_{2n}^n$$

Also, the process of finding all sequences can be issued in the form of a recursive enumeration with clipping (which ideally can be brought up to speed on the first algorithm).

Finding the following sequence

Remaining learn to look for that same position of the first changes. For this we follow the line from right to left and maintain a balance depth open and closed brackets (opening brace at the meeting will reduce depth And when the cover - increase). If at any time we are on the opening parenthesis, and the balance after the treatment of the symbol is greater than zero, then we have found the right-most position from which we can begin to change the sequence (in fact, means that the left has not yet closed bracket). Deliver to the current position closing parenthesis, and then the maximum possible number of opening parenthesis, and then all remaining closing brackets - the answer is found.

If we viewed the entire row and have not found a suitable position, the current sequence - the maximum, and there is no answer.

Implementation of the algorithm:

```
string s;
cin >> S;
int n = (int) S.length();
string ans = "No solution";
for (int i = n-1, Depth = 0; i >= 0; -
    I) { if (s[i] == '(')
        - Depth;
    else
        + + Depth;
    if (s[i] == ')' && Depth > 0) {
        - Depth;
        int open = (n-i-1 - Depth) / 2;
        int close = n-i-1 - Open;
        ans = s.substr(0, I) + ')' + String ('(', Open) + String
        (')', Close);
        break;
    }
}
cout << Ans;
```

Thus, we have solved this problem for $O(n)$.

Sequence Number

Here let n - The number of pairs of parentheses in a sequence. Required for a given proper bracketing sequence to find her number in the correct bracket lexicographically ordered sequences.

Learn to consider supporting **dynamics** Wherein i - The length of the bracket sequence
(She "semiregular": any closing bracket features a steam room opens, but not closed all open parentheses)

j - The balance (ie the difference between the number of opening and closing parentheses) the number of such sequences. When calculating the dynamics of this, we believe that the brackets are the same type.

Consider this trend as follows. Let $d[i][j]$ - A value that we want to calculate. If

$i = 0$, The answer is clear immediately: 1. All other $d[0][j] = 0$ Suppose now that Then mentally iterate over what was equal to the last character of the sequence. If it is equal to '(', until this symbol we are able to. It was equal to if ')', the previous was $i, j - 1$ state $(i - 1, j + 1)$. Thus, we obtain:

(Assuming that all values $d[i][j]$ zero if negative). Thus, this dynamic we can count for $O(n^2)$.

We now turn to the solution of the problem.

Please let permissible only brackets **one** type. Head count depth nesting in parentheses, and will move in sequence from left to right. If the current character $s[i] (i = 0 \dots 2n - 1)$ is "(", then we increase depth by 1 and go to the next character. If the current symbol equals ')', we must add to the answer $d[2n - i - 1][\text{depth} + 1]$. Thereby considering that in this position could be preceded by '(' (which would lead to a sequence of lexicographically smaller than the current one), then we reduce depth unit.

Now let parenthesis are allowed **more** types. Then when considering the current character $s[i]$ to Conversion depth we need to go through all the brackets that are less current character, try to put this bracket at the current position (thereby obtaining a new balance $\text{depth} = \text{depth} \pm 1$) And and add to the answer amount corresponding to the "tails" - completions (which have a length $2n - i - 1$ Balance ndepth and types of brackets). It is alleged that the formula for this number is:

This formula is derived from the following considerations. First, we "forget" about the fact that there are several brackets types, and just take the response $d[2n - i - 1][\text{ndepth}]$. Now calculate how to change the response due to the of k bracket types. We have $2n - i - 1$ uncertain positions, of which ndepth are brackets, closing some of the previously discovered - hence, the type of brackets we can not vary. But all the other brackets (there are pairs $(2n - i - 1 - \text{ndepth})/2$) the answer is multiplied by the power of this k .

Location k Second sequence

Here let n - The number of pairs of parentheses in a sequence. In this problem, for a given required to find - th correct bracket sequence list lexicographically ordered sequences.

As in the previous section, we can calculate **dynamics** - The number of correct bracketed sequences of length i to balance j .

Suppose first that allowed only brackets **one** type.

We move on the characters of the string, with 0Th to $2n - 1$ Trader. As in the previous problem, we will store counter depth current depth of nesting in parentheses. In each current position will iterate possible symbol - opening or closing parenthesis. Suppose we want to put here an opening bracket

then we need to look at the value $d[i + 1][\text{depth} + 1]$. If it $\geq k$ current position opening bracket increase depth by one and move to the next character. Otherwise, we Deduct from the value $d[i + 1][\text{depth} + 1]$. Put a closing bracket and reduce the value of depth . In Eventually, we obtain the required sequence of the bracket. Implementation in

Java using long arithmetic:

```
int n; BigInteger k; // input
```

```
BigInteger d[] [] = new BigInteger [n *2+1] [n  
+1];for (int i =0; i <= n *2; + + I)
```

```

        for (int j = 0; j <= n; ++ j)
            d[i] [j] =
BigInteger.ZERO; d[0] [0] =
BigInteger.ONE;
for (int i = 0; i <n *2; ++ I)
    for (int j = 0; j <= n; ++ J)
        { if (j +1 <=
N)
            d[i +1] [j +1] = D[i +1] [j +1]. Add(
d[i] [j]);if (j> 0)
            d[i +1] [j-1] = D[i +1] [j-1]. Add( d[i] [j]);
        }
}

String ans = new String();
if (k.compareTo( d[n *2] [0]) >
    0) ans = "No solution";
else {
    int depth = 0;
    for (int i = n *2-1; i>= 0; - I)
        if (depth +1 <= N && d[i] [depth +1]. CompareTo( k ) > =
            0) { ans += '(';
            + + Depth;
        }
        else {
            ans += ')';
            if (depth +1 <= N)
                k = k.subtract( d[i] [depth +1]);
            - Depth;
        }
}
}

```

Suppose now allowed not one, but **types** parentheses. Then the solution algorithm is different from previous only case that we should value is multiplied $D[i+1][n\text{depth}]$ by the amount $k^{(2n-i-1-n\text{depth})}$. To take into account that this residue could be any type of braces and a pair of brackets This balance is only $2n - i - 1 - n\text{depth}$ As $n\text{depth}$ brackets are closing for opening brackets outside of this residue (and their types because we can not vary). Implementation

of the Java language for the case of two types of brackets - round and square:

```

int n; BigInteger k; / / input

BigInteger d[] [] = new BigInteger [n *2+1] [n
+1];for (int i = 0; i <= n *2; ++ I)
    for (int j = 0; j <= n; ++ J)
        d[i] [j] =
BigInteger.ZERO; d[0] [0] =
BigInteger.ONE;
for (int i = 0; i <n *2; ++ I)
    for (int j = 0; j <= n; ++ J)
        { if (j +1 <=
N)
            d[i +1] [j +1] = D[i +1] [j +1]. Add(
d[i] [j]);if (j> 0)
            d[i +1] [j-1] = D[i +1] [j-1]. Add( d[i] [j]);
        }
}

String ans = new
String();int depth = 0;
char [] stack = new char[n
*2]; int stacksz = 0;
for (int i = n *2-1; i>= 0; -
I) { BigInteger cur;
/ / '('
if (depth +1 <= N)
    cur = d[i] [depth +1]. ShiftLeft((i-depth-1)/2 );
else
    cur = BigInteger.ZERO;
if (cur.compareTo( k ) > = 0)
{

```

```
ans += '(';
stack[stacksz++] = '('; ++Depth;
continue;
}
k = k.subtract( cur );
```

```

    // ')'
if (stacksz> 0 && Stack[stacksz-1] == '(' && Depth-1 >= 0)
    cur = d[i] [depth-1]. ShiftLeft((i-depth +1)/2 );
else
    cur = BigInteger.ZERO;
if (cur.compareTo( k ) >= 0)
{
    ans +=
    ')'; -
    Stacksz; -
    Depth;
    continue;
}
k = k.subtract( cur
); / /
if (depth +1 <= N)
    cur = d[i] [depth +1]. ShiftLeft((i-depth-1)/2 );
else
    cur = BigInteger.ZERO;
if (cur.compareTo( k ) >= 0)
{
    ans += '[';
    stack[stacksz + +] =
    '['; + + Depth;
    continue;
}
k = k.subtract( cur
); / /
ans +=
']'; -
Stacksz; -
Depth;
}

```

Number of labeled graphs

Given the number N vertices. Count the number of required G_N labeled graphs with different N vertices (ie vertices of the graph is labeled with different numbers from 1 to N). And graphs are compared in view of this painting vertices). Undirected edges, loops and multiple edges are forbidden.

Consider the set of all possible edges. For every edge (i, j) Suppose that $i < j$ (Based for an undirected graph and the absence of loops). Then the set of all possible edges of the graph has a capacity $C_{N!}^2 \frac{N(N-1)}{2}$.

Because any labeled graph uniquely defined by its edges, then the number of labeled graphs with N vertices is equal to:

$$G_N = 2^{\frac{N(N-1)}{2}}$$

The number of connected labeled graphs

Compared with the previous task, we impose an additional restriction that the graph must be connected. Denote the number required by $Conn_N$.

Learn to the contrary, count the number of **disconnected** graphs; then the number of connected graphs obtained as minus the number found. Moreover, learn to count the G_N number of **Root** (ie, with a distinguished vertex

- Root) **disconnected graphs**; then the number of disconnected graphs will be obtained from it by dividing by

N . Note that since the graph is disconnected, then there exists a connected component, which lies inside the square, and the rest of the graph will be a few more (at least one) connected components.

Iterate over the number

of K vertices in the connected component containing the root (obviously $K = 1 \dots N - 1$)

and find the number of such graphs. First, we must choose K of vertices, ie the answer is multiplied by C_N^K .

Secondly, the connected component with the root gives a factor $Conn_K$. Third, the remaining graph of $N - K$ vertices is an arbitrary graph, but because he gives the factor G_{N-K} . Finally, the number of ways allocate the root in a connected component of K vertices is K . Overall, at a fixed K number

Root disconnected graphs of power:

$$K C_N^K Conn_K G_{N-K}$$

Hence, the number of **disconnected** graphs with N vertices is equal to:

$$\frac{1}{N} \sum_{K=1}^{N-1} K C_N^K Conn_K G_{N-K}$$

Finally, the required number of **connected** graphs of power:

$$Conn_N = G_N - \frac{1}{N} \sum_{K=1}^{N-1} K C_N^K Conn_K G_{N-K}$$

Number of labeled graphs with K connected components

Based on the above formula, learn to count the number of labeled graphs with N and peaks K connected components.

$$D[N][K]$$

You can do this by using dynamic programming. Learn to count - the number of labeled graphs with N and peaks K connected components.

Learn how to calculate the next element $D[N][K]$. Knowing the previous values. We use the standard technique for solving such problems: take the vertex with the number 1, it belongs to some component, that this component and we will sort out. Iterate over the size of the components, then the number of ways to choose S

a set of vertices is C_{N-1}^{S-1} (One vertex - the top one - iterate is not necessary). The amount of ways to build a connected component of the vertices, we already know how to count - it . After removal this component of the graph, we are left with a graph and peaks connected components, ie

we got a recursive relationship on which you can calculate values :

Subtotal get code like this:

```
int d[n +1] [k +1]; // initially filled with  
zeros d[0] [0] = 1;  
for (int i =1; i <= n; ++ I)  
    for (int j =1; j <= i && j <= k;  
        ++ J) for (int s =1; s  
        <= i; ++ S)  
            d[i] [j] += C[i-1] [s-1] * Conn[s] * D[i-s] [j-1];  
cout << D[n] [k] [n];
```

Of course, in practice, will likely be needed [long arithmetic](#).

Generating combinations of N elements

Combinations of N elements on K lexicographically

Statement of the problem. Given positive integers N and K. Consider the set of numbers from 1 to N. required to withdraw all its various subsets of cardinality K, and in lexicographical order.

The algorithm is quite simple. The first combination is likely to be a combination of (1,2, ..., K). Learn how to find the current combination of the following lexicographically. To do this, find the current combination, the rightmost element not yet reached its maximum value; then increase it by one and all subsequent elements assign the lowest values.

```
bool next_combination (vector <int> & a, int n)
    {int k = (int) a.size ();
     for (int i = k-1; i >= 0; -
          i) if (a [i] < n-k + i
                  +1) {
                     ++ a [i];
                     for (int j = i +1; j < k;
                          ++ j) a [j] = a
                                  [j-1] +1;
                     return true;
                 }
     return false;
}
```

From a performance standpoint, this algorithm is linear (on average) if K is not close to N (ie, if not satisfied that $K = N - o(N)$). It suffices to prove that the comparison " $a[i] < n-k + i + 1$ " performed in the amount of $C_n^{n+1} k$ times, ie in $(N + 1) / (N-K + 1)$ times more than there are combinations of all N elements in K.

Combinations of N elements on K with changes of exactly one element

Required to write all combinations of N elements on K, but in such a manner that any two adjacent combinations will differ by exactly one element.

Intuitively, one can immediately see that this problem is similar to the problem of generating all subsets of a given set in such a manner, when two adjacent subsets differ by exactly one element. This problem is solved by directly [Gray code](#). If we each subset we associate

bitmask is generating Gray codes using these bit masks, and we get a response.

It may seem surprising, but the task of generating combinations also directly solved using

[Gray code](#). Namely generate Gray codes for the numbers 0 to $2N-1$ and leave only those codes which contain exactly K units. Surprising is the fact that in any sequence derived two adjacent mask (as well as the first and last mask) will differ in exactly two bits, that we just needed.

We prove this.

To prove recall the fact that the sequence G (N) Gray codes can be obtained as follows:

$$G(N) = OG(N-1) \cup 1G(N-1) R$$

ie take a sequence of Gray codes for the $N-1$, is appended to the beginning of each mask 0, add to the answer; then take a sequence of Gray codes for the $N-1$, invert it appends to the beginning of each mask and add 1 to the answer.

Now we can produce the proof.

First we prove that the first and last mask will differ in exactly two bits. It is sufficient to note that the first mask will look NK zeros and K units, and the last mask will look like: a unit then $NK-1$ zeros, then $K-1$ unit. It is easy to prove by induction on N, using the above formula

Gray codes for the sequence.

Now we prove that any two adjacent code will differ in exactly two bits. To do this, we turn again to the formula for the sequence of Gray codes. Suppose that within each of the halves (derived from $G(N-1)$) statement is true, we prove that it is true for the entire sequence. It suffices to prove that it is true in "gluing" two halves of $G(N-1)$, and it is easy to show, based on the fact that we know the first and last elements of these halves.

We now present a naive implementation that works for $2N$:

```
int gray_code (int n)
    {return n ^ (n >>
1);
}

int count_bits (int n)
    {int res = 0;
    for (; n; n >> =
1)
        res += n &
1; return res;
}

void all_combinations (int n, int k)
    {for (int i = 0; i <(1 << n);
++ i) {
        int cur = gray_code (i);
        if (count_bits (cur) == k) {for
            (int j = 0; j <n; ++
j)
                if (cur & (1 << j))
                    printf ("% d", j +1);
            puts ("");
        }
    }
}
```

It should be noted that it is possible and in some ways a more efficient implementation, which will build all kinds of combinations on the go, and thus work for $O(Cnk n)$. On the other hand, the implementation is a recursive function, and so for small n , she probably has a large hidden constant than the previous one.

Proper implementation itself - is a direct follow formula:

$$G(N, K) = 0G(N-1, K) \cup 1G(N-1, K-1) R$$

This formula can be easily obtained from the above formula for the sequence of Gray - we just choose a subsequence of elements suitable for us.

```
bool ans [MAXN];

void gen (int n, int k, int l, int r, bool rev)
    {if (k> n | | k <0) return;
    if (! n) {
        for (int i = 0; i <n; ++ i)
            printf ("% d", (int) ans
[i]); puts ("");
        return;
    }
    ans [rev? r: l] = false;
    gen (n-1, k,! rev? l +1: l,! rev? r: r-1,
rev); ans [rev? r: l] = true;
    gen (n-1, k-1,! rev? l +1: l,! rev? r: r-1,! rev);
}

void all_combinations (int n, int k)
    {gen (n, k, 0, n-1, false);
}
```

Burnside lemma. Polya theorem

Burnside lemma

This lemma was formulated and proved **Burnside** (Burnside) in 1897, but it was found that this formula has been previously opened **Frobenius** (Frobenius) in 1887, and even earlier - **Cauchy** (Cauchy) in 1845, therefore this formula is sometimes called Burnside's lemma, and sometimes - the Cauchy-Frobenius theorem.

Burnside lemma allows to count the number of equivalence classes in a set based on some of its internal symmetry.

Objects and representations

We draw a clear distinction between the number of objects and the number of views.

