

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

## algo

Here is a 145 algorithms. To all algorithms and provides a brief description of the program in C ++.

---

Show: [last added](#) , [last edited](#) algorithms.

---

You can also download the [PDF-book](#) , in which all the articles collected in the form of a single large file.

---

## Algebra (23)

### basic algorithms (20)

- Euler function and its calculation [\[TeX\]](#)
- Binary exponentiation of O ( $\log N$ ) [\[TeX\]](#)
- Euclid's algorithm finding the GCD (greatest common divisor) [\[TeX\]](#)
- Sieve of Eratosthenes [\[TeX\]](#)
- Advanced Euclidean algorithm [\[TeX\]](#)
- Fibonacci numbers and their rapid calculation [\[TeX\]](#)
- The inverse of the ring modulo [\[TeX\]](#)
- Gray code [\[TeX\]](#)
- Long arithmetic [\[TeX\]](#)
- Discrete logarithm modulo M algorithm baby-step-giant-step Shanks for O ( $\sqrt{M} \log M$ ) [\[TeX\]](#)
- Diophantine equations with two unknowns:  $AX + BY = C$  [\[TeX\]](#)
- Modular linear equation of the first order:  $AX = B$  [\[TeX\]](#)
- Chinese remainder theorem. Garner's algorithm [\[TeX\]](#)
- Finding degree divider factorial [\[TeX\]](#)
- Balanced ternary notation [\[TeX\]](#)
- The factorial  $N!$  modulo P for O ( $P \log N$ ) [\[TeX\]](#)

- Enumeration of all subpatterns this mask. Grade  $3^N$  to the total number of all subpatterns masks [TeX]
- A primitive root. Algorithm for finding [TeX]
- Discrete root extract [TeX]
- Sieve of Eratosthenes with linear time work [TeX]

### complex algorithms (3)

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

## Graphs (51)

### basic algorithms (4)

- Breadth-first search [TeX]
- Dfs
- Topological sorting [TeX]
- Search connected components [TeX]

### strongly connected components, bridges, etc. (4)

- Search strongly connected component, condensation build a graph for  $O(N + M)$  [TeX]
- Search for bridges  $O(N + M)$  [TeX]
- Search articulation points for the  $O(N + M)$  [TeX]
- Search bridges online in  $O(1)$  an average of [TeX]

### the shortest path from one vertex (4)

- Dijkstra's algorithm finding the shortest paths from a given vertex to all other vertices of  $O(N^2 + M)$  [TeX]
- Dijkstra's algorithm for sparse graphs of finding the shortest paths from a given vertex to all other vertices of  $O(M \log N)$  [TeX]
- Bellman-Ford algorithm for finding the shortest paths from a given vertex to all other vertices of  $O(NM)$  [TeX]
- Leviticus algorithm of finding the shortest paths from a given vertex to all

other vertices of  $O(NM)$

## **shortest paths between all pairs of vertices (2)**

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

## **the minimum spanning tree (5)**

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

## **cycles (3)**

- Finding a negative cycle in the graph of  $O(NM)$  [TeX]
- Finding Euler path or cycle of  $O(M)$
- Checking on the graph is acyclic and finding cycle of  $O(M)$

## **lowest common ancestor (LCA) (5)**

- Lowest common ancestor. Finding of  $O(\sqrt{N})$  and  $O(\log N)$  with preprocessing  $O(N)$
- Lowest common ancestor. Finding of  $O(\log N)$  with preprocessing  $O(N \log N)$  (Binary lifting method)
- Lowest common ancestor. Finding the  $O(1)$  preprocessing with  $O(N)$  (algorithm Farah-Colton and Bender)
- The task RMQ (Range Minimum Query - at least in the interval). Solution in  $O(1)$  preprocessing  $O(N)$
- Lowest common ancestor. Finding the  $O(1)$  in offline mode (Tarjan algorithm) [TeX]

## **flows and related problems (10)**

- Edmonds-Karp algorithm for finding the maximum flow of  $O(NM^2)$
- Method push predpotoka finding the maximum flow of  $O(N^4)$

- Modification of the method of pushing predpotoka for  $O(N^3)$
- Flow restrictions
- The flow of minimum cost (min-cost-flow). Algorithm for ways to increase the  $O(N^3M)$
- Assignment problem. Solution with a min-cost-flow of  $O(N^5)$
- Assignment problem. Hungarian algorithm (algorithm Kuna) for  $O(N^3)$  [TeX]
- Finding the minimum cut algorithm for Curtains, Wagner  $O(N^3)$  [TeX]
- The flow of minimum cost, minimum-cost circulation. Algorithm for removing negative weight cycles [TeX]
- Dinic's algorithm for finding the maximum flow [TeX]

## **matchings and related problems (6)**

- Kuhn's algorithm for finding the greatest matchings of  $O(NM)$  [TeX]
- Checking on the graph bipartition and splitting into two parts for the  $O(M)$
- Finding the maximum weight vertex-weighted matchings of  $O(N^3)$
- Edmonds algorithm for finding maximum matching in an arbitrary graph of  $O(N^3)$  [TeX]
- Floor ways directed acyclic graph [TeX]
- Tutte matrix. Randomized algorithm for finding maximum matching in an arbitrary graph [TeX]

## **connection (3)**

- Edged connectivity. Properties and finding [TeX]
- Vertex connectivity. Properties and finding [TeX]
- Graphing with the specified vertex and rib connections and the lower of the degrees of the vertices [TeX]

## **By way-s (0)**

## **Inverse Problems (2)**

- The inverse problem of SSSP (inverse-SSSP - inverse problem of shortest paths from a single vertex) for  $O(M)$
- The inverse problem of MST (inverse-MST - inverse problem minimum spanning tree) for  $O(NM^2)$

## **Miscellaneous (3)**

- Paint the 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 [TeX]
- 

## Geometry (23)

### basic algorithms (10)

- The length of the union of the intervals on the line for  $O(N \log N)$
- Sign area of a triangle and the predicate 'Clockwise' [TeX]
- Checking on the intersection of two segments [TeX]
- Finding the equation of the line for the interval [TeX]
- Finding the point of intersection of two lines [TeX]
- Finding the intersection points of two segments [TeX]
- Finding the area of a simple polygon for  $O(N)$
- Pick theorem. Finding the square lattice polygon in  $O(1)$  [TeX]
- The problem of covering of points
- Centroids of polygons and polyhedra [TeX]

### more complex algorithms (13)

- 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 the area of combining triangles. Vertical decomposition method
- Checking points on the convex polygon belonging 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 method of compression sides of  $O(N \log N)$  [TeX]
- Voronoi diagram in two dimensions, its properties and application. The simplest algorithm for constructing of  $O(N^4)$  [TeX]
- Finding all the faces, the outer edge of a planar graph of  $O(N \log N)$  [TeX]
- Finding a pair of nearest points algorithm divide-and-rule for the  $O(N \log N)$  [TeX]
- Transformation of geometric inversion [TeX]

- Search common tangents to two circles [TeX]
  - Search pair of intersecting segments algorithm swept out by lines of O ( $N \log N$ ) [TeX]
- 

## String (12)

- Z-line function and its calculation of O ( $N$ ) [TeX]
  - Prefix function, its calculation and application. Algorithm Knuth-Morris-Pratt [TeX]
  - Hashing algorithms in problems on the line
  - Rabin-Karp algorithm search substring of O ( $N$ )
  - Expression parsing of O ( $N$ ). Reverse Polish Notation [TeX]
  - Suffix array. The construction of the O ( $N \log N$ ) and application [TeX]
  - Suffix machine. The construction of the O ( $N$ ) and application [TeX]
  - Finding all subpalindromes for O ( $N$ ) [TeX]
  - Lyndon decomposition. Duval algorithm. Finding the smallest cyclic shift of the O ( $N$ ) time and O ( $1$ ) memory [TeX]
  - Algorithm Aho-Korasik [TeX]
  - Suffix tree. Ukkonen's algorithm [TeX]
  - Search all tandem repeats in a string algorithm Maine-Lorentz (divide-and-conquer) for O ( $N \log N$ ) [TeX]
- 

## Data structures (7)

- Sqrt-decomposition [TeX]
  - Fenwick tree
  - The system of disjoint sets [TeX]
  - Segment tree [TeX]
  - Treap (treap, deramida)
  - Modification of the stack and queue for finding the minimum in O ( $1$ )
  - Randomized heap [TeX]
- 

## Algorithms on strings (3)

- The task RMQ (Range Minimum Query - at least in the interval)
  - Finding of the longest increasing subsequence of  $O(N^2)$  and  $O(N \log N)$  [TeX]
  - K-th order statistic of  $O(N)$
- 

## Dynamics (2)

- The dynamics of the profile. The problem of "parquet"
  - Finding the most zero submatrix of  $O(NM)$  [TeX]
- 

## Linear Algebra (3)

- Gauss solving a system of linear equations for the  $O(N^3)$  [TeX]
  - Finding the rank of a matrix of  $O(N^3)$
  - Calculating the determinant of a matrix by Gauss for the  $O(N^3)$
- 

## Numerical Methods (3)

- Integration over Simpson's formula [TeX]
  - Search roots by Newton's method (tangent) [TeX]
  - Ternary search [TeX]
- 

## Combinatorics (9)

- Binomial coefficients [TeX]
- Catalan numbers [TeX]
- Necklaces [TeX]
- The alignment of elephants on a chessboard
- Correct bracket sequence. Finding the lexicographically next, K-oh, caller ID [TeX]
- The number of labeled graphs, connected labeled graphs, labeled graphs with K connected components [TeX]
- Generation of combinations of N elements

- Burnside lemma. Polya theorem [TeX]
  - The principle of inclusion-exclusion [TeX]
- 

## Game Theory (2)

- Games on arbitrary graphs. The method of retrospective analysis of O (M)
  - Theory Shpraga Grande. Him [TeX]
- 

## Schedule (3)

- Johnson problem with one machine [TeX]
  - Johnson problem with two machines [TeX]
  - The optimal choice of job completion times for known and long-running [TeX]
- 

## Miscellaneous (4)

- Joseph task [TeX]
- Game Fifteen: the existence of solutions [TeX]
- Wood Stern-Brock. Farey sequence [TeX]
- Search subsegment array with maximum / minimum amount of O (N) [TeX]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 10:57  
EDIT: 18 Oct 2011 20:20

## Euler function

### Determination

**Euler function**  $\phi(n)$  (sometimes labeled  $\varphi(n)$  or  $\text{phi}(n)$ ) - the number of numbers 1 to  $n$  prime to  $n$ . In other words, the number of segment numbers  $[1; n]$ , the greatest common divisor with which  $n$  is unity.

### Contents [hide]

- Euler function
  - Determination
  - Properties
  - Implementation
  - The application of Euler's function
  - Tasks in the online judges

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

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

### Properties

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

- If  $p$ - a prime number, then  $\phi(p) = p - 1$ .  
(This is obvious, since any number except the  $p$  prime to it.)
- If  $p$ - simple  $a$ - a natural number, then  $\phi(p^a) = p^a - p^{a-1}$ .  
(As with the number of  $p^a$  not only relatively prime numbers of the form , which pieces.)  $pk (k \in \mathcal{N}) p^a / p = p^{a-1}$
- If  $a$  and  $b$  are relatively prime, then  $\phi(ab) = \phi(a)\phi(b)$  ("multiplicative" Euler function).  
(This follows from the Chinese remainder theorem . Consider an arbitrary number  $z \leq ab$ . We denote by  $x$  and  $y$  the remnants of the division  $z$  at  $a$  and  $b$ , respectively. Then  $z$  prime to  $ab$  if and only if  $z$  is prime to  $a$  and  $b$  separately, or what is the same,  $x$  mutually simply  $a$  and  $y$  relatively prime to  $b$ . Using the Chinese remainder theorem, we see that any pair of numbers

$x$  and the number of one-to-one correspondence , which completes the proof.)  $y (x \leq a, y \leq b) z (z \leq ab)$

From here you can get the Euler function for all  $n$  through his **factorization** (decomposition  $n$  into prime factors):

if

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

(Where everything  $p_i$ - simple), the

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

## Implementation

The simplest code calculating the Euler function, the quotient of the number of elementary method for  $O(\sqrt{n})$ :

```
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;
}
```

A key place for the calculation of the Euler function - is to find the **factorization** of  $n$ . It can be done in a time much smaller  $O(\sqrt{n})$ : see. [Efficient algorithms for factorization](#) .

## The application of Euler's function

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

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

where  $a$  and  $m$  are relatively prime.

In the particular case when  $m$  is prime Euler's theorem turns into the so-called **Fermat's little theorem** :

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

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

## Tasks in the online judges

A list of tasks that require 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: Low]
  - UVA # 10299 "Relatives" [Difficulty: Low]
  - UVA # 11327 "Enumerating Rational Numbers" [Difficulty: Medium]
  - TIMUS # 1673 "Admission to the exam" [Difficulty: High]
- 



# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 16:41  
EDIT: May 3, 2012 2:34

## Binary exponentiation

Binary (binary) exponentiation - is a technique that allows you to build any number  $n$  of th degree of  $O(\log n)$  multiplications (instead of  $n$  multiplications in the usual approach).

Moreover, the technique described here is applicable to any **of the associative** operation, and not only to the multiplication numbers.

Recall operation is called associative if for any  $a, b, c$  executed:

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

The most obvious generalization - on balances of some module (obviously, associativity is maintained). The next "popularity" is a generalization to the matrix product (it is well known associativity).

## Algorithm

Note that for any number  $a$  and **an even** number of  $n$  feasible 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, for an even  $n$  we showed how, by spending only one multiplication, we can reduce the problem to a half less.

It remains to understand what to do, if the degree **is odd**. Here we do is very simple: move on to the extent that would have an even:  $n - 1$

$$a^n = a^{n-1} \cdot a$$

So, we actually found the recurrence formula: on the degree  $n$  we go, if it chëtna to  $n/2$ , and otherwise - to  $n - 1$ . It is clear that there will be no more  $2 \log n$  transitions, before we come to  $n = 0$  (the basis of the recurrence formula). Thus,

### Contents [hide]

- Binary exponentiation
  - Algorithm
  - Implementation
  - Examples of solving problems
    - Efficient computation of Fibonacci numbers
    - Erection reshuffle  $K$  of th degree
    - Rapid application of a set of geometric operations to points
    - The number of paths in a fixed length column
    - Variation binary exponentiation: multiplication of two integers modulo
  - Tasks in the online judges

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 also optimized (divide-by-2 replaced with bit operations):

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

This implementation can be more simplified somewhat by noting that the construction of *a*the square is always performed, regardless of whether the condition is odd worked *nor* 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 already implemented in the language of Java, but only for long arithmetic class BigInteger (pow function of this class is working on the construction of the binary algorithm).

## Examples of solving problems

### Efficient computation of Fibonacci numbers

**Condition**. Given the number  $n$ . You want to calculate  $F_n$  where  $F_i$ - Fibonacci sequence .

**Solution**. More details are described in the decision [paper on the Fibonacci sequence](#) . Here we only briefly we present the essence of the decision.

The basic idea is as follows. The calculation of the next Fibonacci number is based on the knowledge of the previous two Fibonacci numbers: namely, each successive Fibonacci number is the sum of the previous two. This means that we can construct a matrix  $2 \times 2$  that will fit this transformation: how two Fibonacci numbers  $F_i$  and  $F_{i+1}$  calculate the next number, ie go to the pair  $F_{i+1}, F_{i+2}$ . For example, applying this transformation  $n$  to a couple of times  $F_0$  and  $F_1$  we get a couple  $F_n$  and  $F_{n+1}$ . Thus, elevating the matrix of this transformation in the  $n$ -th degree, we thus find the required  $F_n$  time for  $O(\log n)$  what we required.

### Erection reshuffle $k$ of th degree

**Condition**. Given permutation of  $p$  length  $n$ . Required to build it in  $k$ -s degree, ie find out, if to the identity permutation  $k$  permutation times apply  $p$ .

**Solution**. Simply apply to the interchange of  $p$  the algorithm described above binary exponentiation. No differences as compared with the construction numbers in degree - no. With the asymptotic solution is obtained  $O(n \log k)$ .

(Note. This problem can be solved more efficiently **in linear time** . To do this, simply select all the cycles in the permutation, then consider separately each cycle and taking the  $k$  modulo length of the current cycle, to find an answer for this cycle.)

### Rapid application of a set of geometric operations to points

**Condition**. Given  $n$  points  $p_i$  and are  $m$  transformations that should be applied to each of these points. Each transformation - a shift to a predetermined vector, or scaling (multiplication coefficients to specified coordinates), or around a predetermined rotation axis by a predetermined angle. Furthermore, there is a component of a cyclic repetition operation: it has the form "a predetermined number of times to repeat a predetermined list of transformations" (cyclic repetition of operations can be nested in each other).

You want to calculate the result of applying these operations to all points (effectively, ie in less than than  $O(n \cdot \text{length})$  where  $\text{length}$ - the total number of transactions that need to be done).

**Solution**. Look at the different types of transformations in terms of how they change the coordinates:

- Shift operation - it just adds to all the coordinates of the unit, the multiplication by constants.
- Scaling operation - it multiplies each coordinate by some constant.
- Operation rotational axis - it can be represented as follows: new coordinates obtained can be written as a linear combination of the old ones.

(We will not specify how this is done. For example, for simplicity, imagine it as a combination of five-dimensional rotations: first, in the planes  $OXY$ , and  $OXZ$  so that the axis of rotation coincides with the positive direction of the axis  $OX$ , then the desired rotation around an axis in the plane  $YZ$ , then reverse rotation in the plane  $OXZ$  and  $OXY$  so that the axis of rotation back to its starting position.)

Easy to see that each of these transformations - a recalculation of coordinates on linear equations. Thus, any such transformation can be written in matrix form  $4 \times 4$ :

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix},$$

which when multiplied by (left) to the line of the old coordinates and the constant unit gives a string of new coordinates and constants units:

$$(x \ y \ z \ 1) \cdot \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = (x' \ y' \ z' \ 1).$$

(Why need to enter a dummy fourth coordinate is always equal to one? Without this we would have to implement the shift operation: after the shift - this is just adding to the coordinates of the unit, the multiplication by some factors. Without the dummy unit we could only implement linear combinations of the coordinates themselves, and add to them given constants - could not.)

Now the solution of the problem becomes almost trivial. Once each elementary operation is described by the matrix, then the process described by the product of these matrices, and the operation of the cyclic repetition - the erection of this matrix to a power. Thus, during the time we  $O(m \cdot \log \text{repetition})$  can predposchitat matrix  $4 \times 4$ describing all the changes, and then simply multiply each point  $p_i$ on the matrix - thus, we will respond to all requests for time  $O(n)$ .

## The number of paths in a fixed length column

**Condition**. Given an undirected graph  $G$  with  $n$  vertices, and are given a number  $k$ . Is required for each pair of vertices  $i$  and  $j$  the number of ways to find among them with exactly  $k$  edges.

**Solution**. More details on this problem is considered in a separate article . Here we only recall the essence of the decision: we simply erect a  $k$ -th degree of adjacency matrix of the graph, and the elements of this matrix and will be a solution. The resulting asymptotic behavior -  $O(n^3 \log k)$ .

(Note. In the same article, consider other variation of this problem: when the graph is weighted, and you want to find the path of minimum weight, containing exactly  $k$  edges. As shown in this paper, this problem is also solved by means of binary exponentiation of the adjacency matrix of the graph, but instead of the usual operation of multiplication of two matrices to be used modified: instead of multiplying the amount taken, and instead of summation - taking a minimum.)

## Variation binary exponentiation: multiplication of two integers modulo

We give here an interesting variation of the binary exponentiation.

Suppose we are faced with such a challenge : to multiply two numbers  $a$  and  $b$  modulo  $m$ :

$$a \cdot b \pmod{m}$$

Suppose that the numbers can be quite large: so that the numbers themselves are placed in a built-in data types, but their immediate work  $a \cdot b$ - no longer exists (note that we also require that the sum of the numbers placed in a built-in data type). Accordingly, the task is to find the unknown quantity  $(a \cdot b) \pmod{m}$ , without the help of a long arithmetic .

**The decision** is as follows. We simply apply the algorithm binary exponentiation described above, but instead of multiplication, we will make the addition. In other words, multiplication of two numbers, we have reduced to  $O(\log m)$  the operations of addition and multiplication by two (which is also, in fact, there is the addition).

(Note. This problem can be solved in another way , by resorting to the help of operations with floating-point numbers. Namely, the count in the floating point expression  $a \cdot b/m$ , and round it to the nearest integer. Thus we find the approximate quotient. take it away from the product  $a \cdot b$ (ignoring the overflow), we are likely to obtain a relatively small number that can be taken modulo  $m$ - and return it as an answer. This solution looks pretty unreliable, but it is very fast, and very briefly implemented.)

## Tasks in the online judges

List of tasks that can be solved using a binary exponentiation:

- SGU # 265 "Wizards" [Difficulty: Medium]
-

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 17:54  
EDIT: 31 Aug 2011 14:48

## Euclid's algorithm 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  $b$ . English "greatest common divisor" is spelled "greatest common divisor", and its common designation is  $\gcd$ :

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

(Here the symbol " $|$ " denotes the divisibility, ie, " $k|a$ " means " $k$  divide  $a$ ")

When one of the numbers is zero, and the other is non-zero, their greatest common divisor, by definition, be it the second number. When both numbers are equal to zero, the result is not defined (any suitable infinite number), we put in this case the greatest common divisor of zero. Therefore, we can speak of such a rule: if one of the numbers zero, the greatest common divisor equal to the second number.

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

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

## Algorithm

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

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

### Contents [hide]

- Euclid's algorithm finding the GCD (greatest common divisor)
  - Algorithm
  - Implementation
  - Proof of correctness
  - Operation time
  - LCM (least common multiple)
  - Literature

## 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 present and non-recursive algorithm form:

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

## Proof of correctness

First, we note that for each iteration of the algorithm of Euclid its second argument is strictly decreasing, therefore, since it is non-negative, then the Euclidean algorithm **always terminates**.

To **prove the correctness** we need to show that  $\gcd(a, b) = \gcd(b, a \bmod b)$  for any  $a \geq 0, b > 0$ .

We show that the quantity on the left-hand side is divided by this in the right, and the right-hand - is divided into the left-hand. Obviously, this will 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:

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

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

$$\begin{cases} d \mid b, \\ d \mid (a \bmod b) \end{cases}$$

Now we use the following simple fact: if for any three numbers  $p, q, r$  fulfilled:  $p|q$  and  $p|r$  then  $p|\gcd(q, r)$ . In our situation, we get:

$$d \mid \gcd(b, a \bmod b)$$

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

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

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

## Operation time

Time of the algorithm is evaluated **Lame theorem**, which establishes a surprising connection of the Euclidean 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, it can be shown that the upper bound of the theorem - optimal. When  $a = F_n, b = F_{n-1}$  it will be done  $n - 2$  recursive call. In other words, **successive Fibonacci numbers - the worst input** to the Euclidean algorithm.

Given that the Fibonacci numbers grow exponentially (as a constant in degree  $n$ ), we find that the Euclidean algorithm is performed for  $O(\log \min(a, b))$  multiplication.

## LCM (least common multiple)

The calculation of the least common multiple (least common multiplier, lcm) is reduced to the calculation of gcd 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  $\text{gcd}$ , and only then is multiplied by  $b$ , as this will help to avoid overflows in some cases)

## Literature

- Thomas feed, Charles Leiserson, Ronald Rivest, Clifford Stein.  
**Introduction to Algorithms** [2005]

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 10 Jun 2008 17:58  
EDIT: 17 Oct 2012 14:55

## Advanced Euclidean algorithm

While the "normal" Euclidean algorithm simply finds the greatest common divisor of two numbers  $a$  and  $b$ , extended Euclidean algorithm is in addition to the GCD and the coefficients  $x$  and  $y$  such that:

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

Ie it finds the coefficients by which the GCD of two numbers expressed in terms of these numbers themselves.

### Contents [hide]

- Advanced Euclidean algorithm
  - Algorithm
  - Implementation
  - Literature

## Algorithm

Make the calculation of these coefficients in the Euclidean algorithm is easy enough to deduce the formula by which they change from pair  $(a, b)$  to pair  $(b \% a, a)$  (percent sign we mean taking the remainder of the division).

Thus, suppose we have found the solution  $(x_1, y_1)$  of the problem for a new pair  $(b \% a, a)$ :

$$(b \% a) \cdot x_1 + a \cdot y_1 = g,$$

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 of  $b \% a$ :

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

Substituting this in the above expression  $x_1$  and  $y_1$  and get:

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

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;
}
```

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

Base of recursion - case  $a = 0$ . Then GCD is  $b$ , and obviously, the desired ratio  $x$  and  $y$  are  $0$  and  $1$ , respectively. In other cases, the usual solution is working, and the coefficients are translated at the above formulas.

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

## Literature

- Thomas feed, Charles Leiserson, Ronald Rivest, Clifford Stein.  
**Introduction to Algorithms** [2005]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 17:57  
EDIT: Mar 23 2012 3:53

## Sieve of Eratosthenes

Sieve of Eratosthenes - an algorithm for finding all prime numbers in the interval  $[1; n]$  of  $O(n \log \log n)$  operations.

The idea is simple - we will write a series of numbers  $1 \dots n$ , and we will strike out at first all numbers divisible by  $2$ , except for the number  $2$  and then dividing by  $3$ , except for the numbers  $3$ , then on  $5$ , then  $7, 11$  and everything else is just up  $n$ .

### Contents [hide]

- Sieve of Eratosthenes
  - Implementation
  - Asymptotics
  - Various optimizations sieve of Eratosthenes
    - Sifting easy to root
    - Sieve only in odd numbers
    - To reduce the amount of memory consumed
    - Block sieve
    - Upgrade to linear time work

## Implementation

Immediately we present 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  $2$  before  $n$ , and if the current number of  $i$  simple, it ticks all the multiples of him as a constituent.

At the same time we start to go from  $i^2$  as fewer multiples  $i$ , be sure to have a prime divisor less  $i$ , which means that they have already been eliminated earlier. (However, as  $i^2$  can easily overwhelm type `int` in the code before the second nested loop is an additional check using the type `long long`.)

With this realization algorithm consumes  $O(n)$  memory (obviously) and performs  $O(n \log \log n)$  actions (this is proved in the next section).

## Asymptotics

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

So, for every prime  $p \leq n$  inner loop will be executed, which will make  $\frac{n}{p}$  action. Therefore, we need to estimate the following value:

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{n}{p} = n \cdot \sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{1}{p}.$$

Let us recall here two well-known fact: that the number of primes less than or equal to  $n$  approximately equal  $\frac{n}{\ln n}$ , and that the  $k$ -th prime number is approximately equal to  $k \ln k$  (this follows from the first statement). Then the sum can be written as follows:

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{1}{p} \approx \frac{1}{2} + \sum_{k=2}^{\ln n} \frac{1}{k \ln k}.$$

Here, we have identified the first prime of the sum, since the  $k = 1$  approximation according to the  $k \ln k$  will 0, which will lead to a division by zero.

Now we estimate this sum by the integral of the same function on  $k$  from 2 before  $\frac{n}{\ln n}$  (we can produce such an approximation, since, in fact, refers to the sum of the integral as his approach to the rectangle formula):

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

Antiderivative for the integrand there  $\ln \ln k$ . Substitute and removing members of the lower order, we obtain:

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

Now, returning to the original 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 gives a more accurate estimate, 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 - the fact that he "walks" from memory, constantly go beyond the cache, which is why the constant hidden in the  $O(n \log \log n)$  relatively large.

In addition, for sufficiently large  $n$  volume becomes the bottleneck of memory consumption.

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

### Sifting easy to root

The most obvious point - that in order to find all simple to  $n$  sufficiently perform only simple sieving not exceeding root  $n$ .

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 (in fact, repeating the proof given above, we obtain an estimate  $n \ln \ln \sqrt{n} + o(n)$  that, on the properties of logarithms, asymptotically have the same thing), although the number of transactions decreased markedly.

### Sieve only in odd numbers

Since all the even numbers, except 2- components, we can not process any way at all even numbers and

odd numbers only operate.

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

## To reduce the amount of memory consumed

Note that the algorithm of Eratosthenes actually operates with  $n$  bits of memory. Consequently, we can save significant memory consumption is not stored  $n$  bytes - booleans, and  $n$  bits, ie  $n/8$  bytes of memory.

However, this approach - "bit compression" - significantly complicate handling these bits. Any read or write bits will be of a few arithmetic operations, which ultimately lead to slower algorithm.

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

In conclusion, it is worth noting that in C ++ containers have already been implemented, automatically performing bit compression: vector <bool> and bitset <>. However, if speed is important, it is better to implement a bit compressed manually using bit operations - today compilers still not able to generate enough fast code.

## Block sieve

Optimization of the "simple screening to the root" it follows that there is no need to store all the time the whole array  $\text{prime}[1 \dots n]$ . To perform screening sufficient to store only simple to root  $n$ , that is  $\text{prime}[1 \dots \sqrt{n}]$ , the remainder of the array  $\text{prime}$  to build block by block, keeping the current time, only one block.

Let  $s$  - constant that determines the block size, then all will be  $\lceil \frac{n}{s} \rceil$  blocks,  $k$ th block ( $k = 0 \dots \lfloor \frac{n}{s} \rfloor$ ) contains a number in the interval  $[ks; ks + s - 1]$ . Will process the blocks of the queue, i.e. for each  $k$ th block will go through all the simple (from 1 before  $\sqrt{n}$ ) and perform their screening only within the current block. Gently should handle the first block - firstly, from simple  $[1; \sqrt{n}]$  do not have to remove themselves, and secondly, the number 0 and 1 must be marked as not particularly simple. When processing the last block should also not forget that the latter desired number  $n$  is not necessarily the end of the block.

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

```

const int SQRT_MAXN = 100000; // корень из максимального значения 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;
            if (j < start)
                continue;
            if (j <= nsqrt)
                nprime[j] = true;
            if (j <= maxk)
                bl[j] = true;
        }
        for (int j=0; j<=maxk; ++j)
            if (bl[j])
                result++;
    }
    cout << result;
}

```

```

        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 a conventional sieve of Eratosthenes (unless, of course, the size of the blocks is not very small), but the amount of memory used will be reduced to  $O(\sqrt{n} + s)$  decrease and "walk" from memory. On the other hand, for each block for each of the simple  $[1; \sqrt{n}]$  division will be performed that will strongly affect in a smaller unit. Consequently, the choice of the constant  $s$  need to keep a balance.