One and the same objects can match different views, but, of course, any representation corresponds to exactly one object. Consequently, the set of all representations is partitioned into equivalence classes. Our task - to count exactly the number of objects, or that the same number of equivalence classes.

Example problem: coloring of binary trees

Suppose we consider the following problem. Required to count the number of ways to paint binary trees with root n vertices in 2 colors, if each vertex we do not distinguish between right and left son.

Many objects here - it's a lot of different colorings in this understanding of trees.

We now define a set of representations. Each coloring we associate its defining feature where $f(v)$ $v = 1 \dots n$. Then the set of ideas - it's a lot of different functions of this form, and its size is obviously equal to 2^n . At the same time, this set of representations we have introduced a partition into equivalence classes.

For example, suppose $n = 3$ and the tree is as follows: root - the top one, and the vertices 2 and 3 - her sons. Then the following functions f_1 and f_2 to be equivalent:

$$\begin{aligned} f_1(1) &= 0 & f_2(1) &= 0 \\ f_1(2) &= 1 & f_2(2) &= 0 \\ f_1(3) &= 0 & f_2(3) &= 1 \end{aligned}$$

Permutation invariant

Why are these two functions and belong to the same equivalence class? Intuitively, this is clear - because we can swap the sons of vertex 1, ie peaks 2 and 3, after such a conversion function and coincide. But formally, this means that there exists permutation invariant π (i.e., that according to the problem does not change the object itself, but only its representation), such that:

$$f_2 \pi \equiv f_1$$

So, based on the conditions of the problem, we can find all permutation invariant, ie applying not that we do not go from one equivalence class to another. Then, to check whether the two functions

f_2 equivalent (ie, whether they are actually the same object), it is necessary for each permutation invariant π check to see whether the conditions: $f_2 \pi \equiv f_1$ (Or, equivalently, $f_1 \pi \equiv f_2$). If at least one permutation found this equation, then the equivalent, otherwise they not equivalent.

Finding all these permutations invariant with respect to which our problem is invariant - a key step for the application of the lemma as Burnside and Polya theorem. It is clear that these invariant permutations depend on the specific problem, and finding them - a process purely heuristic based on intuitive considerations. However, in most cases it is sufficient to manually find a few "core" of permutations, from which all others can be obtained by permutation of all possible products (and this, only the mechanical part of the work can be shifted to the computer, more detail is discussed below for an example of the problem).

It is easy to understand that the invariant permutations form **group** - As the product of any permutation invariant is also invariant permutation. Denote **group**

permutation invariant through G .

The formulation of Lemma

To formulate the left to remind one concept of algebra. **Fixed point** for relocation π is an element which is invariant under this permutation: For example, in our example, the fixed points will be the functions that correspond to colorings, not changing when they apply the permutation π (Not changing it in the formal sense of the equality of two functions). We denote $I(\pi)$ number of fixed points for relocation π .

Then **Burnside lemma** reads as follows: the number of classes is equal to the sum of the amounts of fixed points over all permutations of the group G divided by the size of the group:

$$\text{ClassesCount} = \frac{1}{|G|} \sum_{\pi \in G} I(\pi)$$

Although Burnside lemma itself is not so convenient to use in practice (it's unclear how quickly to seek value), it is most clearly reveals the mathematical essence, which is based on the idea of counting equivalence classes.

PROOF Burnside

Described here is the proof of Lemma Burnside is not so important for its understanding and application in practice, so it can be omitted on first reading.

The proof given here is the easiest of the famous and not using group theory. This proof was published Bogart (Bogart) and Kenneth (Kenneth) in 1991

So we need to prove the following statement:

$$\text{ClassesCount}|G| = \sum_{\pi \in G} I(\pi)$$

The quantity on the right - it is nothing like the amount of "invariant pairs", ie pairs such that $f\pi \equiv f$. Obviously, in the formula, we have the right to change the order of summation - do outside sum the elements f , and inside put value $J(f)$ - The number of permutations, for which f is invariant:

$$\text{ClassesCount}|G| = \sum_f J(f)$$

To prove this formula up a table whose columns will be signed by all the values, row - All the permutations, and in the cells of the table will stand works. Then if we consider the columns of the table as a set, some of them may coincide, and this will mean as that corresponding to each column are also equivalent. Thus, as many different set of columns is unknown quantities ClassesCount . Incidentally, from the point of view of group theory column of the table, signed by some element - is the orbit of this element; for the equivalent elements, obviously the same orbit, and by giving it different orbits ClassesCount .

So, the table columns themselves fall into equivalence classes; Now fix any class and consider the columns in it. First, we note that in these columns can stand only elements of one equivalence class (otherwise it would turn out that some equivalent transformation we transferred to another equivalence class, which is impossible). Secondly, each item will meet the same number of f_i once all the columns (also from the fact that the columns correspond to equivalent elements). It can be concluded that all the columns within the same equivalence class coincide with each other as a multiset.

Now fix an arbitrary element. On the one hand, it is found in a column exactly once (by definition). On the other hand, all the columns within the same equivalence class are identical as a multiset. Hence, within each column of any part of the equivalence class element g occurs exactly $J(g)$ times.

Thus, if we take an arbitrary manner from each equivalence class by one column and sum the number of elements in them, we find on the one hand, $\text{ClassesCount}|G|$ (It obtained simply by multiplying the number of columns on their size), and on the other hand, the sum of the quantities f (This follows from all the previous arguments):

QED.

Polya theorem. The simplest version of

Theorem **Polya** (Polya) is a generalization of Lemma Burnside, besides providing a convenient tool for finding number of equivalence classes. It should be noted that even before the Polya theorem was discovered and proved Redfield (Redfield) in 1927, but its publication was passed unnoticed by mathematicians of the time. Polya independently came to the same result only in 1937, and its publication has been more successful.

Here we look at the formula obtained as a special case of Polya theorem, and which is very useful for computing practice.

Overall Polya theorem in this article will not be considered.

We denote the number of cycles in the permutation π . Then we have the following formula
(special case of Polya)

where k - the number of values that can be taken each element representation
our problem-example (coloring the root of a binary tree in 2 colors) $f(v)$. For example, in
 $k = 2$.

Proof

This formula is a direct consequence of Lemma Burnside. To get it, we just need to find an explicit expression for the value in Lemma (recall that the number of fixed points permutation π).

So, consider some permutation π some element. Under the action of the permutation π moving the elements, as is known, for f cycles permutation. Note that, since the result to be obtained

$f \equiv f\pi$ Then within each cycle permutation must be identical elements. At the same time, to different cycles is no connection between the values of the elements does not occur. Thus, for each cycle permutation π we choose one value (of options), and thus we obtain a representation of all

f Invariant under this permutation, ie:

$$I(\pi) = k^{C(\pi)}$$

where $C(\pi)$ - The number of cycles of the permutation.

Example problem: Necklaces

Task "necklace" - is one of the classical combinatorial problems. Required to count the number of different necklaces n beads, each of which may be colored in one color. When comparing two necklaces they can rotate, but not to turn (ie, allowed to make a cyclic shift).

In this task, we can immediately find a group invariant permutations. Obviously, it will consist of n permutations:

$$n_{n-1} = n + \dots + (n-1)$$

Find an explicit formula for calculating n_{n-1} . First, note that the permutations have the form that - oh permutation on n Second position is $C(\pi)$ taken modulo i If it is larger). If we i consider the cyclic structure of the permutation, we see that the unit goes into n passes

And so on, until we arrive at the $1+i$ $1+i$ for the remaining elements

in $1+2i$ $1+2i$ - $1+3i$ number of $1+kn$ for the remaining elements performed similar allegations. From this we can understand that all cycles have the same length, equal $\text{lcm}(i, n)/\text{gcd}(i, n)$ ("Gcd" - the greatest common divisor, "lcm" - the least common fold). Then the number of cycles in i Second permutation is simply equal to $\text{gcd}(i, n)$.

Substituting these values in the Polya theorem, we obtain **decision**:

$$\text{Ans} = \frac{1}{n} \sum_{i=1}^n k^{\text{gcd}(i, n)}$$

You can leave a formula in this form, but you can minimize it even more. We proceed from the sum of all i to sum only divisors n .

Indeed, in our sum will be many of the same terms: if i is not a divisor n , Then there is such a divisor after calculating $\text{gcd}(i, n)$.

Consequently, for each divisor $d|n$ its

summand $k^{\gcd(d,n)} = k^d$ will take into account several times, i.e. the amount may be represented in the form:

where

re C_d - The number of such numbers i that $\gcd(i, n) = d$. Find an explicit expression for this quantity.

Any such number is: $i = dj$ Wherein $\gcd(j, n/d) = 1$ (Otherwise $\gcd(i, n) > d$).

Remembering Euler's function, We find that the number of such j is the value of the Euler function $\phi(n/d)$.

Thus, $C_d = \phi(n/d)$ And finally obtain formula:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

Applying Lemma Burnside together with the program calculations

Can not always be purely analytical way to obtain an explicit formula for the number of equivalence classes. In many problems, the number of permutations in the group may be too large for manual calculations and calculate analytically the number of cycles in them is not possible.

In this case, you must manually find a few "core" of permutations, which will be enough to generate the whole group. Then you can write a program that generates all permutations of the group G , Consider each of these cycles and substituting them into the formula.

Consider, for example **problem of the number of colorings of the torus**. There is a rectangular checkered paper $n \times m$, Some of the cells are painted in black. Then, this sheet was prepared from cylinder by gluing the two sides with lengths m . Then get out of the cylinder torus by gluing two circles (base cylinder) without twisting. Required to count the number of different tori (sheet was originally painted arbitrarily), assuming that the contact lines are indistinguishable, and the torus can rotate and flip.

In this problem, the representation can be considered a piece of paper $n \times m$. Some cells which are painted in black. It is easy to understand that the following types of transformations preserve equivalence class: cyclic shift rows of the worksheet, the cyclic shift column sheet, turn the sheet 180 degrees; also can be understood intuitively that these three kinds of transformations is enough to generate the whole group of invariant transformations. If we somehow enumerate cells field, then we can write

three permutations p_1, p_2, p_3 Corresponding to these types of transformations. Then we have to generate all permutations obtained as a product of this. Obviously, all such permutations have the form

$p_1^{i_1} p_2^{i_2} p_3^{i_3}$ Wherein $i_1 = 0 \dots m-1, i_2 = 0 \dots n-1, i_3 = 0 \dots 1$.

Thus, we can write the implementation of solving this problem:

```
void mult (vector <int> & A, const vector <int> & B)
    { vector <int> Aa (a);
    for (size_t i =0; i <a.size(); +
        + I) a[i] = Aa[b[i]];
}

int cnt_cycles (vector <int> A)
    { int res = 0;
    for (size_t i =0; i <a.size(); +
        + I) if (a[i] , = -1) {
            + + Res;
            for (size_t j = i; a[j] , = -1;
                ) { size_t nj = a[j];
                    a[j] = -1;
                    j = nj;
                }
            }
    return res;
}

int main() {
    int n, m;
    cin >> N >> m;

    vector <int> P (n * m), P1 (n * m), P2 (n * m), P3 (n
        * m); for (int i =0; i <n * m; + + I) {
```

```
p[i] = I;
p1[i] = (i% n + 1) % N + i / n * n;
p2[i] = (i / n + 1) % M * n + i% n;
p3[i] = (m - 1 - I / n) * N + (n - 1 - I% n);

}

int sum = 0, Cnt = 0;
set <vector <int>> S;
for (int i1 =0; i1 <n; + + I1) {
    for (int i2 =0; i2 <m; + + I2) {
        for (int i3 =0; i3 <2; + +
            I3) { if (!
S.count(p)) {
            s.insert (p);
            + + Cnt;
            sum + = 1 << Cnt_cycles(p);
        }
        mult (p, p3);
    }
    mult (p, p2);
}
mult (p, p1);
}

cout << Sum / cnt;
}
```

The principle of inclusion-exclusion

The principle of inclusion-exclusion - this is an important combinatorial trick to calculate the size of any of the sets, or to calculate the probability of complex events.

Formulation of the principle of inclusion-exclusion

The wording

The principle of inclusion-exclusion is as follows:

To calculate the size of combining multiple sets, it is necessary to sum the sizes of these sets **separately**, then subtract all sizes **pairwise** intersection of these sets, add

intersections of all possible sizes back **triples** sets, subtract the size of intersections

quadruples And so on up to the intersection **all** sets.

Formulation in terms of sets

In mathematical form, the above wording is as follows:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{i=1}^n |A_i| - \sum_{\substack{i,j: \\ 1 \leq i < j \leq n}} |A_i \cap A_j| + \sum_{\substack{i,j,k: \\ 1 \leq i < j < k \leq n}} |A_i \cap A_j \cap A_k| - \dots + (-1)^{n-1} |A_1 \cap \dots \cap A_n|.$$

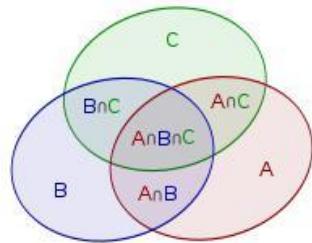
It can be written in a more compact, a sum of over subsets. We denote B set whose elements are A_i . Then the principle of inclusion-exclusion takes the form:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{C \subseteq B} (-1)^{\text{size}(C)-1} \left| \bigcap_{e \in C} e \right|.$$

This formula is credited Moivre (Abraham de Moivre).

Formulation using Venn diagrams

Let the diagram marked three figures A, B and C :



Then the area of association $A \cup B \cup C$ equal to the sum of the areas, and less double coated areas $A \cap B, A \cap C, B \cap C$, But with the addition of three covered surface $A \cap B \cap C$:

Similarly, it can be generalized to the union n figures.

Formulation in terms of probability theory

If A_i - This event - their probability, the probability of their union (ie what happens at least one of these events) is equal to:

$$\begin{aligned}\mathcal{P}\left(\bigcup_{i=1}^n A_i\right) &= \sum_{i=1}^n \mathcal{P}(A_i) - \sum_{\substack{i,j: \\ 1 \leq i < j \leq n}} \mathcal{P}(A_i \cap A_j) + \\ &+ \sum_{\substack{i,j,k: \\ 1 \leq i < j < k \leq n}} \mathcal{P}(A_i \cap A_j \cap A_k) - \dots + (-1)^{n-1} \mathcal{P}(A_1 \cap \dots \cap A_n).\end{aligned}$$

This amount can also be written as a sum over subsets of B whose elements are events A_i :

$$\mathcal{P}\left(\bigcup_{i=1}^n A_i\right) = \sum_{C \subseteq B} (-1)^{\text{size}(C)-1} \cdot \mathcal{P}\left(\bigcap_{e \in C} e\right).$$

Proof of inclusion-exclusion principle

To prove convenient to use the mathematical formulation in terms of set theory:

$$\left|\bigcup_{i=1}^n A_i\right| = \sum_{C \subseteq B} (-1)^{\text{size}(C)-1} \left|\bigcap_{e \in C} e\right|,$$

where B Recall - is the set consisting of A_i 's.

We need to prove that any element contained in at least one of the sets once. (Note that the other elements that are not contained in one of because there is no right-hand side of the formula).

Will allow for exactly the A_i formula
 A_i . Can not be accounted for,

Consider an arbitrary element x Contained exactly $k \geq 1$ sets A_i . We show that it formula will be counted only once.

Note that:

- in terms of those who $\text{size}(C) = 1$ Element x take into account exactly once, with a plus sign;
- in terms of those who $\text{size}(C) = 2$ Element x take into account (with a minus sign) exactly C_k^2 times - because x will be counted only in those terms, which correspond to the two sets of k sets containing x ;
- in terms of those who $\text{size}(C) = 3$ Element x take into account exactly C_k^3 times with a positive sign;
- ...
- in terms of those who $\text{size}(C) = k$ Element x take into account exactly C_k^k times with the sign of $(-1)^{k-1}$;
- in terms of those who $\text{size}(C) > k$ Element x take into account the zero times.

Thus, we need to calculate a sum binomial coefficients:

$$T = C_k^1 - C_k^2 + C_k^3 - \dots + (-1)^{i-1} \cdot C_k^i + \dots + (-1)^{k-1} \cdot C_k^k.$$

The easiest way to calculate this amount by comparing it with the expansion of the binomial theorem in the expression $(1-x)^k$:

$$(1-x)^k = C_k^0 - C_k^1 \cdot x + C_k^2 \cdot x^2 - C_k^3 \cdot x^3 + \dots + (-1)^k \cdot C_k^k \cdot x^k.$$

We see that at $x = 1$ expression is nothing else than $1 - T$. Consequently,

$$T = 1 - (1-1)^k = 1, \text{ As required.}$$

Use in solving problems

The principle of inclusion-exclusion is difficult to understand without learning good examples of its applications.

First, we consider three simple tasks "on paper", illustrating the application of the principle, and then consider the more practical problems that are difficult to solve without the use of the principle of inclusion-exclusion.

Of particular note is the problem of "search number of ways," because it demonstrates that the principle of inclusion-exclusion can sometimes lead to the polynomial solutions, not necessarily exponential.

Simple task of permutations

How many permutations of the numbers from to such that the first element is more
0 9

1 And the last - less 8 ?

Count the number of "bad" permutations, ie those in which the first element

≤ 1 and / or last ≥ 8 .

We denote X the set of permutations in which the first element ≤ 1 And by - in which the last element ≥ 8 . Then the amount of "bad" permutations on inclusion-exclusion formula equals:

After spending a simple combinatorial calculations, we find that it is equal to:

$$2 \cdot 9! + 2 \cdot 9! - 2 \cdot 2 \cdot 8!$$

Subtract this number from the total number of permutations $10!$, We get a response.

Simple task of (0,1,2)-sequences

How many sequences of length n Consisting only of numbers 0, 1, 2 Wherein each number occurs at least once?

Again, we pass to the inverse problem, ie assume the number of sequences which are not present in at least one of the properties.

Let ($i = 0 \dots 2$) The set of sequences in which the number is not found i . Then, by inclusion-exclusion by "bad" sequences is:

The dimensions of each of the A_i are obviously 2^n (Since such sequences may occur only two kind of numbers.) Capacity of each pairwise intersection $A_i \cap A_j$ are equal (as is available only one digit). Finally, the cardinality of the intersection of all three sets is 0 (As available figures do not remain).

Remembering that we solve the inverse problem, we obtain the final answer:

$$3^n - 3 \cdot 2^n + 3 \cdot 1 - 0.$$

Number of integral solutions of the equation

Given the equation:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 20,$$

where all $0 \leq x_i \leq 8$ (Wherein $i = 1 \dots 6$).

Required to count the number of solutions.

Forget about the first limitation $x_i \leq 8$ And simply count the number of non-negative solutions of this equation.

This is easily done through binomial coefficients - We want to break 20 elements on the groups, i.e.

distribute the "walls" separating groups, 25 places:

Now count on inclusion-exclusion formula by "bad" decisions, ie solutions of these in which one or more x_i more 9.

We denote A_k (Wherein $k = 1 \dots 6$) The set of solutions of an equation in which $x_k \geq 10$ And all other $x_i \geq 0$ (All $i \neq k$). To calculate the size of the set A_k essentially the a combinatorial problem that is solved in two paragraphs above, only now 9 items excluded from consideration and exactly belong to the same group. Thus:

$$|A_k| = C_{16}^5$$

Similarly, the cardinality of the intersection of two sets A_k and A_p equal to the number:

$$|A_k \cap A_p| = C_7^5$$

The power of each of three or more sets is equal to zero, since 20 elements is not enough for three or more variables, more than or equal to 9.

Combining all this inclusion-exclusion formula and given that we solve the inverse problem, we finally obtain **answer**:

$$C_{25}^5 - C_6^1 \cdot C_{16}^5 + C_6^2 \cdot C_7^5.$$

Number of relatively prime numbers in a given interval

Let numbers n and r . Required to count the number of numbers in the interval $[1; r]$ prime to n . Go straight to the inverse problem - do not count the number of relatively prime integers.

Consider all prime divisors n ; denote them by p_i ($i = 1 \dots k$). How many numbers in the interval $[1; r]$ divisible by p_i ? Their number is:

$$\left\lfloor \frac{r}{p_i} \right\rfloor$$

However, if we simply sum up these numbers, we get the wrong answer - some numbers to be added together several times (the ones that are divided to several p_i). Therefore it is necessary to use inclusion-exclusion formula.

For example, you can iterate over a subset of all p_i 's, Find them work, and add or subtract to the inclusion-exclusion formula next term.

Summary **implementation** to count the number of relatively prime numbers:

```
int solve (int n, int r) {
    vector<int> P;
    for (int i = 2; i * i <= n;
        ++i) if (n % i ==
        0) {
        p.push_back
        (i); while (n % i ==
        0)
            n /= i;
    }
    if (n > 1)
        p.push_back (n);

    int sum = 0;
    for (int msk = 1; msk < (1 << P.size()); ++
        Msk) { int mult = 1,
        bits = 0;
        for (int i = 0; i < (int)p.size(); +
            +i) if (msk & (1 << I)) {
            ++bits;
            mult *= p[i];
        }

        int cur = r / mult;
        if (bits % 2 == 1)
            sum += cur;
        else
            sum -= cur;
    }

    return r - sum;
}
```

Asymptotics of the solution is $O(\sqrt{n})$.

The number of integers in a given interval, multiple at least one of the given numbers

Dana n Numbers a_i and by r . Required to count the number of numbers in the interval $[1; r]$ The multiples at least one of a_i .

An algorithm for solving almost identical to the previous task - make inclusion-exclusion formula on numbers a_i each term in this equation - the number of numbers divisible by a given subset of properties a_i (In other words, on their dividers **least common multiple**).

Thus, the decision boils down to is that for 2^n iterate over a subset of numbers for their operations to find the least common multiple, and add or subtract from the answer the next value.

The number of rows that satisfy a number of patterns

Given n patterns - lines of equal length, consisting only of letters and question marks. Also, given the number k . Required to count the number of rows that satisfy exactly n patterns or, in another formulation, at least k patterns.

Note first that we can **easy to count the number of rows** Satisfying all of the right patterns. To do this, simply "cross" these patterns: A look at the first character (whether in all the patterns in the first position, the question is whether or not at all - if the first character is uniquely defined) for the second character, etc.

Now learn how to solve **the first version of the problem** When the desired line must satisfy exactly k patterns.

$$X \quad k$$

To do this, let's look over and zafksiruem specific subset size patterns - now we have to count the number of rows that satisfy this set of patterns and only him. For this we use the inclusion-exclusion formula we sum over a superset of the set X And either added to the current account, or subtract from it the number of rows that correspond to the current set:

$$ans(X) = \sum_{Y \supseteq X} (-1)^{|Y|-k} \cdot f(Y),$$

where $f(Y)$ denotes the number of rows that correspond to a set of patterns Y .

If we sum $ans(X)$ all X , We get the answer:

$$ans = \sum_{X : |X|=k} ans(X).$$

However, by doing so we got the solution in a time of $O(3^k \cdot k)$.

Solution can be accelerated, noting that different $ans(X)$ summation is over often one and the same sets Y .

Turn the inclusion-exclusion formula and will conduct a summation over Y . Then it is easy to understand that many Y take into account in $C_{|Y|}^k$ inclusion-exclusion formula, all with the same sign $(-1)^{|Y|-k}$:

$$ans = \sum_{Y : |Y| \geq k} (-1)^{|Y|-k} \cdot C_{|Y|}^k \cdot f(Y).$$

The decision came with the asymptotic $O(2^k \cdot k)$.

We now turn to a **second embodiment of the problem** When the desired line must satisfy at least k patterns.

Clearly, we can simply use the solution of the first version of the problem and summarize the answers from to n . However, you will notice that all the arguments will continue to be true, but in this version of the problem by programming X is not only for those sets whose size is equal to, and over all sets with size $\geq k$.

Thus, in the final formula before $f(Y)$ will stand another factor: not one binomial factor with some familiar, and their sum:

$$\sum_{k=0}^m (-1)^k \cdot C_n^k = (-1)^m \cdot C_{n-1}^m.$$

Looking into Graham (Graham, Whip Patashnik. **"Concrete Mathematics"** [1998]), We see a well-known formula for binomial coefficients:

$$\sum_{k=0}^m (-1)^k \cdot C_n^k = (-1)^m \cdot C_{n-1}^m.$$

Applying it here, we see that the whole sum of the binomial coefficients is minimized in:

$$(-1)^{|Y|-k} \cdot C_{|Y|-1}^{|Y|-k}.$$

Thus, for this version of the problem we got the solution with the asymptotic $O(2^k \cdot k)$:

$$ans = \sum_{Y : |Y| \geq k} (-1)^{|Y|-k} \cdot C_{|Y|-1}^{|Y|-k} \cdot f(Y).$$

Number of ways

There is a field $n \times m$ Some k cells which - impenetrable wall. In the field in the cell $(1, 1)$ (Bottom left

cell) is initially robot. The robot can only move right or up, and eventually he should get squared (n, m) . Avoiding all obstacles. Required to count the number of ways in which he can do it. Assume that the dimensions n and m very large (say, up to 10^9), And the amount of k - Small (of the order 100).

To solve immediately for convenience **sort the** obstacles in the order in which we can get around them: ie, for example, the coordinate x , And at equality - coordinate y .

Also learn how to solve the problem immediately without obstacles: ie learn to count the number of ways to reach from one cell to another. If one coordinate we need to go x cells, and on the other - y cells, from simple combinatorics, we obtain a formula through [binomial coefficients](#):

$$C_{x+y}^x$$

Now count the number of ways to reach from one cell to another, avoiding all obstacles, you can use **inclusion-exclusion formula**: Count the number of ways of walking, stepping at least one obstacle.

For this example, you can iterate over a subset of the obstacles which we do come, count the number of ways to do it (just multiplying the number of ways to reach from the start to the first cell of the selected obstacles, the first obstacle to the second, and so on), and then add or subtract a number from the response in accordance with the standard inclusion-exclusion formula.

However, this again is a non-polynomial solution - for the asymptotic behavior

$$O(2^k k)$$

get **polynomial solution**.

Will solve **dynamic programming**: Learn how to calculate the number $d[i][j]$ - Number ways to reach from the second point to the second, while not stepping on any one obstacle (except the i and j of course). In total we will $k + 2$ point as to the obstacles added start and end cells.

If we for a moment forget about all the obstacles and simply count the number of paths from cells i in a cage, we thus take into account some of the "bad" way through obstacles. Learn to count the number of these "bad" tract. Iterate over the first obstacle $i < t < j$. To which we come, then the number of paths is equally $d[i][t]$. Multiplied by the number of arbitrary ways in $t \rightarrow j$. Summing it all, we count the number of "bad" ways.

. Therefore, the solution of the whole

Thus, the value of $d[i][j]$ we have learned to count during $O(k)$ problem has the asymptotic $O(k^3)$.

Number coprimes quadruples

Given n numbers: a_1, a_2, \dots, a_n . Required to count the number of ways to choose four numbers of them so that their aggregate the greatest common divisor is equal to unity.

We solve the inverse problem - count the number of "bad" quadruples, ie such quadruples in which all numbers are divisible by $d > 1$.

We use the inclusion-exclusion formula, summing the number of fours divisible by the divisor d (but perhaps more dividers and divider)

$$ans = \sum_{d \geq 2} (-1)^{\deg(d)-1} \cdot f(d),$$

where $\deg(d)$ - Is the number of prime number factorization d , $f(d)$ - The number of fours, dividers for d .

To calculate the function $f(d)$ you simply count the number of numbers divisible d And [binomial coefficient](#) count the number of ways to choose four of them.

Thus, using the inclusion-exclusion formula, we summarize the number of fours divisible by primes, then subtract the number of quads that are divisible by the product of two primes, we add four divisible by three simple, etc.

Harmonic number of triples

Given the number $n \leq 10^6$. Required to count the number of triples $2 \leq a < b < c \leq n$ They triples are harmonic, ie:

- or $\gcd(a, b) = \gcd(a, c) = \gcd(b, c) = 1$,
- or $\gcd(a, b) > 1 \quad \gcd(a, c) > 1 \quad \gcd(b, c) > 1$.

First, go straight to the inverse problem - ie count the number of non-harmonic triples.

Secondly, we note that any non-harmonic exactly two triple its numbers are in such a situation, that this number is relatively prime to the number one triple and not just a one with a different number threes.

Thus, the amount equal to the sum of non-harmonic triples over all numbers from 2 to n works number of relatively prime to the current number of properties on the number of non-mutually prime numbers.

Now all that remains for us to solve the problem - is to learn to count every number in the interval $[2; n]$ number of numbers relatively prime (or coprime) with him. Although this task has already been highlighted, the solution described above is not appropriate here - it requires the factorization of each of the numbers 2 to n And then iterate through all possible products of primes in the factorization.

Therefore we need a faster solution that calculates the answers to all the numbers in the interval $[2; n]$ immediately. This can implement such **modification of the sieve of Eratosthenes**:

- First, we need to find all the numbers in $[2; n]$ interval in which no prime factorization is not included twice. In addition, for inclusion-exclusion formula, we need to know how simple factorization contains every such number.

To do this we need to have arrays that store for each number the number of primes in its factorization, and $good[]$ - Containing for each number $true$ or $false$. All simple enter it in power ≤ 1 or not.

Thereafter, during the sieve of Eratosthenes in processing the next prime number, we'll go over all multiple of the current number, and increase them, and all the numbers divisible by the square of the current simple

- Put $good = false$.

- Secondly, we need to calculate the answer for all the numbers from 2 to n le array $cnt[]$ - The number of numbers that are not relatively prime to the data.

For this, we recall how the inclusion-exclusion formula - here we actually implement it the same, but with an inverted logic: if we iterate term and see in which inclusion-exclusion formula for what number this term included.

So, suppose we have a number for which $good[i] = true$ ie this number is participating in formula inclusion-exclusion. Iterate through all the numbers that are And to multiples i answer $cnt[i]$ Each of these properties, we have add or subtract value $\lfloor N/i \rfloor$. Zodiac - addition or subtraction - depends on $deg[i]$ If $deg[i]$ odd, then we must add, subtract otherwise.

Implementation:

```

int n;
bool good[MAXN];
int deg[MAXN], Cnt[MAXN];

long long solve() {
    memset(good, 1, sizeof good);
    memset(deg, 0, sizeof deg);
    memset(Cnt, 0, sizeof Cnt);

    long long ans_bad = 0;
    for (int i = 2; i <= n; ++i)
        if (good[i])
            {
                if (deg[i] == 0) deg[i] = 1;
                for (int j = 1; i * j <= n; ++j)
                    if (j > 1 && Deg[i] == 1)
                        if (j % i == 0)
                            good[i * j] = false;
                        else
                            ++Deg[i * j];
                    Cnt[i * j] += (n / i) * (deg[i] % 2 == 1 ? +1 : -1);
            }
    ans_bad += (Cnt[i] - 1) * 111 * (n - 1 - Cnt[i]);
}

return (n - 1) * 111 * (n - 2) * (n - 3) / 6 - Ans_bad / 2;
}

```

Asymptotics of such solutions is about iterations of a nested loop.

Because almost every number it makes

The number of permutations with no fixed points

We prove that the number of permutations of length n without fixed points is equal to the following number:

$$n! - C_n^1 \cdot (n-1)! + C_n^2 \cdot (n-2)! - C_n^3 \cdot (n-3)! + \dots \pm C_n^n \cdot (n-n)!$$

and approximately equal to the number:

$$\frac{n!}{e}$$

(Moreover, if the expression is rounded to the nearest whole - you get exactly the number of permutations with no fixed points)

We denote A_k set of permutations of length n with a fixed point at the position k ($1 \leq k \leq n$). We now use the inclusion-exclusion formula to calculate the number of permutations with at least one fixed point. To do this, we need to learn to count-size sets of intersections as follows:

A_i They

because if we know that the number of fixed points is x , We thus know the position x permutation elements, and all other $(n-x)$ elements can stand anywhere.

Substituting this in the inclusion-exclusion formula and taking into account that the number of ways to select a subset of size x of n -Element set equal C_n^x Obtain a formula for the number of permutations with at least one fixed point:

$$C_n^1 \cdot (n-1)! - C_n^2 \cdot (n-2)! + C_n^3 \cdot (n-3)! - \dots \pm C_n^n \cdot (n-n)!$$

Then the number of permutations without fixed points is:

$$n! - C_n^1 \cdot (n-1)! + C_n^2 \cdot (n-2)! - C_n^3 \cdot (n-3)! + \dots \pm C_n^n \cdot (n-n)!$$

Simplifying this expression, we obtain **exact and approximate expressions for the number of permutations with no fixed points**:

$$n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \dots \pm \frac{1}{n!} \right) \approx \frac{n!}{e}.$$

(Since the sum in brackets - this is the first $n+1$ members of the Taylor series expansion e^{-1})

In conclusion, it is worth noting that in a similar way to solve the problem when you want to fixed points was not among m the first elements of the permutation (but not among all, as we have just solved). Will result in the formula as the above exact formula, only with the amount will go up k And not to n .