Experiments show that the best speed is achieved when the  $s$  value is about  $10^4$  to  $10^5$ .

## Upgrade to linear time work

Eratosthenes algorithm can be converted into a different algorithm, which already works in linear time - see Article "[Sieve of Eratosthenes linear-time work](#)" . (However, this algorithm has some limitations.)

---

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: Sep 6, 2011 1:03  
EDIT: Mar 23, 2012 3:58

## Sieve of Eratosthenes with linear time work

Given the number  $n$ . You want to find **all the prime** in the interval  $[2; n]$ .

The classic way to solve this problem - **the 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 working for sublinear time (ie  $o(n)$ ), the algorithm described below is interesting for its **simplicity** - it is practically difficult classical sieve of Eratosthenes.

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

The disadvantage driven algorithm is that it uses **more memory** than the classic Sieve of Eratosthenes: Requires start an array of  $n$  numbers, while classical sieve of Eratosthenes only enough  $n$  memory bit (what happens in 32 times less).

Thus, the described algorithm should be applied only up to the order of numbers  $10^7$ , no more.

Authorship algorithm apparently belongs Grice and Misra (Gries, Misra, 1978 - see. List of references at the end). (And, in fact, to call this algorithm "Sieve of Eratosthenes" incorrectly too the difference between these two algorithms.)

### Contents [hide]

- Sieve of Eratosthenes with linear time work
  - Description of the algorithm
  - Implementation
  - Proof of correctness
  - Time and required memory
  - Literature

## Description of the algorithm

Our goal - to count for each number  $i$  in the interval of  $[2; n]$  its **minimal prime divisor**  $lp[i]$ .

In addition, we need to keep a list of all found primes - let's call it an array  $pr[]$ .

Initially, all values are  $lp[i]$  filled with zeros, which means that we still assume all the numbers simple. In the course of the algorithm, this array will gradually be filled.

We will now sort out the current number  $i$  of  $2$  up  $n$ . We can have two cases:

- $lp[i] = 0$ - This means that the number  $i$ - easy because for it has not found other dividers.

Therefore, it is necessary to assign  $lp[i] = i$  and add  $i$  to the end of the list  $pr[]$ .

- $lp[i] \neq 0$ - This means that the current number  $i$ - composite, and its minimal prime divisor is  $lp[i]$ .

In both cases, then begins the process of **alignment of values** in the array  $lp[]$ : we take numbers, **multiples**  $i$ , and update their value  $lp[]$ . However, our goal - to learn how to do it so that in the end of each value  $lp[]$  was found to be no more than once.

It is argued that this is possible to do so. Consider the numbers of the form:

$$x_j = i \cdot p_j,$$

where the sequence  $p_j$ - it's simple, do not exceed  $lp[i]$  (just for this, we need to keep a list of all prime numbers).

All numbers such type places a new value  $lp[x_j]$ - obviously, it will be equal  $p_j$ .

Why such an algorithm is correct, and why it works in linear time - see. Below, but for now we present

its implementation.

## Implementation

Sieve performed prior to said 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 speed up a little, getting rid of the vector  $pr$ (replacing it with a regular array with counter), as well as getting rid of duplicate multiplication in the nested loop  $for$ (for which the result of the product must be easy to remember in any variable).

## Proof of correctness

We prove the **correctness** of the algorithm, ie, he puts all values correctly  $lp[]$ , and each of them will be set only once. This will imply that the algorithm runs in linear time - since all other steps of the algorithm is obviously working for  $O(n)$ .

For this we note that any number of **unique representation** of the form:  $i$

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

where  $lp[i]$ - (as before) the minimum prime divisor of  $i$ , and the number  $x$  has no divisors less  $lp[i]$ , ie.:

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

Now compare this with the fact that our algorithm does - he actually for each  $x$  through all simple, for which it can multiply, ie, Just prior  $lp[x]$  inclusive, to obtain a number of the above representation.

Therefore, the algorithm does take place for each composite number exactly once, putting his right value  $lp[]$ .

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

## Time and required memory

Although the asymptotic behavior of  $O(n)$  the asymptotic behavior of the best  $O(n \log \log n)$  classic sieve of Eratosthenes, the difference between them is small. In practice this means a twofold difference in speed, and optimized versions of the sieve of Eratosthenes and not lose the algorithm given here.

Given the costs of memory that is required by this algorithm - array of  $lp[]$  length  $n$  and an array of all the simple  $pr[]$  length about  $n / \ln n$ - this algorithm seems inferior to the classical sieve on all counts.

However, it saves that array  $lp[]$ , which is calculated with this algorithm allows to search for the factorization of any number in the interval  $[2; n]$  during the order of the size of the factorization.

Moreover, the additional cost of another array, you can do that in this factorization is not required of the division operation.

Knowledge of the factorization of all 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]

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 10 Jun 2008 17:59  
 EDIT: 28 Aug 2011 23:42

## Fibonacci numbers

### Determination

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, ...

### Contents [hide]

- Fibonacci numbers
  - Determination
  - History
  - Fibonacci numbers in nature
  - Properties
  - Fibonacci coding
  - The formula for the n-th Fibonacci number
    - Formula by radicals
    - Matrix formula for the Fibonacci numbers
  - The periodicity of the Fibonacci sequence modulo
  - Literature

### History

These numbers introduced in 1202 by Leonardo Fibonacci (Leonardo Fibonacci) (also known as Leonardo of Pisa (Leonardo Pisano)). However, precisely because of the mathematics of the 19th century Luke (Lucas) called "Fibonacci numbers" became common.

However, the number of Indian mathematicians mentioned this sequence even earlier: Gopal (Gopala) until 1135, Hemachandra (Hemachandra) - in 1150

### Fibonacci numbers in nature

Fibonacci himself mentioned these numbers in connection with this challenge: "A man planted a pair of rabbits in a pen 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 gives birth to a pair? ". The solution to this problem, and will be the number of sequences, now called after him. However, the situation described by the Fibonacci - more mind game than real nature.

Indian mathematicians and Gopal Hemacandra mentioned this number sequence

in relation to the number of rhythmic patterns, formed as a result of alternating long and short syllables in verse or the strengths and weaknesses of shares in the music. The number of such patterns having generally  $n$  share equal  $F_n$ .

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

An interesting example of plants - yarrow, in which the number of stems (and hence the flowers) is always a Fibonacci number. The reason for this is simple: only being initially with stem, the stem is then divided into two, and then branches from the main stem of another, then the first two branch stem again, then all stems, except the last two, branch, and so on. Thus, each stem after his appearance "skips" one branch, and then begins to divide at every level of branches, which results in a Fibonacci number.

Generally speaking, many colors (for example, lilies), the number of petals is one way or another Fibonacci number.

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

For many plants (according to some sources, 90% of them) are true and such an interesting fact. Consider any sheet, and will descend downwardly until, until we reach the sheet disposed on the stem in the same way (i.e., exactly directed in the same direction). Along the way, we assume that all the leaves that fall to 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 the make turns clockwise or counterclockwise, is a different number of turns. But it turns out that the number of turns committed contact clockwise, the number of turns counterclockwise committed, and the number of leaves to form 3 encountered successive Fibonacci numbers.

However, it should be noted that there are plants for which the calculations given above will give the number of very different sequences, so we can not say that the phenomenon of phyllotaxis is the law - it is rather amusing trend.

## Properties

Fibonacci numbers have many interesting mathematical properties.

Here are just a few of them:

- Cassini ratio:

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

- The rule of "addition":

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

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

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

- From the previous equality by induction we can show that

$F_{nk}$  always fold  $F_n$ .

- The converse is true to the previous statement:

if  $F_m$  fold  $F_n$ , the  $m$  fold  $n$ .

- 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 to this algorithm (see. "Theorem Lama" in [the Euclidean algorithm](#) ).

## Fibonacci coding

**Zeckendorf theorem** states that every positive integer  $n$  can be uniquely represented 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 coding**, for example:

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

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

It is easy to obtain and generally adding one to the number in the Fibonacci coding: if the least significant digit is 0, then it is replaced by 1, and if equal to 1 (ie, the end is worth 01), then replace 01 by 10. Then the "fix" record, consistently correcting all 011 to 100. As a result, in linear time a new entry is received numbers.

Translation of a Fibonacci number system is as simple as "greedy" algorithm: just iterate through the Fibonacci numbers from largest to smallest, and if for some  $F_k \leq n$ , it  $F_k$  is included in the record numbers  $n$ , and we subtract  $F_k$  from  $n$ , and continue the search.

# The formula for the n-th Fibonacci number

## Formula by radicals

There is a wonderful formula is named 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 or using the 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 gives "almost" value  $F_n$ . This can be written in the simple form:

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

where the square brackets denote rounding to the nearest integer.

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

## Matrix 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

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

we obtain:

$$(F_0 \ F_1) \cdot P^n = (F_n \ F_{n+1}).$$

Thus, to find  $n$ th Fibonacci array must build  $P$  a degree  $n$ .

Remembering that the construction of a matrix  $n$  of th degree can be

accomplished  $O(\log n)$ (see. [The binary exponentiation](#) ), it turns out that  $n$ the number of Fibonacci retracement can be easily calculated for  $O(\log n)c$  using only integer arithmetic.

## The periodicity of the Fibonacci sequence modulo

Consider the Fibonacci sequence  $F_i$ modulo some number  $p$ . We prove that it is periodic, and moreover period begins  $F_1 = 1$ (ie preperiod contains only  $F_0$ ).

We prove this by contradiction. Consider a  $p^2 + 1$ pair of Fibonacci numbers, taken modulo  $p$ :

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

Since the modulus  $p$ may be only  $p^2$ different pairs, there exists among the sequence of at least two of the same pair. This already means that the sequence is periodic.

We now choose among all these identical pairs of two identical pairs with the lowest number. Let this pair with some numbers  $(F_a, F_{a+1})$ and  $(F_b, F_{b+1})$ . We will prove that  $a = 1$ . Indeed, otherwise there will be for them the previous pair  $(F_{a-1}, F_a)$ , and  $(F_{b-1}, F_b)$ which, in the properties of Fibonacci numbers will also be equal to each other. However, this contradicts the fact that we have chosen the matching pairs with the lowest number, as required.

## Literature

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

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 18:03  
EDIT: 15 Dec 2011 21:27

## An inverse modulo ring

### Determination

Let some natural unit  $m$  and consider the ring formed by the module (ie, consisting of numbers from 0 before  $m - 1$ ). 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  $a^{-1}$ .

It is clear that for the zero inverse element does not exist any time; for the remaining elements of the inverse can either exist or not. It is argued that the inverse exists only for those elements  $a$  that **are relatively prime** to the module  $m$ .

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

In conclusion, let us consider an algorithm that allows us to find back to all those on some module in linear time.

### Finding using the Extended Euclidean algorithm

Consider the auxiliary equation (relatively unknown  $x$  and  $y$ ):

$$a \cdot x + m \cdot y = 1.$$

This **linear Diophantine equation of the second order**. As shown in the relevant article, the condition  $\gcd(a, m) = 1$  implies that this equation has a solution that can be found using the **Extended Euclidean algorithm** (hence the same way, it follows that when  $\gcd(a, m) \neq 1$ , solutions, and therefore return the item does not exist).