Problem in online judges

List of tasks that can be solved using the principle of inclusion-exclusion:

- [UVA # 10325 "The Lottery"](#) [Difficulty: Easy]
- [UVA # 11806 "Cheerleaders"](#) [Difficulty: Easy]
- [TopCoder SRM 477 "CarelessSecretary"](#) [Difficulty: Easy]
- [TopCoder TCHS 16 "Divisibility"](#) [Difficulty: Easy]
- [SPOJ # 6285 NGM2 "Another Game With Numbers"](#) [Difficulty: Easy]
- [TopCoder SRM 382 "CharmingTicketsEasy"](#) [Difficulty: Medium]
- [TopCoder SRM 390 "SetOfPatterns"](#) [Difficulty: Medium]
- [TopCoder SRM 176 "Deranged"](#) [Difficulty: Medium]
- [TopCoder SRM 457 "TheHexagonsDivOne"](#) [Difficulty: Medium]
- [SPOJ # 4191 MSKYCODE "SkyCode"](#) [Difficulty: Medium]
- [SPOJ # 4168 SQFREE "Square-free integers"](#) [Difficulty: Medium]
- [CodeChef "Count Relations"](#) [Difficulty: Medium]

Literature

- Debra K. Borkovitz. "Derangements and the Inclusion-Exclusion Principle"

Games on arbitrary graphs

Let the game is played by two players on a graph G. That is, current state of play - is a vertex of the graph, and each vertex of the edge goes to the vertices, which can go on the next move.

We consider the most general case - the case of an arbitrary directed graph with cycles. Required for a given initial position to determine who will win if both players play optimally (or determine that the result is a draw).

We will solve this problem very effectively - find the answers to all vertices of the graph in linear time relative to the number of edges - $O(M)$.

Description of the algorithm

About some of the vertices of the graph is known in advance that they are winning or as lost; obviously such vertices have no outgoing edges.

We have the following **data**:

- if some vertex of an edge in losing the top, then this vertex is winning;
- if some vertex of all the edges come in winning the top, then this vertex is losing;
- if at some point is still uncertain the top, but none of them do not fit the first, nor under the second rule, then all of these peaks - a draw.

Thus, the algorithm is already clear, working for the asymptotic behavior of $O(NM)$ - we iterate through all the vertices, try apply to every first or second rule, and if we have made any changes, then repeat all over again. However, this process of search and update can be greatly accelerated, bringing to a linear asymptotic behavior.

Iterate through all the vertices of which are initially known to winning or losing. From each of them let the next dfs. This dfs will move in reverse edges. Above all, he will not come to the top, which is defined as winning or losing. Further, if dfs tries to go from losing the top in some vertex, it marks it as winning, and goes into it. If dfs tries to go from winning the top in some vertex, it must verify that all edges are from this vertex in winning. This test is easy to implement in $O(1)$ if at each vertex will store counter edges that lead to winning the top. So, if dfs tries to go from winning the top in some vertex, then it increases its count, and if the counter equalizer with the number of edges emanating from that vertex, then this vertex is marked as losing, and depth-first search is b this vertex. Otherwise, if the target vertex and is not defined as winning or losing, the search

b depth does not enter into it.

Overall, we find that each winning and losing each vertex is visited by our algorithm exactly once, and no man tops and not visited. Consequently, the asymptotic behavior really $O(M)$.

Implementation

Consider the implementation of depth-first search, on the assumption that the graph is built in memory of the game, raising the outcome counted and recorded in degree (this is just a counter, it will decrease if there is an edge in winning the top), as well as winning or losing initially vertices already labeled.

```
vector <int> g
[100]; bool win
[100];
bool loose [100];
bool used [100];
int degree [100];

void dfs (int v) {used
    [v] = true;
    for (vector <int> :: iterator i = g [v]. begin (); i! = g [v]. end
        (); + + i) if (! used [* i]) {
            if (loose [v])
                win [* i] = true;
            else if (- degree [* i] ==
0)
                loose [* i] = true;
            else
                continue;
            dfs (* i);
        }
}
```

Example task. "Police and Thief"

To algorithm become more clear, we consider it a specific example.

Condition of the problem. There is a field size MxN cells in some cells can not go. Known initial coordinates police and thief. Also on the card may be present output. If a police officer would be in the same cage with the thief, the police won. If the thief would be in the cell with the output (in this cell should not police), will win a thief. Cop can walk in 8 directions, the thief - only 4 (along the coordinate axes). And the policeman and the thief may pass their turn. The first move is made by a police officer.

Graphing. Construct a graph of the game. We need to formalize the rules of the game. Current state game defined by the coordinates of police P, thief T, as well as boolean Pstep, which determines who will make the next move. Consequently, the vertex of the graph is defined triple (P, T, Pstep). Count to build easily by simply matching condition.

Next you need to determine which vertices are won or as lost initially. There are **delicate moment**. Winning / losing vertex coordinates in addition depends on the Pstep - whose now course. If you now move the police, then the top of winning, if the coordinates of a policeman and the thief are the same; top of losing if it is not successful and the thief is on the way. If the thief's turn, the top winning, if the thief is on the way, and losing, if it is not successful and coordinate police and thief are the same.

The only time that you need to decide - to build **graph explicitly or do it "on the run"** Right in the DFS. On the one hand, if you build a graph in advance, it will be less likely to make a mistake. On the other hand, it will increase the amount of code, and the work will be several times slower than if you build a graph "on the fly".

Implementation the entire program:

```
struct state
    {char
     p, t;
     bool
     pstep
     ;
    };

vector <state> g [100] [100] [2];
// 1 = policeman coords; 2 = thief coords; 3 = 1 if policeman's
step or 0 if thief's.
bool win [100]
[100] [2]; bool
loose [100] [100]
[2]; bool used
[100] [100] [2];
int degree [100]
[100] [2];

void dfs (char p, char t, bool
pstep) {used [p] [t]
[pstep] = true;
for (vector <state> :: iterator i = g [p] [t] [pstep]. begin
(); i! = g [p] [t] [pstep]. end (); + + i)
if (! used [i-> p] [i-> t]
[i-> pstep]) {if
(loose [p] [t]
[pstep])
win [i-> p] [i-> t] [i-> pstep]
= true; else if (- degree [i-> p] [i->
t] [i-> pstep] == 0)
loose [i-> p] [i-> t] [i-> pstep] = true;
else
continue;
dfs (i-> p, i-> t, i-> pstep);
}
}
```



```
int p_st, t_st;
for (int i = 0; i <n; +
+ i)
    for (int j = 0; j <m; + + j)
        if (a [i] [j] == 'C')
            p_st = i * m + j;
        else if (A [i] [j] == 'T')
            t_st = i * m + j;

cout << (win [p_st] [t_st] [true]? "WIN": loose [p_st]
[t_st] [true]? "LOSS": "DRAW");
}
```

Theory Shpraga Grande. It

Introduction

Theory Shpraga Grande - a theory that describes the so-called **equal** (Eng. "impartial") play two players, ie games in which the permitted moves and winning / losing only depend on the state of the game. Matter which of the two players go, does not depend on anything: ie players are completely equal.

In addition, it is assumed that the players have all the information (about the rules of the game, the possible moves, the position of opponent).

It is assumed that the game **finite** for any strategy players will sooner or later come to **losing** position from which there are no transitions to other positions. This position is for a losing player who must make a move from this position. Accordingly, it is advantageous for a player who came into this position. Clearly, a draw in this game does not happen.

In other words, this game can be completely described **directed acyclic Count**: vertices in it are the state of the game, and ribs - Transitions from one state to another game as a result of the progress of the current player (again, in the first and second player are equal). One or more

vertices have no outgoing edges, it is as lost vertices (for a player who has to make the course of such vertices).

Since a tie does not happen, then all the states of the game are divided into two classes: **winning and losing**. Winning - these are status that there is a move of the current player, which will to inevitable defeat another player even if his best game. Accordingly, losing the state - a state from which all transitions lead to states, leading to the victory of the second player, in spite of the "resistance" to the first player. In other words, winning is a state from which there is at least one transition in a losing state, and losing is a condition from which all transitions lead to winning the state (or from which there are no transitions).

Our task - for any given game to classify the states of this game, ie for each state to determine winning or losing it.

Theory of games independently developed Shprag Roland (Roland Sprague) in 1935 and Patrick Michael Grundy (Patrick Michael Grundy) in 1939

Game "Him"

This game is one of the examples described above games. Moreover, as we shall see later, **any** Games of equal two players actually equivalent to the game "it" (Eng. "nim"), so the study of this game will allow us to automatically solve all the rest of the game (but more on that later).

Historically, this game was popular in ancient times. Probably, the game has its origins in China - at least, the Chinese game "Jianshizi" is very similar to him. In Europe, the first mention of Nimes refer to the XVI century. The name "it" came up with mathematician Charles Bud (Charles Bouton), which in 1901 published a full analysis of this game. Origin of the name "him" is not known.

Game Description

Game "it" is a next game.

There are several piles, in which each of several stones. In one move, the player can take any one of a handful of any non-zero number of stones and throw them away. Accordingly, the loss occurs when there are no more moves, ie all the piles are empty.

So, the state of the game, "it" is uniquely described by an unordered set of natural numbers. In one move allowed strictly to reduce any of the numbers (if the resulting number will be zero, it is removed from the set).

Solution of neem

The solution to this game published in 1901 by Charles Bud (Charles L. Bouton), and it looks as follows.

Theorem. The current player has a winning strategy if and only if when XOR-sum of the sizes of heaps different from zero. Otherwise, the current player is in a losing position. (XOR-sum of the numbers a_i is an expression $a_1 \oplus a_2 \oplus \dots \oplus a_n$ Wherein \oplus - Bitwise exclusive or)

Proof.

The main essence of the proof below - available **symmetric strategy**

for the enemy. We show that being in a state of zero XOR-sum the player can not get out of this state - in any of its transition to a state with nonzero XOR-sum of the enemy there is a retaliatory move that returns the XOR-sum back to zero.

We now proceed to the formal proof (it will be constructive, ie, we show how it looks symmetrical strategy opponent - exactly what you will need to move him to perform).

Will prove the theorem by induction.

To empty neem (when the size of all piles zero) XOR-sum is zero, and the theorem is true.

Suppose now that we want to prove the theorem for a game state from which there is at least one transition. Using the induction hypothesis (and acyclicity of the game), we believe that the theorem is proved for all the states in which we can get from this.

Then the proof falls into two parts: if the XOR-sum s in the current state $= 0$ It is necessary to prove that the current state is losing, ie all transitions from it lead to states with XOR-sum. If the

$s \neq 0$ It is necessary to prove that there is a transition that leads us to a state of $t = 0$.

- Let $s = 0$, Then we want to prove that the current state - disadvantageous. Consider any transition from its current state of neem: denote P Rooms vary a handful through $x_i (i = 1 \dots n)$ - The size of piles to travel through $y_i (i = 1 \dots n)$ - After a stroke. Then, using the elementary properties of the function \oplus We have:

$$t = s \oplus x_p \oplus y_p = 0 \oplus x_p \oplus y_p = x_p \oplus y_p.$$

However, since $t \neq 0$ means that. Hence, the new state will have a nonzero XOR-sum, ie, according to the basis of induction is advantageous, as required.

Let. Then our task - to prove that the current state - a winner, ie of course it exists in a losing state (zero XOR-sum).

Consider the bit representation of the number s . Take senior nonzero bit, let his room number. Let - the number of the heap, in size x_k which th bit different from zero (there is, otherwise in XOR-sum s This bit would not have been different from zero).

Then, allegedly sought move - change it k Th pile, making its size $y_k = x_k \oplus s$. Let us prove this.

You should first check that it is the correct move, ie that $y_k < x_k$ However, it is true, as all the bits, the senior d Th, y_k and x_k coincide, and d Th bit in y_k will be zero, while the x_k will be one.

Now calculate what XOR-sum give the course:

$$t = s \oplus x_k \oplus y_k = s \oplus x_k \oplus (s \oplus x_k) = 0.$$

Thus, we specify the course - really move in a losing state, and this proves that the current state winner.

The theorem is proved.

Consequence. Any condition them games can be replaced by an equivalent condition consisting of only a handful of size equal to the XOR-sum of the sizes of piles in the old state.

In other words, the analysis of neem with several small groups can calculate the amount of XOR- s their size, and proceed to the analysis of only a handful of neem size s - As shown by the theorem just proved, winning / losing will not change.

Equivalence of any game nimu. Theorem Shpraga Grande

Here we show how any game (game two equal players) put into correspondence with him. In other words, any state of any game we will learn how to put them in line-pile, fully describing the state of the original game.

Lemma of Nimes with increases

We first prove a very important lemma - **Lemma of Nimes with increases**.

Namely, consider the following modified them all in the same way as the conventional Nima, but now form an additional stroke permitted instead decrease, on the contrary, **increase the size of a handful of**. To be more precise, that player's turn now is he or takes a non-zero number of stones from some piles, or increases the size of a handful (in

according to certain rules, see next paragraph).

It is important to understand that the rules of how the player can make an increase, **We are not interested**

- But these rules still need to be to our game was still **acyclic**. Below in the section "Examples of games" are considered examples of such games: "stair him", "nimble-2", "turning turtles".

Again, the lemma will be proved by us at all for any of this kind of games - games like "them with increases"; specific rules increases in the proof can not be used.

The formulation of Lemma. It increases with equivalent conventional nimu, in the sense that winning / losing status is determined by Theorem Bouton according XOR-sum of the sizes of heaps. (Or, in other words, the essence of the lemma that the increase useless, they do not make sense to apply the optimal strategy, and they do not change a winning / losing over conventional Nîmes.)

Proof.

The idea of the proof, as in Theorem Bouton - Available **symmetric strategy**. We show that the increase does not change anything, because after one of the players resort to increase, the other will be able to answer it symmetrically.

In fact, suppose that the current player makes a stroke-increasing piles. Then his opponent could just answer him, reducing back to this pile of old values - after the usual moves neem we still may be used freely.

Thus, a symmetric response to an increase in stroke-stroke-reduction will be back to the old size piles.

Therefore, after this answer game will go back to the same size piles, ie player who increase, do nothing for the win. Because acyclic game, sooner or later you run out of moves, increase, and current players have to make a move-reduction, and this means that the presence of increasing moves does not change anything.

Theorem Shpraga Grande on the equivalence of any game nimu

We now turn to the most important fact in this article - the equivalence theorem nimu any equitable two-player games.

Theorem Shpraga Grande. Consider any state v a game of two equal players. Let it out there in some state transitions v_i (Where). It is argued that as $v \xrightarrow{i} v_i$ ($i = 1 \dots k$) $k \geq 0$ games can assign a handful of neem certain size x (Which will fully describe the state of v our game - ie these two states, two different games are equivalent.) This number is x - Called **value Shpraga Grande** state v .

Moreover, by this x You can find the following recursive way: count value Shpraga Grande x_i for each transition (v, v_i) And then performed:

$$x = \text{mex}\{x_1, \dots, x_k\},$$

where the function **mex** from a set of numbers returns the smallest non-negative number is not found in this set (the name "mex" - an abbreviation of "minimum excludant").

Thus, we can, starting from the vertex without outgoing edges, gradually **calculate values Shpraga Grande for all the states of our game**. If the value of what Shpraga Grande- or state is zero, then this condition is disadvantageous otherwise -

advantageous. **Proof.** Will prove the theorem by induction.

For the vertices of which there is no transition, the value of x according to the theorem will be obtained as **mex** from the empty set, ie $x = 0$. But, in fact, the state without transitions - is losing condition, and he really should correspond to the size of a bunch of them- 0.

Consider now any state v From which there are transitions. By induction, we can assume that for all states v_i in which we can move from its current state, the values x_i already counted.

Calculate the value $p = \text{mex}\{x_1, \dots, x_k\}$. Then, by the definition of **mex**, We find that any number in between $[0; p)$ there is at least one corresponding transition into some of the v_i condition. In addition, there may be additional transitions - in the state with values Grundy large p .

This means that the current state of **equivalent to the state nimu with increases with heap size p** : In fact, we have a transition from the current state to the state with heaps of smaller sizes, and can also be transitions to the larger sizes.

Consequently, the value $\text{mex}\{x_1, \dots, x_k\}$ is the desired value Shpraga Grande for the current status, as required.

Application of Theorem Shpraga Grande

We describe finally holistic algorithm is applicable to any two-player game equal to determine the winning / losing the current state v .

Function that each of the game puts him in line-number is called **function Shpraga Grande**.

So to calculate the function Shpraga Grande for the current state of a game, you need:

- Write down all the possible transitions from the current state.
- Each transition may lead either to a single game or in **amount of independent games**.

In the first case - just count the Grundy function recursively for the new state.

In the second case, where the transition from the current state results in the sum of several independent games - recursively calculate for each of these games Grundy function, then we say that the function of the amount of games Grande is XOR-sum of the values of these games.

- Once we felt Grundy function for each possible transition - consider **mex** from these values, and found number - is the desired value for the current state Grande.
- If the value is zero Grande, the current status disadvantageous otherwise - advantageous.

Thus, in comparison with Theorem Shpraga Grande here we take into account the fact that the game can be transitions from individual states in **the sum of several games**. To work with the sums of games, we will first replace every game its value Grande, ie a bunch of them-some size. After that we come to him, the sum of several piles, ie nimu to normal, the answer to which, according to Theorem Bouton - XOR-sum of the sizes of heaps.

Regularities in the values Shpraga Grande

Very often when solving specific problems that require learning to count function Shpraga Grande for a given game, helps **study tables of values** this function in search of patterns.

In many games that seem quite difficult for theoretical analysis, feature Shpraga Grande in practice is periodic, or it has a very simple form that is easy to see "by eye".

In the majority of cases seen patterns are correct, and if you want to prove by mathematical induction.

However, not always function Shpraga Grande contains simple patterns, and for some, even very simple in formulation, games whether such laws are still open (for example, "Grundy's game" below).

Examples of games

To demonstrate the theory Shpraga Grande, we explain a few problems.

Especially should pay attention to the problem "stair him", "nimble-2", "turning turtles", which shows a non-trivial reduction of the original problem nimu with increases.

"Tic Tic"

Condition. Consider plaid stripes size $1 \times n$ cells. In one move, the player has to put one cross, but it is forbidden to put two cross near (to neighboring cells). The player who can not make a move. To say who will win the game at the optimum.

Decision. When a player puts a cross in any cell, we can assume that the whole band splits into two independent halves: the left of the cross and to the right. At the same time the cell itself with a dagger, and its left and right neighbor destroyed - because they can not be anything else to put.

Consequently, if we number the cells from strips 1 to n . Then put a cross in position $1 < i < n$, Band splits into two strips of length $i - 2$ and $n - i - 1$ ie we move into the sum of two games $i - 2$ and $n - i - 1$. If the cross is put in the position 1 or n , It is a special case - we just move on to the state $n - 2$.

Thus, the function Grande $g[n]$ form (for $n \geq 3$)

$$g[n] = \text{mex} \left\{ g[n-2], \bigcup_{i=2}^{n-1} (g[i-2] \oplus g[n-i-1]) \right\}.$$

Ie $g[n]$ obtained as **mex** from the set consisting of the number $g[n-2]$, As well as various

expression values $g[i-2] \oplus g[n-i-1]$.

So we've got a solution for this problem $O(n^2)$.

In fact, considering the computer table of values for the first hundred values n You can see that, since the sequence becomes periodic with a period. This pattern persists
 $n = 52$ 34 and more (which probably can be proved by induction).

"Noughts and crosses - 2"

Condition. Again, the game is on the strip $1 \times n$ cells, and players take turns putting one dagger. The player who put a dagger three in a row.

Decision. Note if $n > 2$ and we left after his move two cross near or over one space, the opponent wins the next move. Consequently, if one player has put a cross somewhere, then another player unprofitable put a cross in its neighboring cells, as well as in neighboring adjacent (ie at a distance 1 and 2 put unprofitable, it will lead to the defeat).

Then the solution is obtained substantially similar to the previous problem, but now the cross removes each half not one, but two cells.

"Pawns"

Condition. There is a field $3 \times n$ in which the first and third row are on n pawns - white and black, respectively. The first player goes white pawns, the second - black. Terms of stroke and stroke - standard chess, except that beat (subject to availability) is obligatory.

Decision. Trace, what happens when one pawn to make a move forward. The next move opponent will be obliged to eat it, then we would be required to eat a pawn of the opponent, then he will eat, and finally, our enemy pawn eat and stay, "resting" in the pawn of the opponent. Thus, if we go to the beginning column pawn $1 \leq i < n$
 $i = 2$ and $n - i - 1$. $i = 1$ and
 $i = n - 2$.

Condition. n

Decision.

$$g[n] = \text{mex} \left\{ \bigcup_{i=0}^{n-1} g[i], \bigcup_{i=1}^{n-1} (g[i] \oplus g[n-i]) \right\}.$$

n

$$\begin{array}{cccc} 4 & g[n] = n & 1 & 2 \\ & & 4 & g[n] = n \pm 1 \\ & & & 3 & 0 \end{array}$$

Condition. Yes n

Decision. -

It is easy to obtain an expression for the function Shpraga Grande:

$$g[n] = \text{mex} \left\{ \bigcup_{i=0}^{n-1} (g[i] \oplus g[n-1-i]), \bigcup_{i=0}^{n-2} (g[i] \oplus g[n-2-i]) \right\}.$$

Calculate the table for it for the first few tens of elements:

$g[0 \dots 11]$:	0	1	2	3	1	4	3	2	1	4	2	6
$g[12 \dots 23]$:	4	1	2	7	1	4	3	2	1	4	6	7
$g[24 \dots 35]$:	4	1	2	8	5	4	7	2	1	8	6	7
$g[36 \dots 47]$:	4	1	2	3	1	4	7	2	1	8	2	7
$g[48 \dots 59]$:	4	1	2	8	1	4	7	2	1	4	2	7
$g[60 \dots 71]$:	4	1	2	8	1	4	7	2	1	8	6	7
$g[72 \dots 83]$:	4	1	2	8	1	4	7	2	1	8	2	7
$g[84 \dots 95]$:	4	1	2	8	1	4	7	2	1	8	2	7
$g[96 \dots 107]$:	4	1	2	8	1	4	7	2	1	8	2	7
$g[108 \dots 119]$:	4	1	2	8	1	4	7	2	1	8	2	7

You may notice that, at some point, the sequence becomes periodic with period 12. In the future, this periodicity is also not violated.

Grundy's game

Condition. Yes n heaps of stones, the dimensions of which we denote by a_i . In one move, the player can take any pile of size at least 3 and divide it into two nonempty piles of unequal sizes. The player who can not move (ie, when the size of the remaining piles is less than or equal to two).

Decision. If $n > 1$, All these multiple piles are obviously - independent games. Therefore, our task

- Learn how to look for a function Shpraga Grande for one heap, and the answer for a few piles will be obtained as their XOR-sum.

For a handful of this function is constructed as easy enough to view all possible transitions:

$$g[n] = \text{mex} \left\{ \bigcup_{\substack{i=[1 \dots n-1], \\ i \neq n-i}} \left(g[i] \oplus g[n-i] \right) \right\}.$$

What is this game is interesting - the fact that so far it has been found for the general law. Despite

the assumption that the sequence must be periodic, it was calculated up until

$g[119]$

2^{35}

and frames in this region were detected.

"Stair him"

Condition. There is a staircase with n steps (numbered from 1 to n) On i Th second step is a_i coins. In one move is allowed to move a non-zero number of coins i Th second to $i-1$ Th step. The player who can not make progress.

Decision. If you try to reduce this problem to nimu "Head" it turns out that the course we have - this decrease a certain number of piles on, and a handful of other simultaneous increase at the same rate. As a result, we obtain a modification of the neem, which is very difficult to solve.

Proceed differently: consider only the odd-numbered steps: a_1, a_3, a_5, \dots . We will see how to change this set of numbers when making a turn.

If the move is made with an even i , Then this means an increase of stroke a_{i-1} . If the move is made with an odd i , It means decrease a_i .

It turns out that our problem - it is common with increases with the size of heaps a_1, a_3, a_5, \dots .

Consequently, the function Grande from him - is XOR-sum of numbers of the form a_{2i+1} .

"Nimble" and "Nimble-2"

Condition. There checkered strip , Coins which are located: th coin is $\frac{a_i}{k}$ Th cell. In one move, the player can take any value and move it to the left by an arbitrary number of cells, but so that she got out beyond the strip. In the game "Nimble-2" jump further prohibited other coins (or even put two coins in one cell). The player who can not make a move.

Decision "Nimble". Note that the coins in this game are independent of each other. Moreover, If we consider the set of numbers $a_i - 1$, It is clear that for one turn, a player may take any of these numbers and reduce it, but a loss occurs when all the numbers are equal to zero. Consequently, the game is "Nimble" - it usual itAnd the answer to the problem is the XOR-sum numbers $a_i - 1$.

Decision "Nimble-2". We enumerate the coins in the order they appear from left to right. Then we denote d_i distance from i Th to $i-1$ Second coin:

$$d_i = a_i - a_{i-1} - 1, \quad (i = 1 \dots k)$$

(Assuming that $a_0 = 0$).

Then one player can take away from some certain number q , And add the same number q to. Thus, this game - it's actually "Stair him" on numbers (we need only change the order of the numbers on the opposite).

"Turning turtles" and "Twins"

Condition. Dana plaid stripe size $1 \times n$. In each cell stands or cross, or toe. In one move, you can take some toe and turn it into a cross.

In this **additionally** permitted to choose one of the cells to the left of the variable and change it to the opposite value (ie replace the toe on the cross, and the cross - at the toe). In the game "turning turtles" do it not necessary (ie a player's turn may be limited to cross conversion Tac), and "twins" - necessarily.

Decision "turning turtles". Allegedly this game - it is a regular on the numbers , where b_i position i On Noughts (1-indexed). Verify this assertion.

- If a player simply reversed toe on the cross, without using an additional course - it can be understood as the fact that he just took the whole pile, corresponding to this crosses. In other words, if a player changed his toe on the cross at the position $x (1 \leq x \leq n)$, By the same token, he took a handful of size x and made her size zero.
- If a player takes an extra swing, ie besides the fact that he changed positions in the cross x to toe, he Change your position in the cell $y (y < x)$, We can assume that it reduced pile x to size y . Indeed, if the position of y used to be a cross - that, in fact, after the player's turn there will be a toe, ie bunch size is displayed y . And if the position of y used to toe, after the player's turn this bunch disappears - or, what is the same, there was a second bunch of exactly the same size y (As in Nimes two piles of equal size actually "kill" each other).

Thus, the answer to the problem - it is XOR-sum numbers - coordinates all zeros in the 1-indexed.

Decision "twins". All the arguments, above, remain true, except that progress "Zero heap" is now a player does not. Ie if we take away from all the coordinates unit - then again, the game will turn him into a regular.

Thus, the answer to the problem - it is XOR-sum numbers - coordinates all zeros in the 0-indexed.

Northcott's game

Condition. There is a board resolution $n \times m$: n rows and m columns. On each line are two chips: one black and one white. In one move, the player can take any piece of his color and move it inside the line to the right or to the left by an arbitrary number of steps, but without jumping over another chip (and not getting it). The player who can not make progress.

Decision. First clear each of n lines form an independent board game. Therefore, the task is to analyze the game in a row, and the answer to the problem is the sum of the XOR-Shpraga Grande for each of the rows.

Solving the problem for a single line, denoted by x the distance between the black and white chip (which can vary from zero to $m - 2$). In one move, each player can either reduce x for some arbitrary value, or possibly increase it to a certain value (increase are not always available). Thus, this game - it "**Him with increases**" And, as we already know, the increase in this game are useless. Consequently, the function Grande for one line - this is the distance x .

(It should be noted that such an argument is formally incomplete - as in "with increases in Nimes" assumed that the game **finite** And here are the rules of the game allow players to play indefinitely. However, an endless game can not take place at the optimal game - because one player stands to increase the distance x (Price approaching the edge of the field), the other player near him, reducing x back. Consequently, for optimal game opponent player will not be able to make moves increase indefinitely, so still described the solution remains in force.)

Triomino

Condition. Given the checkered field size $2 \times n$. In one move, a player can bet on the one figure in the shape of the letter "G" (ie, the shape of the three connected cells that do not lie on the same line). Forbidden to put a figure so that it crossed at least one cell with one of the previous set of figures. The player who can not make a move.

Decision. Note that the formulation of a figure breaks the whole field into two separate fields. Thus, we need to analyze not only the rectangular field, but the field in which the left and / or right edges are uneven.

Drawing a different configuration, you can see that whatever the configuration of the field, the main thing - just how many cells in this field. In fact, if the current field x free cells, and we want to break this field into two fields size y and z (Where) it is always possible, i.e. you can always find an appropriate place for figurines.

Thus, our task becomes such: initially we have a bunch of stones size $2n$ And in one move we can throw a handful of 3 stone and then beat this pile into two piles of arbitrary size. Grundy function for this game is:

$$g[n] = \text{mex} \left\{ \bigcup_{i=0}^{n-3} (g[i] \oplus g[n-i-3]) \right\}.$$

Chips on a graph

Condition. Given a directed acyclic graph. Some vertices of the graph are the chips. In one move the player can take any piece and move it along any edge in the new vertex. The player who can not make a move.

It can also happen, and the second version of this problem when it is considered that if the two pieces come in one vertex, then they both cancel each other out.

Solving the problem of the first embodiment. First all the chips - independent from each other, so Our task - to learn to look Grundy function for one chip in the graph.

Given that the graph is acyclic, we can do it recursively: suppose we thought Grundy function for all descendants of the current node.

Then the function Grande in the current top - it **mex** from this set of numbers.

Thus, the solution of the problem is the following: for each vertex count Grundy function recursively if it was a feature in this top. After that, the answer to the problem is the sum of the XOR-Grande from those vertices of the graph, which by hypothesis are chips.

Solution of the second variant of the problem. In fact, the second version of the problem is no different first. In fact, if the two pieces are in the same vertex of the graph, the resulting XOR-sum of their values Grande cancel each other out. Hence, in fact it is the same problem.

Implementation

From the perspective of implementation of interest may be the implementation of the function.

mex

If this is not the bottleneck in the program, you can write some simple option for **c** - The number of arguments)

$O(c \log c)$ (Wherein

```
int mex (vector <int> A) {
    set <int> B (a.begin(), A.end());
    for (int i = 0; ; ++ i)
        if (! b.count (i))
            return i;
}
```

However, not so complicated is an option in linear time for **mex**. We denote **D** constant equal to the maximum possible value **c** (That is the maximum degree of a vertex in the graph of the game). In this case, the function result **mex** will not exceed **D**.

Consequently, when implementing enough to have an array of size **D + 1** (Array of global or static - Most importantly, that it was not created for each function call). When calling a function **mex** we first note in all this array of arguments (skipping those that are the result). Then pass through this array we are for **O(c)** find the first unmarked element. Finally, end can again go through all the arguments and reset the array back to them. Thus, we will perform all actions for **O(c)**, Which in practice may be considerably less than the maximum power **D**.

```
int mex (const vector <int> & A) {
    static bool used[D + 1] = { 0
    }; int c = (int) a.size();
    for (int i = 0; i < c; ++ i)
        if (a[i] <= D)
            used[a[i]] = true;

    int result;
    for (int i = 0; ; ++ i)
        if (! used[i])
    {
```

```
result = i;  
2break;
```

```

        }

    for (int i = 0; i < c; ++ i)
        if (a[i] <= D)
            used[a[i]] = false;

    return result;
}

```

Another option - use the technique "**Numerical used**". I.e. do not array `used` Boolean variables and numbers ("versions"), and make a global variable indicating the current version number.

we increase the number of the current version, in the first cycle, we shall appear in the

Upon entering function `array`

And the current `version` number. Finally, in the second cycle, we simply compare `version` with the number of `used` `used[i]` the current

not true
the
version - if they do not match, it means that the current
number is not met in the array `a`. The third cycle
(which previously vanish array `used`) In such a
solution is not necessary.

Neem generalization: it Moore (k -It)

One of the interesting generalizations conventional neem was given by Moore (Moore) in 1910

Condition. Yes n heaps of stones size a_i . Also a natural number. In one move, the player can reduce the size of one to piles (ie now allowed simultaneous moves in several piles at once). The player who can not make progress.

Obviously, $k = 1$ Moore turns it into a regular him.

Decision. Solution of this problem is amazingly simple. Record the size of each of a handful of Binary value. Then sum these numbers-ary number system without hyphenation discharges. If sum is zero, the current position is losing, otherwise - winning (and from there move to the position with zero value).

In other words, for every bit we look, there is this bit or not in the binary representation of each number a_i . Then we summarize the resulting zero / one, and take the sum modulo. If as a result of this sum for each bit turned zero, then the current position - losing, otherwise - winning.

Proof. As for neem proof is to describe the strategies of the players: on the one hand, we show that the game with zero, we can only go into the game with a nonzero value, and on the other side - that of the game with a non-zero value is the course of the game to zero.

First, we show that the game with a zero value can only go into the game with a nonzero value. It is quite clear: if the sum modulo is zero, then after the change from one to the bit we could not get back a zero sum.

Second, we show how a non-zero sum game go to a zero sum game. Let's take the bits in which a nonzero amount, in order from highest to lowest.

We denote u the number of piles, which we have already begun to change; initially. Note that in these u piles we can put any bits at our request (as any handful of which fall). These u heaps already decreased from the previous one, older, bits).