On the other hand, if we take on both sides of the balance of the module  $m$ , we get:

### Contents [hide]

- An inverse modulo ring
  - Determination
  - Finding using the Extended Euclidean algorithm
  - Finding via binary exponentiation
  - Finding all simple for a given module in linear time

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

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

The implementation (including that found  $x$  necessary to take modulo  $m$ , and  $x$  could 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;
}
```

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

## Finding via 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 even 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 get:

- for each module  $m$ :

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

- for easy module  $m$ :

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

Thus, we have the formula for the direct calculation of the return. For practical applications typically use an efficient [algorithm for binary exponentiation](#), which in this case will help to make for exponentiation  $O(\log m)$ .

This method appears to be somewhat easier to describe in the preceding paragraph, but it requires knowledge of the values of the Euler function that actually requires the factorization of the module  $m$ , which can sometimes be very difficult.

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

## Finding all simple for a given module in linear time

Let a simple module  $m$ . Required for each number in the interval  $[1; m - 1]$  to find its inverse.

Using the algorithms described above, we get a solution with the asymptotic behavior  $O(m \log m)$ . Here we present a simple solution with the asymptotic behavior  $O(m)$ .

**The decision** is as follows. Denoted by  $r[i]$  the required number of inverse  $i$  modulo  $m$ . Then the  $i > 1$  true identity:

$$r[i] = - \left\lfloor \frac{m}{i} \right\rfloor \cdot r[m \bmod i]. \quad (\bmod m)$$

**Implementation of** 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** of this solution is a chain of simple transformations:

We write the value  $m \bmod i$ :

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

hence, both taking part in modulus  $m$ , we obtain:

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

Multiplying both sides by the inverse  $i$  and inverse to  $(m \bmod i)$  obtain the required formula:

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

QED.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 18:05  
EDIT: 24 Aug 2011 1:41

## Gray code

### Determination

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

#### Contents [hide]

- Gray code
  - Determination
  - Finding a Gray code
  - Finding the inverse Gray code
  - Applications
  - Tasks in the online judges

For example, for the numbers of length 3 bits have the following 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 a Gray code

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

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

### Finding the inverse Gray code

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

Let's go from the older to the younger bits (albeit the least significant bit has the number 1, and the oldest -  $k$ ). Obtain the following relations between the bits  $n_i$  number of  $n$  bits and  $g_i$  number  $g$ :

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

In the form of 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 various fields, sometimes quite unexpected:

- $n$ -bit Gray code corresponds to a Hamiltonian cycle on  $n$ -dimensional cube.
- In the art, Gray codes are used to **minimize errors** when converting the analog signals into digital signals (for example, sensors). In particular, the Gray codes were discovered in connection with this application.
- Gray codes are used in solving the problem of the **Towers of Hanoi**.

Let  $n$ - the number of disks. Let's start with the Gray code length  $n$ , consisting of zeros (that  $G(0)$ ), and we will move on Gray codes (from  $G(i)$  proceeding to  $G(i + 1)$ ). With every  $i$ -bit of this -omu Gray code  $i$ th disc (and the most significant bit corresponds to the smallest size disk, and the oldest bat - the greatest). Since at each step exactly one bit is changed, then we can understand the change bit  $i$ as moving  $i$ th disc. Note that for all drives except the smallest, at each step, there is exactly one version of the course (except for the starting and final products). For the smallest drive always has two options progress, but there is progress in the strategy of choice, always leads to the answer: if  $n$ is odd, then the sequence of movements of the smallest drive has the form

$f \rightarrow t \rightarrow r \rightarrow f \rightarrow t \rightarrow r \rightarrow \dots$  (where  $f$ - starting rod  $t$ - the final rod  $r$ - the remainder of the rod), and if it  $n$ is even, then

$f \rightarrow r \rightarrow t \rightarrow f \rightarrow r \rightarrow t \rightarrow \dots$

- Gray codes are also used in the theory of **genetic algorithms**.

## Tasks in the online judges

List of tasks that can be taken using Gray codes:

- SGU # 249 "Matrix" [Difficulty: Medium]

## Long arithmetic

Long arithmetic - is a set of software tools (data structures and algorithms) that allow you to work with numbers much larger quantities than permitted by the standard data types.

### Types of integer arithmetic long

Generally speaking, even if only in the Olympiad problems toolkit is large enough, so proizvedem classification of different types of long arithmetic.

#### Classic long arithmetic

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

The numbers can be used from a given number system, commonly used decimal system and its extent (ten thousand billion), or binary system.

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

It describes the work only with non-negative long numbers. Support for negative numbers must enter and maintain an additional flag "negative" number or the same work in complementary codes.

#### Data structure

Keep long numbers will be in the form of a vector of numbers *int*, where each element - a 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 the first are the least significant digit (ie, units, tens, hundreds, etc.).

Furthermore, all the operations are implemented in such a manner that after any of them leading zeros (i.e., extra number of leading zeros) are absent (of course under the assumption that prior to each operation as no leading zeros). It should be noted that the implementation representation for the number of zero correctly supported once two views: the empty vector numbers and vector of numbers containing a single element - zero.

#### Output

The most simple - it is the conclusion of a long number.

First, we simply display the last element of the vector (or 0, if the vector is empty), and then displays all the remaining elements of the vector, adding zeros to their 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 little thin point: you do not forget to write down the cast (*int*), because otherwise the number *a.size()* will be unsigned, and if *a.size()*  $\leq$  1, when subtracting the overflow happens)

#### Reading

Reads a line in *string*, and then convert it to 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()));
```

### Contents [hide]

- Long arithmetic
  - Types of integer arithmetic long
    - Classic long arithmetic
      - Data structure
      - Output
      - Reading
      - Addition
      - Subtraction
      - Multiplying the length of a short
      - Multiplication of two long numbers
      - Dividing the length of a short
      - Long arithmetic in factored form
      - Long arithmetic system of simple modules (Chinese theorem or scheme Garner)
  - Types of fractional arithmetic long
    - Long arithmetic in an irreducible fraction
    - Isolation of the floating-point position in a particular type of

If used instead of *string* the array *char*'s, the code will be more compact:

```
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 *a* the 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* the number of *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 subtraction remove leading zeros to maintain the predicate that they do not exist.

## Multiplying the length of a short

Multiples long *a* to short *b* ( $b < \text{base}$ ) 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 dividing remove leading zeros to maintain the predicate that they do not exist.

(Note: The way to **further optimization**. If speed is critical, you can try to replace the two division one: to count only the integer part of the division (the code is a variable *carry*), and then count on it the remainder of the division (with a single multiplication). As a rule, this technique allows faster code, but not very much.)

## Multiplication of two long numbers

Multiples *a* on *b* and the result is stored in *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();
```

## Dividing the length of a short

Divides the length of a short  $b$  ( $b < \text{base}$ ), private stores  $a$ , the remainder in  $carry$ :

```
int carry = 0;
for (int i=(int)a.size()-1; i>=0; --i) {
    long long cur = a[i] + carry * 1ll * 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

Here the idea is to store not the number itself, and its factorization, ie degree of each incoming it simple.

This method is also very simple to implement, and it is very easy to perform multiplication and division, but you can not make addition or subtraction. On the other hand, this method saves memory compared to the "classic" approach, and allows the multiplication and division significantly (asymptotically) faster.

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

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

The bottom line is that a system of modules selected (usually small, placed in a standard data types), and the number is stored as a vector from the remnants of its division into each of these modules.

According to the Chinese remainder theorem, it is sufficient to uniquely store any number in the range from 0 to the product of these modules minus one. At the same time there [Garner algorithm](#) which allows to produce a restoration of the modular type of a conventional "classical" form of numbers.

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

However, given the cost of all this is very time consuming translation of this type of unit in the usual form, which, in addition to considerable time-consuming, require also the implementation of "classical" long arithmetic multiplication.

In addition, to make **the division** of numbers in this representation system of simple modules is not possible.

## Types of fractional arithmetic long

Operations on fractional numbers found in the Olympiad problems are much less common and work with large fractional numbers is much more difficult, so in competitions found only a specific subset of fractional long arithmetic.

## Long arithmetic in an irreducible fraction

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

Normally this storage numerator and denominator have to use long arithmetic, but, however, it is the simplest kind of - "classic" long arithmetic, although sometimes is sufficiently embedded 64-bit numeric type.

## Isolation of the floating-point position in a particular type of

Sometimes the task required to make calculations with very large or very small numbers, but it does not prevent them from overflowing. Built 8 – 10-baytov type *double* known to allow the exponent value in a range  $[-308; 308]$ , which sometimes may be insufficient.

Admission actually very simple - enter another integer variable representing the exponent, and after each operation fractional number of "normal", ie returns to the segment  $[0.1; 1]$ , by increasing or decreasing the exponent.

When multiplying or dividing two such numbers must be correspondingly folded or subtract their exponents. When adding or subtracting before performing this operation, the number should lead to a single exponential function, for which one of them is multiplied by the 10 difference in the degree of the exhibitor.

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 most profitable seems to put an equal basis 2.

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 10 Jun 2008 18:08  
EDIT: Mar 23 2012 3:50

## Discrete logarithm

The discrete logarithm problem is that according to the whole  $a, b, m$  to solve the equation:

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

where  $a$  and  $m$  are relatively prime (note: if they are not relatively prime, then the algorithm described below is incorrect, although, presumably, it can be modified so that it still worked).

Here we describe an algorithm, known as "**Baby-Step-giant-Step algorithm**", proposed by **Shanks (Shanks)** in 1971, working at the time for  $O(\sqrt{m} \log m)$ . Often simply referred to as the algorithm algorithm "**meet-in-the-middle**" (because it is one of the classic applications of technology "meet-in-the-middle": "separation of tasks in half").

### Contents [hide]

- Discrete logarithm
  - Algorithm
  - Asymptotics
  - Implementation
    - The simplest implementation
    - Improved implementation

## Algorithm

So, we have the equation:

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

where  $a$  and  $m$  are relatively prime.

Transform equation. Let

$$x = np - q,$$

where  $n$ - is the preselected constant (as her chosen depending on  $m$ , we understand a little later). Sometimes  $p$  called "giant step" (since the increase per unit increase it  $x$  at once  $n$ ), and in contrast to it  $q$ - "baby step".

Obviously, any  $x$  (the interval  $[0; m]$ ) - it is clear that such a range of values will suffice) can be represented in this form, with this value will suffice:

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

Then the equation becomes:

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

where, using the fact that  $a$  and  $m$  are relatively prime, we obtain:

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

To solve the original equation, we must find the appropriate values  $p$  and  $q$  to the values of the left and right parts of the match. In other words, it is necessary to solve the equation:

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

This problem is solved by the method meet-in-the-middle follows. The first phase of the algorithm: calculate the value of the function  $f_1$  for all values of the argument  $p$ , and we can sort these values. The second phase of the algorithm: we will sort out the value of the second variable  $q$ , calculate a second function  $f_2$ , and look for this value among the predicted values of the first function using a binary search.

## Asymptotics

First we estimate the computation time of each of the functions  $f_1(p)$  and  $f_2(q)$ . And she and the other contains exponentiation, which can be performed using the algorithm of binary exponentiation. Then both of these functions, we can calculate a time  $O(\log m)$ .

The algorithm itself in the first phase comprises computing function  $f_1(p)$  for each possible value  $p$  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  $f_2(q)$  for each possible value  $q$  and the binary search on an array of values  $f_1$  that gives us the asymptotic behavior:

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

Now, when we add these two asymptotic behavior we would get  $\log m$  multiplied by the amount  $n$  and  $m/n$  and almost obvious that the minimum is attained when  $n \approx m/n$ , that is, algorithm for optimal constant  $n$  should be selected:

$$n \approx \sqrt{m}.$$

Then the asymptotic behavior of the algorithm takes the following form:

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

Note. We could swap roles  $f_1$  and  $f_2$  (ie, in the first phase to calculate the value of the function  $f_2$ , and the second -  $f_1$ ), but it is easy to understand that the result will not change, and the asymptotic behavior, we can not improve.

## Implementation

### The simplest implementation

The function `powmod` performs a binary construction of  $a$  the power  $b$  modulo  $m$  see. [A binary exponentiation](#).

Function `solve` to produce its own solution to the problem. This function returns a response (the number in the interval  $[0; m)$ ), or more precisely, one of the answers. Function will return  $-1$  if there is no solution.

```

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  $f_1(i)$  stores the argument  $i$  in which this value is achieved. Moreover, 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 found in the response interval  $[0; m]$ .

Given that the argument of  $f_1()$  the first phase we have fingered from one and up to  $n$  and argument of  $f_2()$  the second phase moves from zero to  $n$ , in the end, we cover the whole set of possible answers, because segment  $[0; n^2]$  contains a gap  $[0; m]$ . In this case, a negative response could not get, and the responses of greater than or equal  $m$ , we can not ignore - still must be corresponding answers from the interval  $[0; m]$ .

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

## Improved implementation

When **optimizing for speed**, you can proceed as follows.

Firstly, immediately catches the eye uselessness binary exponentiation in the second phase of the algorithm. Instead, you can just make it multiply the variable and every time  $a$ .

Second, in the same way you can get rid of the binary exponentiation and the first phase: in fact, once is enough to calculate the value  $a^n$ , and then simply multiplied by it.

Thus, the logarithm in the asymptotic will remain, but it will only log associated with the data structure *map* <> (ie, in terms of the algorithm, sorting and binary search values) - ie it will be the logarithm of  $\sqrt{m}$  that in practice gives a noticeable boost.

```

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 unit  $m$  is small enough, then we can do to get rid of the log in the asymptotic behavior - instead of just having got  $\text{map} <>$ a regular array.

You can also remember the hash table: on average they work well for  $O(1)$  that, in general gives the asymptotic behavior  $O(\sqrt{m})$ .

---

## Linear Diophantine equations in two variables

Diophantine equation with two unknowns is as follows:

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

where  $a, b, c$ - given integers,  $x$ and  $y$ - unknown integers.

Below are several classical problems for these equations: finding any solution, obtaining all solutions, finding the number of solutions and the solutions themselves in a certain interval, finding a solution with the least amount of unknowns.

### Contents [hide]

- Linear Diophantine equations in two variables
  - Degenerate case
  - Finding a solution
  - Getting all the solutions
  - Finding the number of solutions and the solutions themselves in a given interval
  - Finding solutions in a given interval with the least amount of  $x + y$
  - Tasks in the online judges

### Degenerate case

One degenerate case we immediately exclude from consideration when  $a = b = 0$ . In this case, of course, the equation has infinite number of random or solutions, or solutions have not at all (depending on whether  $c = 0$ or not).

### Finding a solution

Find one of the solutions of the Diophantine equation with two unknowns, you can use [the Extended Euclidean algorithm](#). Assume first that the numbers  $a$ and  $b$ are non-negative.

Advanced Euclidean algorithm for nonnegative numbers  $a$ and  $b$ find their greatest common divisor  $g$ , as well as such factors  $x_g$ and  $y_g$ that:

$$a \cdot x_g + b \cdot y_g = g.$$

It is argued that if  $c$ divisible by  $g = \gcd(a, b)$ , the Diophantine equation  $a \cdot x + b \cdot y = c$ has a solution; otherwise Diophantine equation has no solutions. This follows from the obvious fact that a linear combination of the two numbers must continue to share their common divisor.

Suppose that  $c$ is divided into  $g$ , then obviously performed:

$$a \cdot x_g \cdot (c/g) + b \cdot y_g \cdot (c/g) = c,$$

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

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

We have described the decision in the case where the number  $a$ and  $b$ 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 found to change the sign  $x_0$ and  $y_0$ the present symbol numbers  $a$ and  $b$ , respectively.

Implementation (remember, here we assume that the input data is  $a = b = 0$ not 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 obtain all the other solutions (and there are an infinite number) of the Diophantine equation, knowing one of the solutions  $(x_0, y_0)$ .

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

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

Then we note that, by adding to  $x_0$  the number  $b/g$  and at the same time taking away  $a/g$  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, i.e. 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 of this type are solutions, i.e. We describe the set of all solutions of the Diophantine equation (it turned out to be infinite if not impose additional conditions).

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

Suppose we are given two segments  $[min_x; max_x]$  and  $[min_y; max_y]$ , and you want to find the number of solutions  $(x, y)$  of the Diophantine equations underlying the data segments, respectively.

Note that if one of the numbers  $a, b$  is zero, then the problem has no more than one solution, so these cases we have in this section exclude from consideration.

First, find a suitable solution with minimal  $x$ , ie  $x \geq min_x$ . To do this, first find any solution of the Diophantine equation (see para. 1). Then get out of it the solution with the least  $x \geq min_x$ - for this we use the procedure described in the preceding paragraph and shall decrease / increase  $x$ , until it is  $\geq min_x$ , and thus minimal. This can be done  $O(1)$ , considering with what ratio you want to apply this transformation to obtain the minimum number greater than or equal  $min_x$ . Denote found  $x$  through  $lx1$ .

Similarly, we can find an appropriate solution with a maximum  $x = rx1$ , ie  $x \leq max_x$ .

Then move on to the satisfaction of restrictions on  $y$ , ie consideration of the segment  $[min_y; max_y]$ . In the manner described above, will find a solution with the minimum  $y \geq min_y$  and maximum solution  $y \leq max_y$ . We denote the  $x$  coefficients of these solutions through  $lx2$  and  $rx2$  respectively.

Cross the segments  $[lx1; rx1]$  and  $[lx2; rx2]$ ; We denote the resulting cut through  $[lx; rx]$ . It is argued that any decision in which  $x$  the coefficient is in  $[lx; rx]$ - any such decision is appropriate. (This is true in virtue of the construction of this segment: first we separately satisfy the restrictions  $x$  and  $y$ , having two segments, and then crossed them, having an area in which both conditions are met.)

Thus, the number of solutions will be equal to the length of the segment divided by  $|b|$  (since  $x$  the coefficient may be changed only  $\pm b$ ) plus one.

We give implementation (it is difficult to obtain because it requires carefully consider the cases of positive and negative factors  $a$  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;
}

```

Also, it is easy to add to this the conclusions of all the solutions: it is enough to sort out  $x$  in the interval  $[lx; rx]$  with a step  $|b|$ , finding for each of them corresponding  $y$  directly from the 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 restriction, otherwise the answer will almost always negative infinity.