Thus, let us consider the current bit, wherein the sum modulo through this amount, but in which no account is taken piles, which we have already begun to change. We denote the amount that may be obtained by these heaps current bit equal to one:

We have two options:

- if the $q \leq u$.

Then we can manage only been selected in groups: enough
They put the current bit equal to one, and all the rest - to zero.

$$k + 1 - s = u - q \text{ of}$$

- If $q > u$.

B Then we, on the contrary, we put in already selected u heaps current
bit equal to zero. Then the sum of the current bit is equal $s > 0$,
Which means, among unselected $n - u$ piles in the current bit has at
least s units. We choose some s heaps of them, and they reduce the
current bit from one to zero.

B result, the number u mutable heaps grow by s And make $q \leq k$.

Thus, we have shown how to choose the set of
variable piles and which bits should change them to
their total number u never exceeded k .

Consequently, we have proved that the desired transition from the zero-sum into a state with zero
sum exists, as required.

"Him giveaway"

That it that we considered throughout this article - also called "normal Nimes" ("normal nim"). In contrast, there is
also "**These giveaway**" ("Misère nim") - when a player
has made the last move loses (and not winning).

(Incidentally, appear to him like the board game - it is more popular in the version of "giveaway",
rather than "normal" version)

Decision such neem surprisingly simple: will act in the same way as in the usual Nimes (I.e., count XOR-sum of all
sizes of heaps, and if it is zero, then we lose with any strategy, and otherwise - to win by finding the transition to the
position with zero Shpraga Grande). But there is one **exception** if
sizes of all piles are unity, the winning / losing swapped compared with conventional Nimes. Thus, the winning /

losing neem "giveaway" is defined by number:

$$a_1 \oplus a_2 \oplus \dots \oplus a_n \oplus z,$$

where by z denotes a Boolean variable equal to one if $a_1 = a_2 = \dots = a_n = 1$.

With this exception, **optimal strategy** for the player in a winning position is determined as follows. Find a move
that would have made the player if he played him normal.

Now, if this move leads to a position in which the sizes of all the heaps are equal to one (and thus
before this move was a bunch of size greater than one), then the move is necessary to change:
change so that the number of remaining non-empty heaps changed its parity.

Proof. We note that general theory Shpraga Grande refers to "Normal" games rather than to games
giveaway. But it - one of those games for which the solution of the game "giveaway" is not much different from
the decision a "normal" game. (Incidentally, neem solution "giveaway" was given the same Charles Bouton,
who described the decision of the "normal" neem.)

How can we explain such a strange pattern - that the winning / losing neem "giveaway" coincides with the winning /
losing "normal" neem almost always?

Consider some **During the game**: le choose an arbitrary starting position players and write the moves until the end
of the game. It is easy to understand that provided optimal performance rivals - the game ends that remain one lot
size 1 And the player will be forced to go there and play.

Consequently, in any game two optimal players sooner or later **time** When the size of all non-empty heaps equal to one.
We denote the number of non-empty heaps at this point - then $\frac{1}{2}$ the current player for this position is a win-win if and only if
is even. le we have seen that in these cases the winning / losing neem "giveaway" **opposite** "Normal" nimu.

Again back to the time when the first time in the game all the piles size steel 1, And rolled one move ago - right before the
situation turned out. We were in a situation that has a size of one lot > 1 And all the other handful (maybe there were zero
pieces) - size 1. This position is obviously a win-win (because we really can always make such a move, so that there is an
odd number of heaps of size 1 le let an opponent

to defeat). On the other hand, XOR-sum of the sizes of heaps at this time is different from zero
- so here, "normal" it **coincides** Nîmes with "giveaway".

Further, if we continue to fall back on the game back, we will come to the point where the game was two piles size, three piles, etc. For each of these states the winning / losing will also coincide

c "Normal" Nîmes - simply because we have more than one heap size $> 1 > 1$

C .

Problem in online judges

[Difficulty: Easy]

- [Difficulty: Medium]
- SGU # 328 "[A Coloring Game](#)" [Difficulty: Medium]

Literature

- John Horton Conway. [On numbers and games](#) [1979]
- Bernhard von Stengel. [Lecture Notes on Game Theory](#)

Johnson's problem with one machine

This problem of optimal treatment schedules n components on a single machine, if i Th item is processed on it during t_i And for t seconds to wait before processing this part to pay fines $f_i(t)$.

Thus, the problem lies in finding such a rearrangement of parts that the next value (the fine) is minimal. If we denote by π rearrangement of parts (π_1 - Number of the first workpiece, π_2 - Second, etc.), the amount of the fine $F(\pi)$ is equal to:

$$F(\pi) = f_{\pi_1}(0) + f_{\pi_2}(t_{\pi_1}) + f_{\pi_3}(t_{\pi_1} + t_{\pi_2}) + \dots + f_{\pi_n}\left(\sum_{i=1}^{n-1} t_{\pi_i}\right).$$

Sometimes this problem is called uni serve multiple applications.

Solution of the problem in some particular cases

The first special case: a linear penalty function

Learn how to solve this problem in the case where all $f_i(t)$ linear, ie have the form:

$$f_i(t) = c_i \cdot t,$$

where c_i - Non-negative integers. Note that in these linear features free term is zero, because otherwise to answer immediately possible to add this intercept, and solve the problem with zero constant term.

We fix a schedule π' permutation π . We fix some number $i = 1 \dots n - 1$ And let the permutation is a permutation π , which traded the first and i th elements. Let's see how much has changed in this penalty:

$$F(\pi') - F(\pi) =$$

easy to understand that changes have occurred only with i Th and $i + 1$ Th terms:

$$\begin{aligned} &= c_{\pi'_i} \cdot \sum_{k=1}^{i-1} t_{\pi'_k} + c_{\pi'_{i+1}} \cdot \sum_{k=i}^i t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=i}^i t_{\pi_k} = \\ &= c_{\pi_{i+1}} \cdot \sum_{k=1}^{i-1} t_{\pi'_k} + c_{\pi_i} \cdot \sum_{k=1}^i t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=1}^i t_{\pi_k} = \\ &= c_{\pi_i} \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i}. \end{aligned}$$

Clearly, if the schedule π is optimal, then any change leads to an increase in the fine (or retain the previous value), so the optimal plan can write the condition:

$$\forall i = 1 \dots n - 1 : c_{\pi_i} \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i} \geq 0.$$

Rearranging, we obtain:

$$\forall i = 1 \dots n - 1 : \frac{c_{\pi_i}}{t_{\pi_i}} \geq \frac{c_{\pi_{i+1}}}{t_{\pi_{i+1}}}.$$

Thus, **optimal schedule** can be obtained by simply **sorting** all details with respect c_i to t_i in reverse order.

It should be noted that this algorithm we got the so-called **commutation technique**: we tried to swap the position of two adjacent elements of the schedule figured out how at this fine changed and hence derived algorithm for finding the optimal schedule.

The second special case: exponential functions fine

Suppose now that the penalty function are of the form:

$$f_i(t) = c_i \cdot e^{\alpha \cdot t},$$

where all the numbers c_i nonnegative constant α positive.

Then, using a similar way here commutes reception, easy to get parts that need to sort in descending order quantities:

$$v_i = \frac{1 - e^{\alpha \cdot t_i}}{c_i}.$$

The third special case: the same monotone functions fine

In this case it is assumed that all $f_i(t)$ coincide with some function $\phi(t)$, which is increasing. Clearly, in this case optimally positioned in order of the items of processing time t_i .

Theorem Livshits-Kladova

Theorem Livshits-Kladova establishes that commutes reception is only applicable for the above three special cases, and only them, ie:

- Linear case: $f_i(t) = c_i \cdot t + d_i$ - Non-negative constants
- Exponential case: $f_i(t) = c_i e^{\alpha \cdot t} + d_i$ Where c_i, α - are positive constants,
- Identical case: $f_i(t) = \phi(t)$ ϕ - Increasing function.

This theorem was proved under the assumption that the penalty function are sufficiently smooth (there are third derivatives).

In all three cases apply permutation technique whereby the desired optimal schedule can be found by simple sorting, therefore, for the time $O(n \log n)$.

Johnson's problem with two machines

There is n and two machine parts. Every detail must first be processed on the first machine, and then - on the second. In this i th item is processed on the first machine for a_i time, and the second - for the time b_i . Each tool at each time point can only operate on one part.

Required to make such an order on the part supply machines to the final processing time of all the details would be minimal.

This problem is sometimes called the problem of dual-processor maintenance tasks, or task Johnson (named after SM Johnson, who in 1954 proposed an algorithm to solve it).

It is worth noting that when the number of tools greater than two, the task is NP-complete (as demonstrated by Gary (Garey) 1976).

Construction of an algorithm

Note first that we can assume that the order of machining **the first and second machines must be the same**. In fact, because details for the second machine are available Only after the first treatment, and when there are several available for the second time their machine parts

processing is equal to the sum of their independently of their order - that is most advantageous to send a second machine that of the details that has been treated before others on the first machine.

Consider the procedure for submission of details on machines, which coincides with their input order: $1, 2, \dots, n$.

We denote x_i downtime the second machine directly before processing i Second Part (after treatment $i-1$ Second Part). Our goal - **minimize the total simple**:

$$F(x) = \sum x_i \rightarrow \min.$$

For the first part, we have:

For the second - because it becomes ready to be sent to the machine in a second time $a_1 + a_2$ And second Machine released at time $x_1 + b_1$, We have:

The third item becomes available at the time of the second machine $a_1 + a_2$ And the machine is released in $x_1 + b_1 + x_2 + b_2$ So:

Thus, the general form for x_i looks like this:

$$x_k = \max \left(\sum_{i=1}^k a_i - \sum_{i=1}^{k-1} b_i - \sum_{i=1}^{k-1} x_i, 0 \right).$$

Count now **simple summary** $F(x)$. Argues that it has the form:

$$F(x) = \max_{k=1 \dots n} K_i,$$

where

$$K_i = \sum_{i=1}^k a_i - \sum_{i=1}^{k-1} b_i.$$

(As it can be seen by induction, either sequentially finding expression for the sum of the first two, three, etc. x_i .)

We now use **commutation technique**: Try to swap any two adjacent elements j and $j+1$ and see how this will change in the total simple.

By type of function expressions for K_i clear that the only change K_j and K_{j+1} denote their new values through

K'_i and K'_{i+1} .

Thus, the item to j went to details $j + 1$. Sufficient (but not necessary) that:

$$\max(K_j, K_{j+1}) \leq \max(K'_j, K'_{j+1}).$$

(ie, we ignored the rest, is not changed, the arguments in the expression for the maximum, thus obtaining a sufficient but not necessary condition for the old $F(x)$ less than or equal to the new values)

Subtracting $\sum_{i=1}^{j+1} a_i - \sum_{i=1}^{j-1} b_i$ from both sides of this inequality, we get:

$$\max(-a_{j+1}, -b_j) \leq \max(-b_{j+1}, -a_j),$$

or getting rid of negative numbers, we get:

$$\min(a_j, b_{j+1}) \leq \min(b_j, a_{j+1}).$$

Thus, we have **comparator**: Sorting out the details on it, we, according to the above calculations, we obtain the optimal order of parts in which it is impossible to interchange any two parts, improving the final time.

However, one can further **simplify** sorting, if you look at this the other comparator side. In fact, he tells us that if a minimum of four numbers $(a_j, a_{j+1}, b_j, b_{j+1})$ achieved on an element of the array a , The corresponding item should go ahead, and if the element of the array - then later. Thus we obtain another form of the algorithm: sort the items of the minimum, and if (a_i, b_i) at least equal to the current part a_i , Then this item should be processed first of the remaining, otherwise - the last remaining.

Anyway, it turns out that the problem with two machines Johnson comes to sorting items with certain elements of the comparison function. Thus, the asymptotic behavior of the solutions is $O(n \log n)$.

Implementation

Implement the second option the algorithm described above, when the items are sorted by a minimum (a_i, b_i) And then go to the beginning or the end of the current list.

```

struct item {
    int a, b, id;
    bool operator < (item p) const {
        return min(a, b) < Min(p.a, p.b);
    }
};

sort (v.begin(), v.end());
vector <item> a, b;
for (int i = 0; i < n; ++ I)
    (v[i]. A <= v[i]. B? a: b). Push_back (v[i]);
a.insert (a.end(), B.rbegin(), B.rend());

int t1 = 0, t2 = 0;
for (int i = 0; i < n; ++ I) {
    t1 += a[i]. A;
    t2 = max(t2, t1) + A[i]. B;
}

```

Here all the details are stored in the form of structures **item**, Each of which contains values a and b original part number.

Details are sorted, then distributed to lists a (These are the details that were sent to the queue), and (those that were sent to the b end). Thereafter, the two lists are combined (wherein the second list is taken in reverse order), and then found the order of calculated required minimum time supported two variables t_1 and t_2 - The liberation of the first and second machine, respectively.

Literature

- S.M. Johnson. **Optimal two-and three-stage production schedules with setup**

times included [1954]

- M.R. Garey. **The Complexity of Flowshop and Jobshop Scheduling** [1976]

Optimal selection of jobs in certain completion times and durations of execution

Suppose we are given a set of tasks, each task is known point in time to which this task to complete, and the duration of this job. The process of performing a task can not be interrupted before completion. Required to make a schedule to perform the greatest number of jobs.

Decision

An algorithm for solving - **greedy** (Greedy). Sort all tasks by their deadline, and we will consider them one by one in order of deadline. We create all q in which we will gradually put the job from the queue and retrieve the job with the smallest execution time (for example, you can use the set or priority_queue). Initially q empty.

Suppose we consider i th job. Please put it in q . Consider the time interval between the deadline for completion i on the job and for completion $i - 1$ on the job - a segment of a certain length T .

Will be removed from the q job (in increasing order of the remaining time of their implementation) and put on a performance in this segment, until it fills the entire segment T . Important point - if at some time next extracted from the structure of the job can be done in time to partially segment T We perform this task partially - is as far as possible, i.e. for T time units, and the remainder of the job is placed back in q .

At the end of this algorithm, we will choose the optimal solution (or at least one of several solutions). Asymptotics of solutions - $O(n \log n)$.

Implementation

```
int n;
vector<pair<int,int>> A; // assignment as pairs (Deadline duration)
N reading ... and a ...

sort (a.begin(), A.end());

typedef set<pair<int,int>> T_s;
t_s s;
vector<int> Result;
for (int i = n-1; i >= 0; - I) {
    int t = a[i].First - (i? a[i-1].First: 0);
    s.insert (make_pair (a[i].Second, i));
    while (t && ! s.empty()) { t_s :: iterator it = s.begin(); if
        (it-> first <= t) {
            t -= it-> first;
            result.push_back (it-> second);
        }
        else {
            s.insert (make_pair (it-> first - t, it-> second)); t = 0;
        }
        s.erase (it);
    }
}
for (size_t i = 0; i < result.size(); +
    + I) cout << Result[i] << ' ';
```

Josephus problem

Condition of the problem. Given natural n and k . Discharged in a circle all the natural numbers from 1 to n . Please tick the i -th, starting with the first, and remove it. Then, it counts the number of the i -th removed, etc. The process is stopped when there is a single number. Want to find this number.

The problem was posed **Josephus** (Flavius Josephus) in the 1st century (although in a somewhat narrower formulation: the $k = 2$).

To solve this problem can be simulated. The simplest modeling work. Using $O(n^2)$ Segment tree, can be made for modeling $O(n \log n)$.

Solution for $O(n)$

Try to find a pattern that expresses the answer to the problem $J_{n,k}$ through the preceding problem.

Using modeling construct a table of values, for example, this:

$n \setminus k$	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	2	1	2	1	2	1	2	1	2	1
3	3	3	2	2	1	1	3	3	2	2
4	4	1	1	2	2	3	2	3	3	4
5	5	3	4	1	2	4	4	1	2	4
6	6	5	1	5	1	4	5	3	5	2
7	7	7	4	2	6	3	5	4	7	5
8	8	1	7	6	3	1	4	4	8	7
9	9	3	1	1	8	7	2	3	8	8
10	10	5	4	5	3	3	9	1	7	8

And here is quite clearly visible next **regularity**:

$$J_{n,k} = (J_{(n-1),k} + k - 1) \% n + 1$$
$$J_{1,k} = 1$$

Here 1-indexing formula spoils elegance if numbered positions from scratch, you get a very apt formula:

$$J_{n,k} = (J_{(n-1),k} + k) \% n = \sum_{i=1}^n k \% i$$

So we found a solution to the problem of Joseph, working for $O(n)$ operations.