The idea of the solution is the same as in the previous paragraph, first find any solution of the Diophantine equation, and then using the described procedure in the previous paragraph, we arrive at the best solution.

Indeed, we have the right to perform the following transformation (see. The previous paragraph)

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

Note that when this amount  $x + y$  changes as follows:

$$x' + y' = x + y + k \cdot (b/g - a/g) = x + y + k \cdot (b - a)/g.$$

Ie if  $a < b$  it is necessary to choose the smallest possible value  $k$ , if  $a > b$  it is necessary to choose the largest possible value  $k$ .

If  $a = b$  we can not improve the solution - all decisions will have the same amount.

## Tasks in the online judges

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

- SGU # 106 "The Equation" [Difficulty: Medium]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jul 2008 10:35  
EDIT: 8 Sep 2010 21:21

## Modular linear equation of the first order

### Contents [hide]

- Modular linear equation of the first order
  - Statement of the Problem
  - The decision by finding the inverse element
  - Solution using the Extended Euclidean algorithm

### Statement of the Problem

This equation is of the form:

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

where  $a, b, n$ - given integers,  $x$ - unknown integer.

Required to find the desired value  $x$ , which lies in the interval  $[0; n - 1]$  (as on the real line, it is clear there can be infinitely many solutions that are different to each other  $n \cdot k$ , where  $k$ - any integer). If the solution is not unique, then we will see how to get all the solutions.

### The decision by finding the inverse element

Let us first consider the simpler case - where  $a$  and **are relatively prime**. Then we can find **the inverse of** a number , and multiplying on both sides of it, to obtain a solution (and it will be **the only**):

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

Now consider the case  $a$  and **are not relatively prime**. Then, obviously, the decision will not always exist (for example, ). $n2 \cdot x = 1 \pmod{4}$

Suppose  $g = \gcd(a, n)$ , ie their **greatest common divisor** (which in this case is greater than one).

Then, if  $b$  not divisible  $g$ , the solutions do not exist. In fact, if any  $x$  left side of the equation, i.e.  $(a \cdot x) \pmod{n}$ , is always divisible by  $g$ , while the right part it is not divided, which implies that there are no solutions.

If  $b$  divided into  $g$ , then dividing both sides by it  $g$  (ie, dividing  $a, b$  and  $n$  by  $g$ ), we arrive at a new equation:

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

where  $a'$  and  $n'$  are already relatively prime, and this equation we have learned to solve. We denote its solution through  $x'$ .

Clearly, this  $x'$  will also be a solution of the original equation. If, however  $g > 1$ , it is **not the only** solution. It can be shown that the original equation has exactly  $g$  solutions, and they will look like:

$$\begin{aligned}x_i &= (x' + i \cdot n') \pmod{n}, \\i &= 0 \dots (g - 1).\end{aligned}$$

Summarizing, we can say that **the number of solutions** of linear modular equations is either  $g = \gcd(a, n)$  or zero.

## Solution using the Extended Euclidean algorithm

We give our modular equation to Diophantine equation as follows:

$$a \cdot x + n \cdot k = b,$$

where  $x$  and  $k$ - unknown integers.

Way of solving this equation is described in the relevant article [of linear Diophantine equations of the second order](#), and it is in the application of [the Extended Euclidean algorithm](#).

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

---

# MAXimal

home  
algo  
bookz  
forum  
about

## Chinese remainder theorem

### Formulation

In its modern formulation of the theorem is as follows:

Let  $p = p_1 \cdot p_2 \cdot \dots \cdot p_k$ , where  $p_i$ - pairwise relatively prime.

We associate with an arbitrary number of tuple where : $a (0 \leq a < p) (a_1, \dots, a_k) a_i \equiv a \pmod{p_i}$

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

Then this correspondence (between numbers and tuples) will be **one to one**. And, moreover, the operations performed on the number  $a$ , you can be equivalent of the corresponding element of the tuple - by independent operations of 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 AD. Namely, it is shown in the particular solving a system of modular equations and solutions of the 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 numbers  $p_i$  are relatively prime, and a set  $a_1, \dots, a_k$ - an arbitrary set of integers)

### Corollary 2

The consequence is the relationship 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 of  $p_i$  pairwise relatively prime, and  $a$ - an arbitrary integer)

### Garner's algorithm

Of the Chinese remainder theorem, it follows that you can replace operations on the number of operations on tuples. Recall, each number  $a$  is a:  $(a_1, \dots, a_k)$ , where:

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

It can be widely used in practice (in addition to the direct application for the restoration of its residues on the different modules) because we thus long arithmetic operations with an array of "short" numbers. For example, an array of 1000 elements "enough" to the number of approximately 3

### Cor

- Chinese remainder theorem
  - Formulation
    - Corollary 1
    - Corollary 2
  - Garner's algorithm
  - Implementation of the

$p$  the first  $-s$  1000 simple); and if selected as the  $p_i$ s simple about a billion, then enough already with the number of about 9000 signs. But, of course we learn how to **restore** the number  $a$  on the motorcade. From Corollary 1 shows that a recovery is possible, and only one (provided  $0 \leq a < p_1 \cdot \dots \cdot p_k$ ). **algorithm** is an algorithm that allows to perform this restoration, and quite effectively.

We seek a solution in the form of:

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

ie in a mixed radix digits with weights  $p_1, p_2, \dots, p_k$ .

We denote by  $r_{ij}$  ( $i = 1 \dots k-1, j = i+1 \dots k$ ) numbers is the inverse  $p_i$  modulo  $p_j$  (finding the inverse elements in the ring modulo descr)

$$r_{ij} = (p_i)^{-1} \pmod{p_j}.$$

Substituting the expression  $a$  in a mixed radix in the first equation, we get:

$$a_1 \equiv x_1.$$

We now substitute in the second equation:

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

We transform this expression by taking away of both sides  $x_1$  and dividing by  $p_1$ :

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

Substituting into the third equation, we obtain a similar manner:

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

Already clearly visible pattern, which is the easiest way to express 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  $x_i$  of the time  $O(k^2)$ , the very same answer - number  $a$ - can be restored by the formula:

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

It is worth noting that in practice almost always need to calculate the answer using [the Long arithmetic](#), but the coefficients themselves  $x_i$  are stored in types, but because the whole algorithm Garner is very effective.

## Implementation of the algorithm Garner

The most convenient way to implement this algorithm in the language of Java, because it contains a standard length arithmetic, and therefore the transfer of the modular system of the usual number (using a standard class BigInteger).

The below Garner implementation of the algorithm supports addition, subtraction and multiplication, with support work with negative numbers (not after the code). Implemented transfer of conventional desyatichkogo presentation in a modular system and vice versa.

In this example, taken 100 after a simple 10<sup>9</sup>, allowing you to work with numbers up to about 10<sup>900</sup>.

```
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;
}
}

```

On the support **negative** numbers deserves mention (flag `can_be_negative`functions `bigIntegerValue()`). The very modular scheme does no between positive and negative numbers. However, it can be seen that if a specific task response do not exceed half of the product of all primes, be different from the negative that receive less positive numbers this means, and negative - more. Therefore, we Garner after classical algorithm with the middle and if it is, then we derive a minus, and invert the result (ie, subtract it from the product of all primes, and outputs it already).

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 9 Sep 2008 19:28  
EDIT: 20 Sep 2010 18:45

## Factorial calculation modulo

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

Thus, formally **the problem** was. Required to compute  $n!$  modulo a prime  $p$ , while not taking into account all the multiple  $p$  factors that are included in the factorial. By learning to effectively compute a factorial, we can quickly compute the value of different combinatorial formulas (eg, [binomial coefficients](#) ).

### Contents [hide]

- Factorial calculation modulo
  - Algorithm
  - Implementation

## Algorithm

Let us write this "modified" factorial explicitly:

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

If such a record shows that the "modified" factorial divided into several blocks of length  $p$  (the last block may be shorter) that are identical except for the last element:

$$\begin{aligned} n! \% p &= \underbrace{1 \cdot 2 \cdots \cdot (p-2) \cdot (p-1)}_{1st} \cdot \underbrace{1 \cdot 2 \cdots \cdot (p-1)}_{2nd} \cdot \underbrace{2 \cdots 1 \cdot 2 \cdots \cdot (p-1) \cdot 1}_{p-th} \cdots \\ &\quad \cdot \underbrace{1 \cdot 2 \cdots \cdot (n \% p)}_{tail} \pmod{p}. \end{aligned}$$

The total count of the blocks is easy - it's just  $(p-1)!$  mod  $p$  that you can find the software or by Theorem Wilson (Wilson) immediately find  $(p-1)!$  mod  $p = p-1$ . To multiply the common parts of blocks must be obtained value raised to the power mod  $p$  that can be done for the  $O(\log n)$  operations (see. [The binary exponentiation](#) ; however, you can see that we actually erect minus one to some degree, and therefore the result there will always be either 1 or  $p-1$ , depending on the parity of the index. The value of the last, incomplete block also can be calculated separately for  $O(p)$ . Only the last elements of the blocks, consider them carefully:

$$n! \% p = \underbrace{\cdots \cdot 1}_{\text{tail}} \cdot \underbrace{\cdots \cdot 2}_{\text{tail}} \cdot \underbrace{\cdots \cdot 3}_{\text{tail}} \cdots \cdot \underbrace{\cdots \cdot (p-1)}_{\text{tail}} \cdot \underbrace{\cdots \cdot 1}_{\text{tail}} \cdot \underbrace{\cdots \cdot 1}_{\text{tail}} \cdot \underbrace{\cdots \cdot 2}_{\text{tail}} \cdots$$

And again we come to the "modified" factorial, but smaller dimension (as much as it was full of blocks, and they were  $\lfloor n/p \rfloor$ ). Thus, the calculation of the "modified" factorial  $n! \% p$  we reduced for  $O(p)$  operations to the calculation already  $(n/p)! \% p$ . Revealing this recurrence relation, we find that the depth of recursion is  $O(\log_p n)$ , total **asymptotic behavior** of the algorithm is obtained  $O(p \log_p n)$ .

## Implementation

It is clear that the implementation is not necessary to use recursion explicitly: because the 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  $O(p \log_p n)$ .

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jul 2008 13:58  
EDIT: 27 May 2012 17:55

## Finding degree divider factorial

Given two numbers:  $n$  and  $k$ . Required to calculate with any degree of the divisor  $k$  is one of  $n!$ , ie find the greatest  $x$  such that  $n!$  is divided into  $k^x$ .

### Contents [hide]

- Finding degree divider factorial
  - Solution for the case of simple  $k$
  - Solution for the case of the compound  $k$

### Solution for the case of simple $k$

Consider first the case where  $k$  simple.

Let us write the expression for the factorial explicitly:

$$n! = 1 \ 2 \ 3 \ \dots \ (n-1) \ n$$

Note that each  $k$ th member of this work is divided into  $k$ , ie allows one to account; the number of such members of the same  $\lfloor n/k \rfloor$ .

Further, we note that each  $k^2$ th term of this series is divided into  $k^2$ , ie gives one more to the answer (given that  $k$  in the first degree has been considered previously); the number of such members of the same  $\lfloor n/k^2 \rfloor$ .

And so on, every  $k^i$ th term of the series gives one to answer, and the number of members equal  $\lfloor n/k^i \rfloor$ .

Thus, the magnitude of response is:

$$\frac{n}{k} + \frac{n}{k^2} + \dots + \frac{n}{k^i} + \dots$$

This amount, of course, is not infinite, because Only the first of about  $\log_k n$  members nonzero. Consequently, the asymptotic behavior of the algorithm is  $O(\log_k n)$ .

Implementation:

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

```

    }
    return res;
}

```

## Solution for the case of the compound $k$

The same idea is applied directly anymore.

But we can be factored  $k$ , to solve the problem for each of its prime divisors, and then select the minimum of the answers.

More formally, let  $k_i$ - is  $i$ th divisor of  $k$  entering it in the degree  $p_i$ . To solve the problem  $k_i$  using the above formula for  $O(\log n)$ ; even if we got an answer  $\text{Ans}_i$ . Then the answer for the composite  $k$  will be a minimum of values  $\text{Ans}_i/p_i$ .

Given that the factorization is performed in the simplest way  $O(\sqrt{k})$ , we obtain the asymptotic behavior of the final  $O(\sqrt{k})$ .

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 16 Jan 2009 0:58  
EDIT: 25 Apr 2012 3:11

## Through all this mask subpatterns

### Bust subpatterns fixed mask

Dana bitmask  $m$ . Required to effectively sort out all its subpatterns, ie such masks  $s$ , which can 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) {
    ... МОЖНО ИСПОЛЬЗОВАТЬ s ...
    s = (s-1) & m;
}
```

or using a more compact operator *for*:

```
for (int s=m; s; s=(s-1)&m)
    ... МОЖНО ИСПОЛЬЗОВАТЬ s ...
```

The only exception to the two versions of the code - subpattern, which is equal to zero, will not be processed. Her treatment will have to take out of the loop, or use a less elegant design, for example:

```
for (int s=m; ; s=(s-1)&m) {
    ... МОЖНО ИСПОЛЬЗОВАТЬ s ...
    if (s==0) break;
}
```

Let us examine why the code above really finds all subpatterns this mask, with no repetitions, in O (number), and in descending order.

Suppose we have the current subpattern  $s$ , and we want to go to the next subpattern. Subtract from the mask  $s$  unit, thus we will remove the rightmost single bit and all the bits to the right to put it in 1. Next, remove all the "extra" one bits that are not included in the mask  $m$  and therefore can not be included in the

### Contents [hide]

- Through all this mask subpatterns
  - Bust subpatterns fixed mask
  - Through all the masks with their subpatterns. Assessment  $3^n$

subpattern. Removing the bit operation is performed  $\&m$ . As a result, we are "cut off the" mask  $s - 1$  before the greatest importance that it can take, ie, until the next subpattern after  $s$  in descending order.

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

Particularly consider when  $s = 0$ . After doing  $s - 1$  we will get the mask, which included all the bits (bit representation of the number  $-1$ ), and after removing the extra bit operations  $(s - 1) \& m$  will not nothing but a mask  $m$ . Therefore, the 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. Assessment $3^n$

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

```
for (int m=0; m<(1<<n); ++m)
    for (int s=m; s; s=(s-1) &m)
        ... использование s и m ...
```

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

**Proof: 1 way** . Consider  $i$ th bit. For him, generally speaking, there are exactly three possibilities: he is not part of the mask  $m$  (and therefore subpattern  $s$ ); it is included in  $m$ , but is not included  $s$ ; it enters  $m$  in  $s$ . All bits  $n$ , so all the different combinations will  $3^n$ , as required.

**Proof: 2 way** . Note that if the mask  $m$  is  $k$  included bits, it will have  $2^k$  subpatterns. Since the length of the mask  $n$  with  $k$  bits have enabled  $C_n^k$  (see. "binomial coefficients" ), then all combinations will be:

$$\sum_{k=0}^n C_n^k 2^k.$$

Calculate this amount. To do this, we note that it is not nothing but a degradation in the binomial theorem expression  $(1 + 2)^n$ , ie  $3^n$ , as required.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 16 Jan 2009 0:58  
EDIT: 20 Aug 2012 23:51

## Primitive roots

### Determination

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

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

### Contents [hide]

- Primitive roots
  - Determination
  - Existence
  - Communication with the function of Euler
  - Algorithm for finding primitive roots
  - Implementation

### Existence

A primitive root modulo  $n$  exists if and only if there  $n$  is a degree of odd prime, or twice the power of a prime, and in cases  $n = 1, n = 2, n = 4$ .

This theorem (which was completely proved by Gauss in 1801) is given here without proof.

### Communication with the function of Euler

Let  $g$ - primitive root modulo  $n$ . Then we can show that the smallest number  $k$  for which  $g^k \equiv 1 \pmod{n}$  (ie  $k$ - an indicator  $g$ (multiplicative order)), the same  $\phi(n)$ . Moreover, the opposite is true, and this fact will be used later in our algorithm for finding primitive roots.

Furthermore, if the modulus  $n$  has at least one primitive root, the total of  $\phi(\phi(n))$  (a cyclic group as  $k$  elements the  $\phi(k)$  generator).

### Algorithm for finding primitive roots

A naive algorithm would require values for each test of time to calculate all its powers to check that they are all different. This algorithm is too slow, below we are using several well-known theorems of number theory a faster algorithm.  $O(n)$

Above, we present a theorem that if the smallest number  $k$  for which  $g^k \equiv 1 \pmod{n}$  (ie  $k$ - an indicator  $g$ ) as well  $\phi(n)$ , then  $g$ - a primitive root. Since for any number  $a$  relatively prime to  $n$ , performed Euler's theorem ( $a^{\phi(n)} \equiv 1 \pmod{n}$ ), then to verify that the  $g$  primitive root, it suffices to check that for all numbers  $d$  smaller  $\phi(n)$ , performed

$g^d \not\equiv 1 \pmod{n}$ . However, while it is too slow algorithm.

Of Lagrange's theorem implies that the rate of any number modulo  $n$  a divisor  $\phi(n)$ . Thus, it suffices to check that for all proper divisors  $d \mid \phi(n)$  is performed  $g^d \not\equiv 1 \pmod{n}$ . It is already much faster algorithm, but you can go even further.

Factor the number  $\phi(n) = p_1^{a_1} \dots p_s^{a_s}$ . We prove that in the previous algorithm is sufficient to consider as  $d$  a number of species  $\frac{\phi(n)}{p_i}$ . Indeed, suppose that  $d$ - any proper divisor  $\phi(n)$ .

Then, obviously, there exists such  $j$  that  $d \mid \frac{\phi(n)}{p_j}$ , ie  $d \cdot k = \frac{\phi(n)}{p_j}$ . However, if  $g^d \equiv 1 \pmod{n}$  we would get:

$$g^{\frac{\phi(n)}{p_j}} \equiv g^{d \cdot k} \equiv (g^d)^k \equiv 1^k \equiv 1 \pmod{n},$$

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

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

Time of the algorithm (assuming that the number  $\phi(n)$  has  $O(\log \phi(n))$  divisors, and exponentiation algorithm is executed [binary exponentiation](#), ie  $O(\log n)$ ) is  $O(\text{Ans} \cdot \log \phi(n) \cdot \log n)$  plus time factorization of  $\phi(n)$  where  $\text{Ans}$ - the result, ie, value of the unknown primitive root.

About the growth rate of the growth of primitive roots  $n$  are known only rough estimates. It is known that primitive roots - a relatively small amount. One of the well-known estimates - evaluation Shupa (Shoup), which, assuming the truth of the Riemann hypothesis, there is a primitive root  $O(\log^6 n)$ .

## Implementation

Function `powmod()` performs a binary exponentiation modulo a function generator (`int p`) - is a primitive root modulo  $p$ (number factorization  $\phi(n)$  is performed for the simplest algorithm  $O(\sqrt{\phi(n)})$ ).

To adapt this function to arbitrary  $p$ , just add the computation [of the Euler function](#) in a variable `phi`, as well as filter out `res` non-prime to  $n$ .

```

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;
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 16 Jan 2009 0:58  
EDIT: 16 Jan 2009 10:04

## Discrete root extract

The problem of discrete root extraction (similar to the [discrete logarithm problem](#)) is as follows. According to  $n$  ( $n$ - simple)  $a$ , you want to find all  $x$  satisfying:

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

### Contents [hide]

- Discrete root extract
  - Algorithm for solving
  - Finding all solutions, knowing one of them
  - Implementation

## Algorithm for solving

We will solve the problem by reducing it to the discrete logarithm problem.

For this, we apply the concept of a [primitive root modulo  \$n\$](#) . Let  $g$ - primitive root modulo  $n$  (since  $n$ - just that it exists). We can find it, as described in the relevant article of the  $O(\text{Ans} \cdot \log \phi(n) \cdot \log n) = O(\text{Ans} \cdot \log^2 n)$  time plus the number factorization  $\phi(n)$ .

Discard immediately the case when  $a = 0$ - in this case immediately find the answer  $x = 0$ .

Since in this case ( $n$ - simple) any number from 1 to  $n - 1$  be represented as a primitive root of degree, the problem of discrete root, we can be represented as:

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

where

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

Trivial transformations we obtain:

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

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

Suppose we have found a solution to  $y$  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  $x_0 = g^{y_0} \pmod{n}$  to find all the other solutions.

To do this, remember is the fact that a primitive root always has the order  $\phi(n)$  (see. [article on primitive root](#)), ie, the lowest degree  $g$ , which gives the unit is  $\phi(n)$ . Therefore, the addition of a term with exponent  $\phi(n)$  does not change anything:

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

Hence, all solutions are of the form:

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

which  $l$  is chosen so that the fraction  $\frac{l \cdot \phi(n)}{k}$  was intact. To this fraction was intact, the numerator must be a multiple of the least common multiple  $\phi(n)$  and  $k$  where (remembering 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 that gives a general view of all the solutions of the discrete root.

## Implementation

We give full implementation, including finding a primitive root, and finding the discrete logarithm and the withdrawal of all decisions.

```

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]);
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jul 2008 19:33  
EDIT: 8 Sep 2010 17:16

## Balanced ternary number system

Balanced ternary number system - a non-standard positional number system. The base system is equal 3, but it differs from the usual ternary system the fact that the figures are  $-1, 0, 1$ . Since the use  $-1$  of a single digit is very uncomfortable, it usually takes some special notation. Conditions are denoted by minus one letter  $z$ .

For example, the number 5 in the ternary system is written as balanced 1zz, and the number  $-5$  - how z11. Balanced ternary number system allows you to record negative numbers without writing a single "minus" sign. Ternary balanced system allows fractional numbers (for example,  $1/3$  written as 0.1).

### Contents [hide]

- Balanced ternary number system
  - Translation algorithm

## Translation algorithm

Learn to translate numbers into a balanced ternary system.

To do this you must first translate the numbers in the ternary system.

It is clear that now we have to get rid of the digits 2, which we note that  $2 = 3 - 1$ , ie we can replace the two in the current discharge on  $-1$ , while increasing the next (ie, to the left of him in a natural writing) on the discharge 1. If we move from right to left to write and perform the above operation (in this case in some discharges may occur more overflow 3, in this case, of course, "reset" extra triples in the MSB), then 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 a number in the ternary notation, we add to it an infinite number  $\dots 11111.11111 \dots$ , then each bit of the result subtract one (already without any hyphens).

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

## BPSW test for primality

### Contents [hide]

- BPSW test for primality
  - Introduction
  - Brief description
  - Implementation of algorithms in this article
  - Miller-Rabin test
  - Strong test Lucas-Selfridge
    - Algorithm Selfridge
    - Strong algorithm Lucas
    - Discussion of the algorithm Selfridge
    - The implementation of a strong algorithm Lucas-Selfridge
  - Code BPSW
  - Quick implementation
  - Heuristic proof-refutation Pomerance
  - Practical tests test BPSW
    - The average operating time on the segment numbers, depending on the limit of trivial enumeration
    - The average operating time on the segment numbers
  - Application. All programs
  - Literature

### Introduction

Algorithm BPSW - this is a test of the simplicity. This algorithm is named for its inventors: Robert Bailey (Ballie), Carl Pomerance (Pomerance), John Selfridge (Selfridge), Samuel Wagstaff (Wagstaff). The algorithm was proposed in 1980. To date, the algorithm was not found a single counterexample, as well as there was no evidence found.

BPSW algorithm was tested on all numbers 10 to  $10^{15}$ . In addition, trying to find a counterexample using PRIMO (cm. [6]), based on simple test using elliptic curves. Program, having worked for three years, did not find any counterexample, based on which Martin suggested that there is no single BPSW-pseudosimple, less  $10^{10,000}$  (Pseudoprime - a composite number on which algorithm gives the result a "simple"). At the same time, Carl Pomerance in 1984 presented a heuristic proof that there are infinitely many BPSW-pseudosimple numbers.

The complexity of the algorithm BPSW have  $O(\log^3(N))$  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 completely 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 is as follows:

1. Run the test of Miller-Rabin base 2.
  2. Run a strong test of Lucas-Selfridge using Lucas sequence with parameters Selfridge.
  3. Return the "simple" only when both tests returned "simple".
0. In addition, at the beginning of the algorithm, you can add a check for trivial divisors of, say, up to 1000. This will increase the speed of work on the composite number, however, has slowed somewhat simple algorithm.

Thus BPSW algorithm based on the following:

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

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

## Implementation of algorithms in this article

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

The algorithms are implemented using templates (templates), which allows their use as a built-in numeric types, as well as its own class that implements a long arithmetic. [Long long arithmetic article is not included - TODO]

In the article itself will be given only the most essential functions, the texts of the same auxiliary functions can be downloaded in the annex to the article. Here we give only the headers of these functions together with comments:

```

//! Module 64-bit number
Long Long 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 - exact square of a prime number
template <class T>
bool perfect_square (const T & n);

//! Calculates the square root of the number, rounding it down
template <class T>
T sq_root (const T & n);

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

//! Returns the value of the k-th 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);

//! Calculates a ^ k (mod n)
template <class T, class T2>
T powmod (T A, T2 k, const T & n);

//! Puts 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>
T gcd (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<T> get_primes (const T & b, T2 & PI);

```

```

 $\text{/// A trivial verification n its simplicity, get over all divisors of up to m.}$ 
 $\text{/// Result: 1 - if n is just a simple, p - it found divider, 0 - if unknown}$ 
 $\text{template <class T, class T2>}$ 
 $\text{T2 prime_div_trivial (const \& T n, m T2);}$ 

```

---

## 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 is  $O(\log^3(N))$  bit operations, and give ready implementation of this algorithm:

```

 $\text{template <class T, class T2>}$ 
 $\text{bool miller_rabin (T n, T2 b)}$ 
 $\{$ 

     $\text{// First check the trivial cases}$ 
     $\text{if (n == 2)}$ 
         $\text{return true;}$ 
     $\text{if (n < 2 || even (n))}$ 
         $\text{return false;}$ 

     $\text{// Check that n and b are relatively prime (otherwise it will cause an error)}$ 
     $\text{// If they are not relatively prime, then either n is not just a need to increase the b}$ 
     $\text{if (b < 2)}$ 
         $\text{b = 2;}$ 
     $\text{for (T g; (g = gcd (n, b)) != 1; ++ b)}$ 
         $\text{if (n > g)}$ 
             $\text{return false;}$ 

     $\text{// Decompose n-1 = q * 2 ^ p}$ 
     $\text{T n_1 = n;}$ 
     $\text{--n_1;}$ 
     $\text{T p, q;}$ 
     $\text{transform_num (n_1, p, q);}$ 

     $\text{// Compute b ^ q mod n, if it is 1 or n-1, n is a prime (or pseudosimple)}$ 
     $\text{T rem = powmod (T (b), q, n);}$ 
     $\text{if (rem == 1 || rem == n_1)}$ 
         $\text{return true;}$ 