Simple **recursive implementation** (1-indexing):

```
int joseph (int n, int k) {
    return n>1 ? (joseph (n-1, K) + K - 1) % N + 1 : 1;
}
```

Nonrecursive form:

```
int joseph (int n, int k) {
    int res = 0;
    for (int i =1; i <= n; ++
        I) res = (res + k) %
        I;
    return res + 1;
}
```

Solution for $O(k \log n)$

For relatively small, you can come up with a better solution than the above-considered recursive solution for. It's small, even intuitive that an algorithm makes a lot of unnecessary actions: significant changes occur only when there is the modulus n . But until then the algorithm simply adds a few times to account number. Accordingly, we can get rid of these unnecessary steps,

Partly arising in this case the difficulty lies in the fact that after the removal of these numbers, we obtain the problem with a smaller n . But not the starting position in the first number, and elsewhere. Therefore, recursively calling itself the task of the new n . Then we must be careful to translate the result in our numbering system of its own.

Also need to separately analyze the case when n will be less - in this case the above optimization degenerate into an infinite loop.

Implementation (For convenience 0-indexed)

```
int joseph (int n, int k) {
    if (n == 1)    return 0;
    if (k == 1)    return n-1;
    if (k > n)   return (joseph (n-1, K) + K) % N;
    int cnt = N / k;
    int res = Joseph (n - cnt, k);
    res -= n % k;
    if (res < 0)   res += n;
        res += res / (k -
    else 1);
    return res;
}
```

We estimate the **asymptotics** This algorithm. Immediately, we note

that the case

$n < k$ we disassembled the old solution that will work in this case for $O(k)$. Now consider the algorithm itself. In fact, each its iteration instead n numbers, we get about $n(1 - \frac{1}{k})$ number, so the total number of iterations algorithm can be found approximately from the equation:

$$n \left(1 - \frac{1}{k}\right)^x = 1,$$

the logarithm of this, we obtain:

$$\ln n + x \ln \left(1 - \frac{1}{k}\right) = 0,$$
$$x = -\frac{\ln n}{\ln \left(1 - \frac{1}{k}\right)},$$

using the expansion of the logarithm in a Taylor series, we obtain a rough estimate:

$$x \approx k \ln n$$

Thus, the asymptotic behavior of the algorithm really $O(k \log n)$.

Analytical solution for $k = 2$

In this particular case (in which this problem was posed by Josephus) problem can be solved much easier.

In the case of even n get that will be struck by all the even number, and then left for the task, then the answer to n will be obtained from the response to $\frac{n}{2}$ multiplying by two and subtracting one (due to the shift position):

$$J_{2n,2} = 2J_{n,2} - 1$$

Similarly, in case of odd n will be struck by all the even number, then the first number, and will challenge for $\frac{n-1}{2}$. And taking into account second shift positions obtain formula:

$$J_{2n+1,2} = 2J_{n,2} + 1$$

When implementing this can be used directly recursive dependency. This pattern can be transformed into

another form: represent the sequence of odd $J_{n,2}$

Numbers "restarts" a unit whenever n is a power of two. This can be written in the form of a formula:

$$J_{n,2} = 1 + 2 \left(n - 2^{\lfloor \log_2 n \rfloor} \right)$$

Analytical solution for $k > 2$

Despite the simple form of the problem and a large number of articles on this and related problems, a simple analytic representation of the solution of the problem of Joseph is still not found. For smaller k derived some formulas, but, apparently, they are hardly usable in practice (for example, see Halbeisen, Hungerbuhler "The Josephus Problem" and Odlyzko, Wilf "Functional iteration and the Josephus problem").

Game Fifteen: the existence of solutions

Recall that the game is a field 4 on 4On which are arranged 15 chips, numbered from 1 to 15And one field is left blank.
Required at each step moving any chip on the loose
position in the end come to the following positions:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	○

Fifteen game ("15 puzzle") was invented in 1880 Noyes Chapman (Noyes Chapman).

Existence of solutions

Here we consider the following problem: for a given position on the board to say whether there is a sequence of moves leading to a decision or not.

Let some position on the board:

a_1	a_2	a_3	a_4
a_5	a_6	a_7	a_8
a_9	a_{10}	a_{11}	a_{12}
a_{13}	a_{14}	a_{15}	a_{16}

wherein one of the elements is equal to zero and represents the

empty cage $a_z = 0$. Consider a permutation:

(ie a permutation corresponding position on the board, without the zero element)

We denote N the number of inversions in the permutation (ie, the number of such elements a_i and that $i < j$, but $a_i > a_j$).

Further, let K - The line number in which there is an empty element (ie, in our notation $K = (z - 1) \text{ div } 4 + 1$).

Then, **solution exists if and only if when $N + K$ even.**

Implementation

We illustrate the above algorithm using code:

```
int a[16];
for (int i = 0; i < 16; +
    + I) cin >>
    A[i];

int inv = 0;
for (int i = 0; i + +
    < 16; I)
    if (a[i]) (int j = 0; j < i; + +
        for J)
            if (a[j] > A[i])
for (int i = 0; i + +
    < 16; I)
    if (a[i] == 0)
        +
        inv += I / 4;
```

```
puts ((inv & 1) ? "No Solution" : "Solution Exists");
```

Proof

Johnson (Johnson) in 1879 proved that if $N + K$ odd, then there is no solution, and Storey (Story) in the same he proved that all items for which $N + K$ even have a solution.

However, both of these proofs were quite complex.

In 1999, Archer (Archer) suggested a much simpler proof (article can download it [here](#)).

Stern-Brocot tree. Farey series

Stern-Brocot tree

Stern-Brocot tree - it's elegant design, allowing you to build the set of all non-negative fractions. It was independently discovered by the German mathematician Maurice Stern (Moritz Stern) in 1858 and French watchmaker Achilles Brokaw (Achille Brocot) in 1861, however, according to some sources, this design has been opened ancient Greek scholar Eratosthenes (Eratosthenes).

On **zero** iteration we have two fractions:

$$\frac{0}{1}, \frac{1}{0}$$

(Second value, strictly speaking, not a shot, it can be understood as an irreducible fraction, indicating infinity)

Then, each **follow** iteration takes this list and fractions between neighboring fractions $\frac{a}{b}$ and $\frac{c}{d}$ inserted them **mediantle** fraction $\frac{a+c}{b+d}$.

So, on the first iteration the current set is:

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

The second:

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

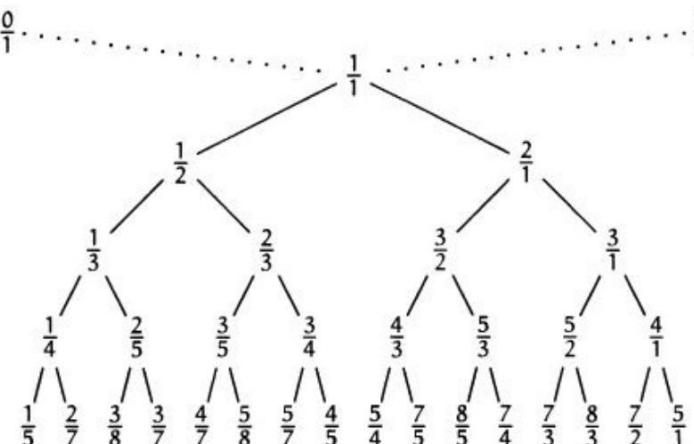
On the third:

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

Continuing this process to **infinityls** approved, you can get a lot of **all** non-negative fractions. Moreover, all the fractions are obtained **different** (i.e., the current set of each fraction occurs not more than once), **irreducible** (Numerators and denominators will be obtained relatively prime). Finally, all fractions will be automatically **ordered** ascending. The proof of all these remarkable properties of wood-Stern Brokaw will be described below.

It remains only to bring the image of the tree Stern-Brokaw (as we have described it with the changing of the set). At the

root of this tree is infinite fraction $\frac{1}{1}$. And the left and right of the tree and fractions are $\frac{0}{1}$ and $\frac{1}{0}$. Any tree node has two sons, each of which is obtained as the mediant of its left and right ancestor ancestor:



Proof

Order. It proved very simple: note that the mediant of two fractions is always

between them, i.e.:

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$$

provided that

$$\frac{a}{b} \leq \frac{c}{d}$$

This can be proved simply by bringing the three fractions to a common denominator.

Since zero-order iteration took place, it will be stored and each new iteration.

Irreducibility. For this show, that at each iteration, for any two adjacent fractions in the list and $\frac{c}{d}, \frac{a}{b}$ performed:

Indeed, recalling Diophantine equations with two unknowns ($ax + by = c$), obtain from this assertion, which $\gcd(a, b) = \gcd(c, d) = 1$ That we required.

So, we have to prove the truth of statements $bc - ad = 1$ for every iteration. We prove it by induction also. At zero iteration it is run (as is easily seen). Now let it was made at the previous iteration, we show that it holds at the current iteration. For this we need to consider three fractions neighbors in the new list:

$$\frac{a}{b}, \frac{a+c}{b+d}, \frac{c}{d}$$

For them, the conditions take the form:

However, the truth of these conditions is obvious, provided the $ad = 1$. Thus, actually, this property holds at the current iteration, as required.

The presence of all fractions. The proof of this property is closely related to the algorithm of finding the fraction in Stern-Brocot tree. Given that the Stern-Brocot tree all fractions in order, we find that for every vertex of the tree in the left subtree are a fraction smaller than her, and the right - its great. Hence we get the obvious and search algorithm in a fraction of Stern-Brocot tree: first, we are at the root; compare our fraction and a fraction recorded in the current node, if the fraction is less than ours, then go to the left subtree if our shot more - go to the right, and if the same - found shot, the search is completed.

To prove that an infinite tree Stern-Brokaw contains all the fractions, it suffices to show that this search algorithm fraction completed in a finite number of steps for any given shot. This algorithm can be

understood as follows: we have the current segment $[\frac{a}{b}; \frac{c}{d}]$ in which we are looking for our shot. Initially $\frac{a}{b} = \frac{0}{1}$, $\frac{c}{d} = \frac{1}{0}$. At each step, a fraction $\frac{x}{y}$ compared with MEDIAN endpoints, ie with $\frac{a+c}{b+d}$. And depending on this, we or stop the search, or go to the left or right part of the segment. If the search algorithm fraction worked indefinitely, the following conditions have been fulfilled for each iteration:

$$\frac{a}{b} < \frac{x}{y} < \frac{c}{d}$$

But they can be rewritten in the following form:

$$bx - ay \geq 1, \\ cy - dx \geq 1$$

(Used here is that they are integral, and therefore from > 0 should ≥ 1) Then, multiplying

the first to $c+d$ And the second - on the $a+b$ And adding them, we obtain:

The brackets on the left and the fact that $ad = 1$ (See the proof of the previous properties) we finally obtain:

Since at each iteration is at least one of the variables a, b, c, d strictly increasing, the process of finding fractions $\frac{x}{y}$ will not contain more than $x+y$ iterations, as required.

An algorithm for constructing the tree

To build any subtree of the Stern-Brokaw enough to know only the left and right ancestors. Initially, at the first level, is the ancestor of the left and the right -. They can be used to calculate the fraction in $\frac{1}{0}$ current node, and then start from the left and right sons (left son passing itself as an ancestor of the right and the right son - as a left ancestor).

Pseudocode of this procedure, all trying to build an infinite tree:

```
void build (int a=0,int b = 1,int c = 1,int d = 0,int level = 1) {
    int x = a + c, y = b + d;
    Conclusion ... current fraction x / y at the tree level
    build (a, b, x, y, level + 1);
    build (x, y, c, d, level + 1);
}
```

Search algorithm fraction

Search algorithm fraction has already been described in the proof of the fact that the Stern-Brocot tree contains all the fractions, we repeat it here. This algorithm - in fact binary search algorithm, or ability to set value in a binary search tree. Initially, we are at the root of the tree. Standing in the current node, we compare our fraction and a fraction in the current node. If they match, then the process stops - we found shot in the tree. Otherwise, if the fraction is less than our current fraction in the top, then go to the left child, otherwise - to the right.

As was shown in the property that the Stern-Brocot tree contains all non-negative fractions, fractions when searching $\frac{x}{y}$ algorithm will make no more $x + y$ iterations.

We present an implementation which returns the path to the top, containing a given fraction, returning it to as a sequence of characters 'L' / 'R': if the current character is 'L', it marks the transition to a tree in the left child, and otherwise - in the right (initially we are at the root of the tree, ie at the top of shot). In fact, $\frac{1}{1}$ a sequence of characters, creatures, and uniquely identifying any non-negative fraction, called **Stern-value system Brokaw**.

```
string find (int x, int y, int a = 0,int b = 1,int c = 1,int d = 0) {
    int m = a + c, n = b + d;
    if (x == m && y == n)
        return "";
    if (x * n < y * m)
        return 'L' + Find (x, y, a, b, m, n);
    else
        return 'R' + Find (x, y, m, n, c, d);
}
```

Irrational number in a Stern-Brokaw will meet endless sequence of characters; if you know any preassigned accuracy, we can restrict some prefix of this infinite sequence. During this endless search irrational fractions in Stern-tree algorithm Brokaw each time will find a simple fraction (with gradually increasing denominators), which provides a better approximation of irrational numbers (just use this important time in the art and therefore Achilles Brokaw and discovered this tree).

Farey sequences

Farey sequence of order n is the set of all irreducible fractions between 0 and 1 whose denominators do not exceed n . And fractions are arranged in ascending order.

This sequence is named after the English geologist John Farey (John Farey), who tried in 1816 to prove that in any number of Farey mediant fraction is two adjacent. It is known that it proof was wrong and correct proof offered later Cauchy (Cauchy). However, back in 1802 mathematician Haros (Haros) in one of his works came to almost the same results.

Farey sequence and have their own set of interesting properties, but the most obvious of their **connection with the Stern-Brocot tree**: Actually, Farey sequence is obtained by removing some branches of the tree. Or we can say that for Farey sequence need

take the set of fractions obtained in the construction of wood-Stern Brokaw on infinite iteration, and leave this set only fractions with denominators not exceeding n and numerators, denominators not exceeding.

Of the algorithm for constructing a tree-Stern Brokaw should and similar **algorithm** for Farey sequences.

At zero iteration include in the set and only shot. At each subsequent iteration we between each two neighboring fractions insert their mediant, if the denominator does not exceed n . Sooner or later cease to be in the set of changes, and the process can be stopped - we found the required sequence Farey.

We compute **length** Farey sequence. Farey sequence of order n contains all Farey sequence of elements of order $n - 1$ And all irreducible fractions with denominators equal n But this quantity is known, power $\phi(n)$ Thus, the length L_n Farey sequence of order n expressed by the formula:

$$L_n = L_{n-1} + \phi(n)$$

or revealing recursion:

$$L_n = 1 + \sum_{k=1}^n \phi(k)$$

Literature

- Ronald Graham, Donald Knuth, and Oren Patashnik. **Concrete Mathematics. Base Informatics** [1998]

Search subsegments array with a maximum / minimum amount

Here we consider the problem of finding subsegments array with a maximum amount ("maximum subarray problem" in English), as well as some of its variations (including a variant of this algorithm for solving the problem in the online

- A self-described algorithm - KADR (Yaroslav Tverdohleb)).

Statement of the Problem

Given an array of numbers $a[1 \dots n]$. It is required to find such a subsegment $a[l \dots r]$ Amount thereon **maximum**:

$$\max_{1 \leq l \leq r \leq n} \sum_{i=l}^r a[i].$$

For example, if all the numbers of the array would be non-negative, the response could take the entire array. Nontrivial solution when the array can contain both positive and negative numbers.

It is clear that the problem of finding **minimum** subsegments - essentially the same, you just change the signs of all the numbers on the opposite.

Algorithm 1

Here we consider the almost obvious algorithm. (Next we consider another algorithm, which is slightly more difficult to come up with, but its implementation is obtained even shorter.)

Description of the algorithm

The algorithm is quite simple.

For convenience we introduce **designation**: $\sum_{j=0}^i s[j]$ - An array of partial sums array $a[]$. Also set value $s[0] = 0$.

We now **sort out** index $r = 1 \dots n$ And learn the current value for each r quickly find the optimum l , At which the maximum amount on the subsegments $[l; r]$

Formally, this means that we need for the current r find a (not exceeding r) Value to a maximum. After trivial transformations we see that we need

find the minimum value in an array $s[]$ minimum interval $[0; r - 1]$.

Hence we immediately obtain the solution algorithm: we simply store where in the array $s[]$ is current minimum. Using this minimum, we are for $O(1)$ find the best code for the current and at the transition from the current index to the next, we just update this minimum.

Obviously, this algorithm runs in $O(n)$ and asymptotically optimal.

Implementation

For implementation we do not even need to explicitly store an array of partial sums - from it we will required only the current item.