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

     $\text{return false;}$ 
 $\}$ 

```

---

## 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 five sequences, -7, 9, -11, 13, ... to find the number of the first D, for which  $J(D, N) = -1$  and  $\gcd(D, N) = 1$ ,

where  $J(x, y)$  - Jacobi symbol.

**Selfridge parameters** are  $P = 1$  and  $Q = (1 - D) / 4$ .

Note that there is no parameter Selfridge for numbers that are precise 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 Selfridge wrong for even numbers and units; however, verification of these cases is not difficult.

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

Finally, we note that if  $D$  for some number  $N$  is too large, the algorithm is computationally would be inapplicable. Although in practice this has not been noticed (it appears quite enough 4-byte number), though the probability of this event should not be excluded. However, for example, in the interval  $[1; 10^6]$   $\max(D) = 47$ , and in the interval  $[10^{19}; 10^{19} \cdot 10^6]$   $\max(D) = 67$ . Furthermore, in Bailey and Wagstaff 1980 analytically proved that observation (see. Ribenboim, 1995/96, p. 142).

## Strong algorithm Lucas

**Algorithm parameters** are the number of Lucas **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** - a sequence of  $U_k$  and  $V_k$ , defined as follows:

$$\begin{aligned} U_0 &= 0 \\ U_1 &= 1, \\ U_k &= PU_{k-1} - QU_{k-2} \\ V_0 &= 2 \\ V_1 &= P \\ V_k &= PV_{k-1} - QV_{k-2} \end{aligned}$$

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

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

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

In particular, the parameters  $D, P, Q$  are calculated algorithm Selfridge, we have:

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

The converse is not true in general. However, pseudosimple numbers for a given algorithm is not very much, on what, in fact, based algorithm Lucas.

Thus, the **algorithm is calculating Lucas  $U_M$  and comparing it with zero**.

Next, you need to find some way to accelerate compute  $U_K$ , otherwise, of course, no practical sense in this algorithm would not be.

We have:

$$\begin{aligned} U_k &= (A^k - b^k) / (A - b), \\ V_k &= A^k + b^k, \end{aligned}$$

where  $a$  and  $b$  - different roots of the quadratic equation  $x^2 - Px + Q = 0$ .

Now we can prove the following equation is simple:

$$\begin{aligned} U_{2K} &\equiv U_k V_k \pmod{N} \\ V_{2K} &\equiv k^2 - 2Q^k \pmod{N} \end{aligned}$$

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

$$U_M \equiv U_E V_E V_{2E} V_{4E} \dots V_{2^{T-2}E} V_{2^{T-1}E} \equiv 0 \pmod{N},$$

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

It is understood that it suffices to calculate  $U_E$  and  $V_E$ , and all subsequent multipliers  $V_{2E}$   $V_{4E}$  ...  $V_{2^{T-2}E}$   $V_{2^{T-1}E}$  can already receive from them .

Thus, it remains to learn quickly calculate  $U_E$  and  $V_E$  for odd E.

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

$$\begin{aligned} U_{+j}i &= (U_i V_j + U_j V_i) / 2 \pmod{N} \\ V_{+j}i &= (V_i V_j + D U_i U_j) / 2 \pmod{N} \end{aligned}$$

Note that the division is performed in the field ( $\pmod{N}$ ).

These formulas are proved very simple, and here is their proof is omitted.

Now, having the formulas for addition and doubling members Lucas sequences, concepts and methods to accelerate compute  $U_E$  and  $V_E$  .

Indeed, consider the binary representation of the number of E. Let us first result -  $U_E$  and  $V_E$  - equal, respectively,  $U_1$  and  $V_1$  . Walk into all bits of E from younger to older, skipping only the first bit (the initial term of the sequence). For each i-th bit will calculate  $U_{2^i}$  and  $V_{2^i}$  of the previous members, by doubling formulas. Furthermore, if the current i-th bit equal to one, then the answer will be added to the current  $U_{2^i}$  and  $V_{2^i}$  using addition formulas. At the end of the algorithm that runs in  $O(\log(E))$ , we obtain the desired  $U_E$  and  $V_E$  .

If  $U_E$   $V_E$  were zero ( $\pmod{N}$ ), then N is prime number (or pseudosimple). If they are both different from zero, then calculate  $V_{2E}$  ,  $V_{4E}$  , ...  $V_{2^{T-2}E}$  ,  $V_{2^{T-1}E}$  . If at least one of them is comparable with 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 to get and which is the algorithm Selfridge.

Recall the basic requirements for parameters:

$$\begin{aligned} P > 0, \\ D = P^2 - 4 * Q \neq 0. \end{aligned}$$

Now continue the study of these parameters.

**D should not be a perfect square ( $\pmod{N}$ ) .**

Indeed, otherwise we get:

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

Ie if D - perfect square, then the algorithm becomes almost Lucas usual 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, let P = 1. Then  $Q = (1 - D) / 4$ . The method proposed Selfridge.

However, there are other methods of selecting can select his D. sequence of 5, 9, 13, 17, 21, ... Also, let P - 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 differ 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.

## The implementation of 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 the 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 gives an error
if (perfect_square (n))
    return false;

// Algorithm Selfridge: find the first number d such that:
// Jacobi (d, n) = - 1 and it belongs to a number 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)
        // Found divider - d_abs
        return false;
    if (jacobi (T (dd), n) == -1)
        break;
}

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

// Expand the 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 a 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

It now remains only to combine the results of all three tests: checking for small trivial divisors, Miller-Rabin test, a strong test of Lucas-Selfridge.

```

template <class T>
bool baillie_pomerance_selfridge_wagstaff (T n)
{

    // First check for trivial divisors - for example, up to 29
    int div = prime_div_trivial (n, 29);
    if (div == 1)
        return true;
    if (div> 1)
        return false;

    // Miller-Rabin test base 2
    if (! miller_rabin (n, 2))
        return false;

    // Strong test of Lucas-Selfridge
    return lucas_selfridge (n, 0);
}

```

```
}
```

[From here you can download a program \(source + exe\), containing the full realization of the test BPSW. \[77 KB\]](#)

## Quick implementation

Code length can be significantly reduced at the expense of flexibility, giving up 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,
                  t3 = (v2m * v)% n, t4 = (((u2m * 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;
        if (v> = n) v -= n;
        if (v> = n) v -= 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 should 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 is greater than  $X^{1-a}$  for any  $a > 0$ .

Proof.

Let  $k > 4$  - arbitrary but fixed number. Let  $T$  - a large number.

Let  $P_k(T)$  - the set of primes  $p$  in the interval  $[T; T^k]$ , 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 ordinary  $q$ , as  $q \equiv 1 \pmod{4}$
- (5) the number of  $(p+1)/4$  is not a perfect square
- (6) the number  $(p+1)/4$  composed exclusively of ordinary  $d < T$
- (7) The number  $(p+1)/4$  composed solely of ordinary  $d$ , such that  $q \equiv 3 \pmod{4}$

It is understood that about 1/8 of all prime in the interval  $[T; T^k]$  satisfies the condition (1). You can also show that the conditions (2) and (5) retain some of the numbers. Heuristically conditions (3) and (6) also allow us to leave some of the numbers from the interval  $(T; T^k)$ . Finally, the event (4) has a probability  $(C(\log T)^{-1/2})$ , as well as an event (7). Thus, the cardinality of  $P_k(T)$  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  $n$ , which is not a perfect square, made up of simple  $l$  of  $P_k(T)$ , where  $l$  is odd and less than  $T^2 / \log(T^k)$ . Number of ways to choose a number  $n$  is approximately

$$\binom{[cT^k / \log^2 T]}{\ell} > e^{T^2(1-3/k)}$$

for large  $T$  and a fixed  $k$ . In addition, each number is less than  $n e^{T^2}$ .

Let  $Q$  be one product of prime  $q < T$ , for which  $q \equiv 1 \pmod{4}$ , and by  $Q_3$  - a product of prime  $q < T$ , for which  $q \equiv 3 \pmod{4}$ . Then  $\gcd(Q_1, Q_3) = 1$  and  $Q_1 Q_3 \leq e^T$ . Thus, the number of ways of choosing  $n$  additional terms

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

be heuristically at least

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

for large  $T$ .

But every such  $n$  - is a counterexample to the test BPSW. Indeed,  $n$  is the number of Carmichael (ie the date on which Miller-Rabin test is wrong with any ground), so it will automatically be pseudosimple to base 2. Since  $n \equiv 3 \pmod{8}$  and each  $p | n \equiv 3 \pmod{8}$ , it is obvious that also  $n$  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 any 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 the big  $T$ , will be at least  $e^{T^2(1-4/k)}$  counterexamples to test BPSW among numbers less than  $e^{T^2}$ . Now, if we put  $x = e^{T^2}$ , will be at least  $x^{1-4/k}$  counterexamples less than  $x$ . Since  $k$  -

random number, then our proof means that **the number of counterexamples, smaller x, there is a number greater than  $x^{1-a}$  for any  $a > 0$** .

## Practical tests test BPSW

This section will examine the results obtained from testing my test implementation BPSW. All tests were carried out on the internal type - including a 64-bit long long. Long arithmetic untested.

Testing was conducted on a computer with processor Celeron 1.3 GHz.

All times are in **microseconds** ( $10^{-6}$  sec).

### The average operating time on the segment numbers, depending on the limit of trivial enumeration

This refers to the parameter passed to the prime\_div\_trivial (), which in the above code is 29.

[Download a test program \(source code and exe-file\). \[83 KB\]](#)

If you run a test **on all the odd numbers** from the interval, the results turn out to be:

beginning of the segment	end segment	limit> sorting>	0	$10^2$	$10^3$	$10^4$	$10^5$
1	$10^5$		8.1	4.5	0.7	0.7	0.9
$10^6$	$10^6 10^5$		12.8	6.8	7.0	1.6	1.6
$10^9$	$10^9 10^5$		28.4	12.6	12.1	17.0	17.1
$10^{12}$	$10^{12} 10^5$		41.5	16.5	15.3	19.4	54.4
$10^{15}$	$10^{15} 10^5$		66.7	24.4	21.1	24.8	58.9

If the test is run **only by primes** of the segment, the speed is as follows:

beginning of the segment	end segment	limit> sorting>	0	$10^2$	$10^3$	$10^4$	$10^5$
1	$10^5$		42.9	40.8	3.1	4.2	4.2
$10^6$	$10^6 10^5$		75.0	76.4	88.8	13.9	15.2
$10^9$	$10^9 10^5$		186.5	188.5	201.0	294.3	283.9
$10^{12}$	$10^{12} 10^5$		288.3	288.3	302.2	387.9	1069.5
$10^{15}$	$10^{15} 10^5$		485.6	489.1	496.3	585.4	1267.4

Thus, optimally choose **the trivial limit busting at 100 or 1000**.

For all the following tests, I chose within 1000.

### The average operating time on the segment numbers

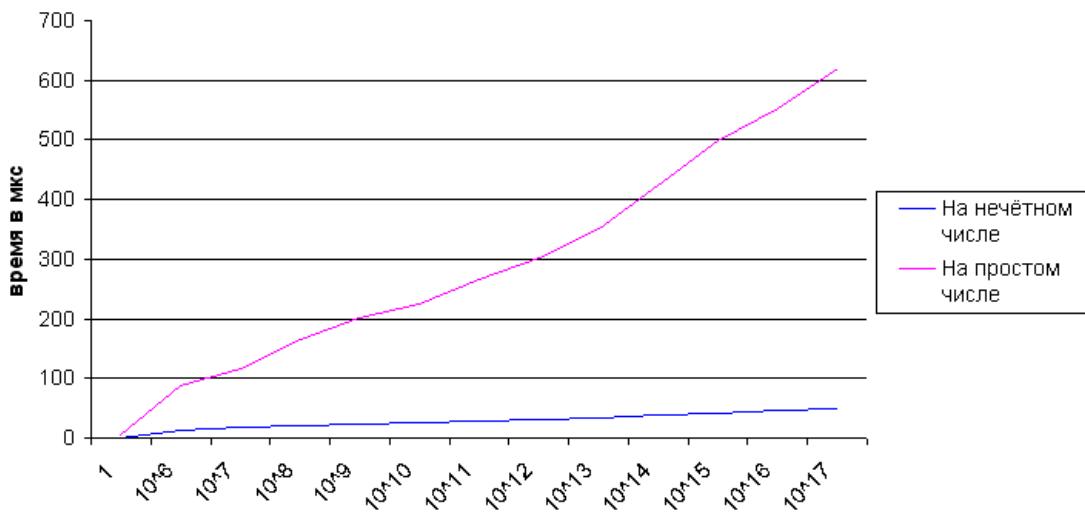
Now that we have chosen limit trivial enumeration can be more accurately test the speed at various intervals.

[Download a test program \(source code and exe-file\). \[83 KB\]](#)

beginning of the segment	end segment	while working at odd numbers	while working on the prime numbers
1	$10^5$	1.2	4.2
$10^6$	$10^6 10^5$	13.8	88.8
$10^7$	$10^7 10^5$	16.8	115.5
$10^8$	$10^8 10^5$	21.2	164.8
$10^9$	$10^9 10^5$	24.0	201.0

$10^{10}$	$10^{10} 10^5$	25.2	225.5
$10^{11}$	$10^{11} 10^5$	28.4	266.5
$10^{12}$	$10^{12} 10^5$	30.4	302.2
$10^{13}$	$10^{13} 10^5$	33.0	352.2
$10^{14}$	$10^{14} 10^5$	37.5	424.3
$10^{15}$	$10^{15} 10^5$	42.3	499.8
$10^{16}$	$10^{16} 10^5$	46.5	553.6
$10^{17}$	$10^{17} 10^5$	48.9	621.1

Or, in the form of a graph, the approximate time of the test on one BPSW including:



That is, we have found that in practice, a small number ( $10$  to  $17$ ), **the algorithm works for  $O(\log N)$** . This is because the built-in type int64 for the division operation is performed in  $O(1)$ , i.e. fission zavisit complexity of the number of bits in number.

If we apply the test to BPSW long arithmetic, it is expected that it will work just for the  $O(\log^3(N))$ . [TODO]

## Application. All programs

Download all programs of this article. [242 KB]

## Literature

I used 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](http://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; HW Lenstra **Primality Testing and Jacobi Sums** Amsterdam (1984)  
[www.openaccess.leidenuniv.nl/bitstream/1887/2136/1/346\\_065.pdf](http://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](http://www.ellipsa.net)
  7. F. Morain **Elliptic curves and Primality Proving** Math. Comp. 61 (203)
  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/Baillie-PSWPrimalityTest.html](http://mathworld.wolfram.com/Baillie-PSWPrimalityTest.html)
  10. Eric W. Weisstein **Strong Lucas pseudoprime** MathWorld (2005)  
[mathworld.wolfram.com/StrongLucasPseudoprime.html](http://mathworld.wolfram.com/StrongLucasPseudoprime.html)
  11. Paulo Ribenboim **The Book of Prime Number Records** Springer-Verlag (1989) [without reference]

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

### Contents [\[hide\]](#)

- Efficient algorithms for factorization
  - Method Pollard p-1
  - Pollard's method "Ro"
  - Bent method (modified method of Pollard "Ro")
  - Pollard method Monte Carlo
  - Method Ferm
  - Trivial division
  - Putting it all together
  - Application

Here are implementing several factorization algorithms, each of which individually may not work as quickly or very slowly, but together they provide a very fast method.

Descriptions of these methods are given, the more that they are quite well documented on the internet.

### Method Pollard p-1

Probability test, quickly gives the answer is not for all numbers.

Returns or found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_p_1 (T n)
{
    // Algorithm parameters significantly affect the performance and quality of search
    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++)
    {

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

        // Compute 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 if 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 "Ro"

Probability test, quickly gives the answer is not for all numbers.

Returns or 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 (modified method of Pollard "Ro")

Probability test, quickly gives the answer is not for all numbers.

Returns or 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 method Monte Carlo

Probability test, quickly gives the answer is not for all numbers.

Returns or 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 is one hundred percent method, but it can be very slow if there are small number of dividers.

Therefore, it should run only after all other methods.