The implementation is given in 0-indexed arrays, but not in 1-numbering as described above. We first present a solution

that is simple numerical answer without finding the desired segment codes:

```
int ans = a[0],  
    sum = 0,  
    min_sum = 0;  
for (int r = 0; r < n; ++  
R) { sum +=  
    a[r];  
    ans = max (ans, sum - min_sum);  
    min_sum = min (min_sum, sum);  
}
```

We now give the full version of the solution, which in parallel with the numerical solution of the boundary is the desired interval:

```
int ans = a[0], ans_l
    = 0, Ans_r =
0, Sum = 0,
Min_sum = 0,
Min_pos = -
1;

for (int r = 0; r < n; + +
R) { sum + =
a[r];

int cur = sum -
min_sum; if (cur > ans)
{
    ans = cur;
    ans_l = min_pos +
1; ans_r = r;
}

if (sum < min_sum) {
    min_sum = sum;
    min_pos = r;
}
}
```

Algorithm 2

Here we consider a different algorithm. It's a bit more complicated to understand, but it is more elegant than the above, and realized just a little shorter. This algorithm was proposed by Jay Kadane (Jay Kadane) in 1984

Description of the algorithm

Itself **algorithm** as follows. We will go through the array and store in a variable s current partial sum. If at any time s is negative, then we simply assign $s = 0$. It is alleged that the maximum of all values of the variable s that occurred during the work, and will answer to the problem.

We prove This algorithm.

In fact, consider the first point in time when the amount s becomes negative. This means that, starting from a zero partial sum, we come up to the negative partial sum - hence, the entire array of the prefix, as well as its suffix have any negative amount. Consequently, all of the prefix array in the future can not be any good, he can only give a boost to a negative answer. However, this is not enough to prove the algorithm. In the algorithm we actually restrict to answering only those segments that begin immediately after the places where cases $s < 0$.

But, in fact, consider an arbitrary interval $[l; r]$. Wherein l is not in a "critical" positions (ie is $l > p + 1$ Where - the last such position, in which $s < 0$). Since the last critical position is strictly earlier than $p + 1$ It turns out that the sum of $[p + 1 \dots l - 1]$ nonnegative. This means that the moving l position $p + 1$ We will increase the response or, in extreme cases, do not change it. Anyway, but it turns out that when you search for the answer really can restrict the segments, starting immediately after the position in which exerted $s < 0$. This proves the correctness of the algorithm.

Implementation

As in Algorithm 1, we present first a simplified implementation that searches only the numeric response, finding the boundaries of the desired interval:

```

int ans = a[0],
    sum = 0;
for (int r = 0; r <n; + +
R) { sum + =
a[r];
ans = max (ans,
sum); sum = max
(sum, 0);
}

```

Full version of the solution, maintaining the boundaries of the desired index-segment:

```

int ans = a[0],
    ans_l = 0,
    Ans_r = 0,
    Sum = 0,
    minus_pos = -1;for
(int r = 0; r <n; + + R) {
    sum + = a[r];

    if (sum > ans) { ans
        = sum;
        ans_l = minus_pos + 1;
        ans_r = r;
    }

    if (sum < 0) {
        sum = 0;
        minus_pos = r;
    }
}

```

Related tasks

Search the maximum / minimum subsegments with restrictions

If in the problem on the desired segment $[l; r]$ additional restrictions (for example, that length $r - l + 1$ segment must be within predetermined limits) then this algorithm are likely easily generalized to these cases - either way, the problem will continue to be to find a minimum in the array $s[]$ when given additional restrictions.

The two-dimensional case of the problem: find the maximum / minimum submatrix

Described in this article the problem naturally generalizes to higher dimensions. For example, in the case of a two-dimensional it turns into a search for a submatrix predetermined matrix, which has the maximum amount of numbers in it. $[l_1 \dots r_1; l_2 \dots r_2]$

From the above described solutions for the one **easy to get** solution for and calculate sums with an array on each row of the matrix; we came to the one-dimensional problem of finding indices $l_1 \dots r_1$ in the array, which can already be solved in linear time.

Faster algorithms for solving this problem although known, however, they are not very fast with very complex (so complex that hidden constant many of them are inferior to the trivial algorithm for any reasonable restrictions). Apparently, the best known algorithms running

for $O\left(n^3 \frac{\log^3 \log n}{\log^2 n}\right)$ (T. Chan 2007 "More algorithms for all-pairs shortest paths in weighted graphs").

This algorithm Chan, and others in the art results in fact describe **rapid multiplication** matrices (Where the multiplication of matrices implied modified multiplication instead of addition uses a minimum, and instead of multiplying - Addition). The fact that the problem of

finding submatrices with the largest sum is reduced to the problem of finding the shortest paths between all pairs of vertices, and the task in turn - is reduced to the matrix multiplication.

Search subsegments with maximum / minimum average amount

This problem lies in the fact that it is necessary to find an interval $[l; r]$. The average of the maximum it was:

$$\max_{l \leq r} \frac{1}{r - l + 1} \sum_{i=l}^{r-1} ra[i].$$

Of course, if desired by the segment condition not imposed other conditions, the solution will always be a segment of length 1 at the maximum-array. Problem makes sense only if there are **additional restrictions** (Eg, the length of the desired interval is bounded below).

In such a case applicable **the standard method** when dealing with the problems of the average value: will select the desired maximum average value **binary search**.

To do this, we need to learn how to solve this subproblem: given by \bar{x} And you should check whether the array subsegment $a[l:r]$ (Of course, additional satisfying all constraints of the problem), in which the mean value is greater than \bar{x} .

To solve this subproblem, subtract \bar{x} of each array element. Then our subproblem actually turns into this: whether or not in the array subsegment positive amount. And this problem we can solve.

So we got the solution for asymptotic behavior $O(T(n) \log W)$ Wherein W - Required accuracy, time solutions for an array of length subtasks n (Which may vary depending on the specific additional constraints imposed).

Solution of the problem online

Condition of the problem is: given an array of n Numbers and given by L . Receives requests form, and in response to a request (l, r) requires a subsegment of the segment $[l; r]$ length at least L with the maximum possible arithmetic mean.

An algorithm for solving this problem is quite complicated. Author of this algorithm - KADR (Yaroslav Tverdohleb) - [the algorithm described in his message on the forum](#).

Literature

+ C +

Visual C++, MFC

- Kruglinski, Uingou, Shepherd. **Programming Microsoft Visual C++ 6.0 for professionals** (PDF [in RAR], 54.4 MB)

+ C +

- ANSI. **C++ International Standard (second edition, 2003-10-15)** (PDF, 2.3 MB)
- Eckel. **Philosophy C++. Introduction Standard C++ (2nd ed.)** (DJVU, 6.4 MB)
- Eckel, Allison. **Philosophy C++. Practical Programming** (DJVU, 6.5 MB)
- Sutter. **Complex tasks on C++. 87 puzzling examples decisions** (DJVU, 3.8 MB)
- Sutter. **New challenges for C++. 40 new puzzling examples with solutions** (DJVU, 3.6 MB)
- Stroustrup. **Programming language C++ (2nd, enlarged ed.)** (PDF, 2.9 MB)
- Stroustrup. **The C++ Programming Language (3rd edition)** (PDF, 3.4 MB)
- Abrahams, Gurtovoy. **C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond** (CHM, 0.62 MB)
- Dzhosyutis. **C++. Standard Library. For professionals** (DJVU, 4.8 MB)
- Dzhosyutis. **C++. Standard Library. For professionals (CD the book)** (ZIP, 0.14 MB)
- Vandervoorde, Josuttis. **C++ Templates: The Complete Guide** (CHM, 0.72 MB)
- Sutter, Alexandrescu. **C++ Coding Standards: 101 Rules, Guidelines, and Best Practices** (CHM, 0.51 MB)
- Golub. **Rope sufficient length to shoot yourself in the foot. Terms of programming in C and C++** (PDF, 1.29 MB)
- Meyers. **Effective C++. More effective C++** (CHM, 2.0 MB)
- Dyuherst. **Slippery C++. How to avoid problems when designing and compile your programs** (DJVU, 9.3 MB)
- Dyuherst. **C++. The sacred knowledge** (DJVU, 6.7 MB)

Algorithms

Fundamental benefits

- Cormen, Leiserson, Rivest, Stein. **Algorithms. Design and analysis (2nd bldg.)** (DJVU, 18.3 MB)
- Knuth. **The Art of Computer Programming. Volume 1 (3rd edition)** (DJVU, 6.0 MB)
- Knuth. **The Art of Computer Programming. Volume 2 (3rd edition)** (DJVU, 7.6 MB)
- Knuth. **The Art of Computer Programming. Volume 3 (2nd edition)** (DJVU, 7.7 MB)
- Cormen, Leiserson, Rivest, Stein. **Algorithms. Design and analysis (1st ed.?)** (PDF, 4.5 MB)
- Sedgwick. **Fundamental Algorithms (3rd ed.). Parts 1-4** (DJVU, 15.0 MB)
- Sedgwick. **Fundamental Algorithms (3rd ed.). Part 5** (DJVU, 16.7 MB)
- Knut. **Art of Computer Programming. Tom 1** (DJVU, 5.6 MB)
- Knut. **Art of Computer Programming. Tom 2** (DJVU, 6.1 MB)
- Knut. **Art of Computer Programming. Tom 3** (DJVU, 6.4 MB)
- Graham, Knuth, Patashnik. **Concrete Mathematics** (DJVU, 8.9 MB)
- Papadimitriou, Stayglits. **Combinatorial Optimization: Algorithms and Complexity** (DJVU, 5.6 MB)
- Motwani, Raghavan. **Randomized Algorithms** (DJVU, 4.4 MB)
- Tucker. **Computer Science Handbook** (PDF, 27.0 MB)
- Mehlhorn, Sanders. **Algorithms and Data Structures: The Basic Toolbox** (PDF, 2.0 MB)

Olympiad problems

- Menshikov. **Olympiad programming tasks** (DJVU, 4.4 MB)

- Menshikov. **Olympiad programming tasks (CD the book)** (ZIP, 4.0 MB)
- Okulov. **Programming algorithms** (DJVU, 3.6 MB)
- Dolinsky. **Addressing the complex and Programming Olympiad problems** (DJVU, 2.9 MB)
- Skiena, Revile. **Olympiad programming tasks** (DJVU, 5.3 MB)

Line

- Gasfield. **Lines trees and sequence algorithms** (DJVU, 12.1 MB)
- Smyth. **Computing patterns in strings** (DJVU, 26.4 MB)
- Crochemore, Rytter. **Jewels of Stringology** (DJVU, 2.6 MB)
- Crochemore, Hancart. **Automata for matching patterns** (PDF, 0.44 MB)

Compilation, interpretation

- Aho, Lam, Sethi, Ullman. **Compilers: Principles, Techniques and Tools** (DJVU, 5.7 MB)
- Mogensen. **Basics of Compiler Design** (PDF, 0.81 MB)
- Pratt Zelkovits. **Programming Languages: development and implementation (4th ed. 2002)** (DJVU, 5.7 MB)

Game theory

- Conway. **On Numbers and Games** (DJVU, 2.1 MB)

Algebra, Number Theory

- Ribenboim. **The New Book of Prime Number Records** (DJVU, 11.0 MB)
- Shoup. **A Computational Introduction to Number Theory and Algebra (version 2)** (PDF, 3.5 MB)
- William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. **Numerical Recipes: The Art of Scientific Computing** (PDF, 20.4 MB)

Computational Geometry

- Drug Shamos. **Computational Geometry. Introduction** (DJVU, 4.5 MB)
- Andreeva, Egorov. **Computational geometry on the plane** (PDF, 0.61 MB)
- Mount. **Lecture notes for the course Computational Geometry** (PDF, 0.77 MB)
- de Berg, van Kreveld, Overmars, Schwarzkopf. **Computational Geometry: Algorithms and Applications (2nd, revised edition)** (DJVU, 3.7 MB)
- Chen. **Computational Geometry: Methods and Applications** (PDF, 1.14 MB)
- Starlings. **Delaunay triangulation and its application** (PDF, 2.5 MB)
- Miu. **Voronoi Diagrams: lecture slides** (PDF, 0.14 MB)
- Held. **Voronoi Diagram: slides** (PDF, 1.35 MB)

Counts

- Ahuja, Magnanti, Orlin. **Network flows** (DJVU, 13.8 MB)
- Priezzhev. **Dimer problem and the Kirchhoff theorem** (PDF, 1.18 MB)
- Thorup. **Unidirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time** (PPT, 1.10 MB)
- Eppstein. **Finding the K Shortest Paths** (PDF, 0.18 MB)
- Sokkalingham, Ahuja, Orlin. **Inverse Spanning Tree Problems: Formulations and Algorithms** (PDF, 0.07 MB)
- Ahuja, Orlin. **A Faster Algorithm for the Inverse Spanning Tree Problem** (PDF, 0.10 MB)
- Brander, Sinclair. **A Comparative Study of K-Shortest Path Algorithms** (PDF, 0.16 MB)
- Gabow. **An Efficient Implementation of Edmonds Maximum-Matching Algorithm** (PDF, 2.7 MB)
- Bender, Farach-Colton. **The LCA Problem Revisited** (PDF, 0.08 MB)
- Mayniga. **Optimization algorithms for networks and graphs** (DJVU, 4.0 MB)
- Mehlhorn, Uhrig. **The minimum cut algorithm of Stoer and Wagner** (PDF, 0.12 MB)
- Åre. **Graph Theory** (DJVU, 4.3 MB)
- Harari. **Graph Theory** (DJVU, 8.7 MB)
- Stoer, Wagner. **A Simple Min-Cut Algorithm** (PDF, 0.20 MB)
- Lovasz, Plummer. **Matching theory** (PDF, 9.9 MB)
- Tutte. **The Factorization of Linear Graphs** (PDF, 0.47 MB)

Combinatorics

- Stepanov. **Burnside lemma and the problem of coloring** (PDF, 0.18 MB)

- Harari. [Enumerating graphs](#) (DJVU, 4.1 MB)

Complexity Theory

- Gary Johnson. [Computers and intractability of the problem](#) (DJVU, 11.5 MB)

Mathematics

- Aitken. [Determinants and Matrices](#) (PDF, 10.2 MB)

Data Structures

- Tarjan. [Efficiency of a Good But Not Linear Set Union Algorithm](#) (PDF, 0.63 MB)
 - Tarjan, Leeuwen. [Worst-Case Analysis of Set Union Algorithms](#) (PDF, 1.55 MB)
-

Optimization

- Kaspersky. [Code Optimization: Effective Memory Usage](#) (CHM, 10.4 MB)
 - Kaspersky. [Code Optimization: Effective Memory Usage \(CD the book\)](#) (ZIP, 4.6 MB)
 - Fog. [Optimization Manuals \(Optimizing software in C + +, in assembly, processors microarchitecture\) \(last edited - 2008\)](#) (PDF [in ZIP], 2.9 MB)
 - Intel. [Intel Architecture Optimization Manual \(1997\)](#) (PDF, 0.49 MB)
-

Java

Java

- Eckel. [Philosophy Java \(4th ed.\)](#) (DJVU, 5.4 MB)
 - Horstmann, Cornell. [Java 2. Library professional. Tom 1 \(The Basics\)](#) (7th ed.) (DJVU, 10.5 MB)
 - Horstmann, Cornell. [Java 2. Library professional. Tom 2 \(Subtleties programming\)](#) (7th ed.) (DJVU, 13.2 MB)
 - Horstmann, Cornell. [Java 2. Library professionals \(7th ed.\) \(CD the book\)](#) (ZIP, 0.66 MB)
-

TeX

TeX

- Knut. [All about TeX](#) (DJVU, 17.1 MB)
- Abrahams, Hargreaves, Berry. [TeX for the Impatient](#) (PDF, 1.36 MB)

LaTeX

- Gratzer. [Math into LaTeX. An Introduction to LaTeX and AMS-LaTeX](#) (DJVU, 0.34 MB)
- Oetiker. [The Not So Short Introduction to LaTeX \(version 4.26, 2008-Sep-25\)](#) (PDF, 2.3 MB)

About the Author

Permanent site author e-maxx.ru and all articles on algorithms I am, e-maxx, also known as Maxim Ivanov :)



All materials, including this book, laid out under license **Public Domain** in the public domain, not copyrighted and can be distributed is unlimited. The main objective of this resource - the dissemination of information and the elimination of obstacles to this.

Except for me in creating, updating and improving articles attended many people - thank you for the countless comments, suggestions for improvement, pointing out inaccuracies, etc. Unfortunately, none of the community did not dare to write a full article describing any algorithm. If you wish to be the first - welcome ;)

You can contact me by email: e-maxx@inbox.ru Or by [Forum Site e-maxx](#).