```

template <class T, class T2>
T fermat (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? xy: 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 all the very small divisors.

```

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

```

```

        return 2;

    // Generate 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 the methods in one function.

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

```

template <class T, class T2>
void factorize (const T & n, std :: map <T, unsigned> & result, T2 unused)
{
    if (n == 1)
        ;
    else
        // Check if the number is not a simple
        if (isprime (n))
            ++ Result [n];
        else
            // If the number is small enough, it decompose 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 factorization algorithms
                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% algorithm Farm
                if (div == 1)
                    div = ferma (n, unused);
                // Recursively Point multipliers
                factorize (div, result, unused);
                factorize (n / div, result, unused);
            }
}

```

---

## Application

[Download \[5k\]](#) source program that uses all these methods of factorization and test BPSW at ease.

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 10 Jun 2008 19:04  
EDIT: 9 Nov 2012 12:38

## Fast Fourier transform of O (N log N). Application to the multiplication of two polynomials or long numbers

Here we consider an algorithm that allows to multiply two polynomials of length  $n$  of time  $O(n \log n)$  that is much better time  $O(n^2)$ , achieved a trivial multiplication algorithm. It is obvious that the multiplication of two long numbers can be reduced to multiplication of polynomials, so the two long numbers can also be multiplied over time  $O(n \log n)$ .

The invention Fast Fourier Transform attributed Cooley (Coolet) and Tukey (Tukey) - 1965. In fact, the FFT has repeatedly invented before, but its importance was not fully realized until the advent of modern computers. Some researchers credited with the discovery of the FFT Runge (Runge) and Konig (Konig) in 1924. Finally, the discovery of this method is attributed to more Gaussian (Gauss) in 1805.

### Contents [hide]

- Fast Fourier transform of O (N log N). Application to the multiplication of two polynomials or long numbers
  - Discrete Fourier Transform (DFT)
  - Application of DFT for fast multiplication of polynomials
  - Fast Fourier transform
  - Inverse FFT
  - Implementation
  - Improved implementation: computing "on the spot" without additional memory
  - Additional optimization
  - The discrete Fourier transform in modular arithmetic
  - Some applications
    - All sums
    - All kinds of scalar products
    - Two strips

## Discrete Fourier Transform (DFT)

Let there be a polynomial of  $n$  degree -s:

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

Without loss of generality, we can assume that  $n$  is a power of 2. If in fact  $n$  is not a power of 2, we simply add the missing factors, setting them equal to zero.

From the theory of functions of a complex variable is known that the complex roots of unity exists exactly  $n$ . Denote these roots by  $w_{n,k}$ ,  $k = 0 \dots n - 1$  then 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 degree of unity) is such that all the other roots are its powers:  $w_{n,k} = (w_n)^k$ .

Then the **discrete Fourier transform (DFT)** (discrete Fourier transform, DFT) of a polynomial  $A(x)$  (or, equivalently, the DFT of the vector of its coefficients  $(a_0, a_1, \dots, a_{n-1})$ ) are the values of this polynomial at points  $x = w_{n,k}$ , ie is a vector:

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

Is defined similarly and **inverse discrete Fourier transform** (InverseDFT). Inverse DFT for the vector of a polynomial  $(y_0, y_1, \dots, y_{n-1})$  a 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 DFT coefficients of the polynomial passes from its value in the complex roots of  $n$ th roots of unity, then the inverse DFT - on the contrary, from the values of the coefficients of the polynomial recovers.

## Application of DFT for fast multiplication of polynomials

Suppose we are given two polynomials  $A$  and  $B$ . Calculate the DFT for each of them:  $\text{DFT}(A)$  and  $\text{DFT}(B)$  - two vector values of polynomials.

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

$$(A \times B)(x) = A(x) \times B(x).$$

But it does mean that if we multiply the vector  $\text{DFT}(A)$  and  $\text{DFT}(B)$ , simply multiply each element of a vector to the corresponding element of another vector, we get nothing else, as the DFT of a polynomial  $A \times B$ :

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

Finally, applying the inverse DFT 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. This work obviously requires to compute only  $O(n)$  operations. Thus, if we learn to calculate the DFT and inverse DFT of the time  $O(n \log n)$ , then the product of two polynomials (and, consequently, the two long numbers) we can find for the same asymptotic behavior.

It should be noted that, firstly, the result should be two polynomials of degree one (simply adding one of these coefficients with zeros). Second, as a result of the product of two polynomials of degree  $n$  polynomial of degree obtained  $2n - 1$ , so that the result is correct, pre-need to double every polynomial of degree (again, adding to their zero coefficients).

## Fast Fourier transform

**Fast Fourier transform** (fast Fourier transform) - a method that allows to calculate the DFT of the time  $O(n \log n)$ . This method is based on the properties of the complex roots of unity (namely, that some degree of give other roots roots).

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

Thus, suppose there is a polynomial  $A(x)$  of degree  $n$  where  $n$  - 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 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:

$$A(x) = A_0(x^2) + xA_1(x^2). \quad (1)$$

Polynomials  $A_0$  and  $A_1$  have twice lower degree than the polynomial  $A$ . If we can in linear time by the calculated DFT( $A_0$ ) and DFT( $A_1$ ) calculate DFT( $A$ ), and we obtain the desired fast Fourier transform algorithm (because it is a standard chart of "divide and rule", and it is 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  $\{y_k^1\}_{k=0}^{n/2-1} = \text{DFT}(A_1)$ . Let us find the expression for  $\{y_k\}_{k=0}^{n-1} = \text{DFT}(A)$ .

Firstly, 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 after the transformation coefficients 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 identities  $w_n^n = 1$ ,  $w_n^{n/2} = -1$ .)

So as a result we got a formula for calculating the total vector  $\{y_k\}$ :

$$\begin{aligned} y_k &= y_k^0 + w_n^k y_k^1, \quad k = 0 \dots n/2 - 1, \\ y_{k+n/2} &= y_k^0 - w_n^k y_k^1, \quad k = 0 \dots n/2 - 1. \end{aligned}$$

(These formulas, ie two formulas of the form  $a + bc$  and  $a - bc$ , sometimes called "butterfly transformation" ("butterfly operation"))

Thus, we finally built the FFT algorithm.

## Inverse FFT

Thus, suppose given a vector  $(y_0, y_1, \dots, y_{n-1})$  - the value of the polynomial  $A$  degree  $n$  in points  $x = w_n^k$ . You need to restore the coefficients  $(a_0, a_1, \dots, a_{n-1})$  of the polynomial. This well-known problem called **interpolation**, for this task, there are some common algorithms for the solution, but in this case will get a very simple algorithm (simple fact that it is

virtually identical to the 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  $(y_0, y_1, \dots, y_{n-1})$  by the inverse matrix to the matrix on the left (which, incidentally, is called Vandermonde matrix):

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

A direct check 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 almost indistinguishable, so that the coefficients  $a_k$  can be found in the same algorithm "divide and rule" as a direct FFT, but instead  $w_n^k$  should be used everywhere  $w_n^{-k}$ , and every element of the result should be divided into  $n$ .

Thus, the calculation of the inverse DFT is almost indistinguishable from a direct calculation of the DFT, and it can also perform a time  $O(n \log n)$ .

## Implementation

Consider a simple recursive **implementation of FFT** and inverse FFT, realize them in the form of a single function, since the differences between the direct and inverse FFT minimal. To store the use of complex numbers in standard 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++) {
        if (i < n/2)
            a0[j] = a[i];
        else
            a1[j] = a[i];
        j++;
    }

    fft(a0, invert);
    fft(a1, invert);

    for (int i = 0, j = 0; i < n; i++) {
        base t = a0[j];
        if (invert)
            t = conj(t);
        a[i] = a0[j] + t * a1[j];
        a[i + n/2] = a[i] - t * a1[j];
        j++;
    }
}
```

```

        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 the argument of `a` the function is passed to the input vector of coefficients in it and will contain the same result. The argument `invert` indicates direct or inverse DFT should be calculated. Inside the function first checks if the length of the vector `a` is equal to one, then 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. Then we calculate the value of  $w_n$ , and the plant variable `w` containing the current level  $w_n$ . Then calculated the resulting DFT elements on the above formulas.

If the flag is specified `invert = true`, it  $w_n$  is replaced by  $w_n^{-1}$ , and each element of the result is divided by 2 (noting that these divisions by 2 will take place in each level of recursion, then eventually just turns out that all the elements are divided into  $n$ ).

Then the function for **multiplying two polynomials** is 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, in theory, nothing prevents her work with fractional coefficients). However, it appears the problem of a large error in the calculation of the DFT: the error can be significant, so the rounding of the best most reliable way - adding 0.5 and then rounding down ( **note** : this will not work properly for negative numbers, if any, may appear in your application).

Finally, the function for **multiplying two long numbers** is virtually no different from the function for multiplication of polynomials. The only feature - that after the multiplication of numbers as polynomials should be normalized, ie perform all transfers 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 is never surpass the total length of numbers, the size of the vector `res` will be enough to carry out all translations.)

## Improved implementation: computing "on the spot" without additional memory

To increase the efficiency abandon recursion explicitly. In the above recursive implementation, we clearly separated the vector  $a$  into two vectors - elements on the even positions attributed to the same time create a vector, and on odd - to another. However, if we rearrange the elements in a certain way, the need for a temporary vectors would then be eliminated (ie, all the calculations we could produce "on the spot", right in the vector  $a$ ).

Note that on the first level of recursion elements younger (first) position bits are zero, refer to the vector  $a_0$ , and the least significant bits of positions which are equal to one - to the vector  $a_1$ . At the second level of recursion is done the same thing, but for the second bit, etc. Therefore, if we are in the position  $i$  of each element  $a[i]$  invert the order of bits, and reorder elements in an array  $a$  according to the new index, then we get the desired order (it is called **bitwise inverse permutation** (bit-reversal permutation)).

For example, for  $n = 8$  this procedure 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,  $[a_0, a_2, a_4, a_6]$  and  $[a_1, a_3, a_5, a_7]$ . As we can see, in the bitwise inverse permutation, this corresponds to a separation of the vector into two halves: the first  $n/2$  element and the last  $n/2$  element. Then the recursive call of each half; suppose DFT resulting from each of these was returned to the place as the elements (i.e., the first and second halves of the vectors  $a$ , respectively):

$$a = \left\{ \left[ y_0^0, y_1^0, y_2^0, y_3^0 \right], \left[ y_0^1, y_1^1, y_2^1, y_3^1 \right] \right\}.$$

Now we need to perform a union of two into one DFT for the entire vector. But the elements stood out so well that the union can be performed directly in the array. Indeed, we take the elements  $y_0^0$  and  $y_0^1$  is applicable to them transform butterflies, and the result is put in their place - and this place and would thus, and that was to happen:

$$a = \left\{ \left[ y_0^0 + w_n^0 y_0^1, y_1^0, y_2^0, y_3^0 \right], \left[ y_0^0 - w_n^0 y_0^1, y_1^1, y_2^1, y_3^1 \right] \right\}.$$

Similarly, the transformation is applied to the butterfly  $y_1^0$  and  $y_1^1$  the result put in their place, etc. As a result, we obtain:

$$a = \left\{ \begin{aligned} &\left[ y_0^0 + w_n^0 y_0^1, y_1^0 + w_n^1 y_1^1, y_2^0 + w_n^2 y_2^1, y_3^0 + w_n^3 y_3^1 \right], \\ &\left[ y_0^0 - w_n^0 y_0^1, y_1^0 - w_n^1 y_1^1, y_2^0 - w_n^2 y_2^1, y_3^0 - w_n^3 y_3^1 \right] \end{aligned} \right\}.$$

Ie we got exactly the desired DFT of the vector  $a$ .

We describe the process of calculating the DFT at the first level of recursion, but it is clear that the same arguments are valid for all other levels of recursion. Thus, **after applying the bitwise inverse permutation is possible to calculate the DFT on the spot**, without additional arrays.

But now you can **get rid of the recursion** explicitly. So we applied bitwise inverse permutation elements. Now do all the work being done by the lower level of recursion, ie vector  $a$  divide into pairs of elements for each applicable conversion butterflies, resulting in the vector  $a$  will be the results of the lower level of recursion. In the next step the vector divide  $a$  at quadruple elements applied to each butterfly transform, thus obtaining a DFT for every four. And so on, finally, in the last step, we received the results of the DFT for the two halves of the vector  $a$  by applying the transformation of butterflies and obtain the DFT for the entire 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  $a$  is applied bitwise inverse permutation, which calculates the number of significant bits ( $\lg n$ ) in the number  $n$ , and for each position  $j$  is the corresponding position, the bit which has a bit entry record number  $i$  recorded in the reverse order. If The resulting position was more  $i$ , the elements in these two positions must be exchanged (if not this condition, each couple will exchange twice, and in the end nothing happens).

Then, the  $\lg n - 1$  algorithm steps, on  $k$  which th of ( $k = 2 \dots \lg n$ ) are computed for the DFT block length  $2^k$ . 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 through  $i$  iterated for blocks, and invested in it cycle for  $j$  conversion applies to all elements of the butterfly unit.

You can perform further **optimization of the reverse bits**. In the previous implementation, we obviously took over all bits of the number, incidentally bitwise inverted order number. However, the reverse of bits can be performed in a different way.

For example, suppose  $j$ - has the counted number equal to the inverse permutation of bits  $i$ . Then, during the transition to the next number  $i + 1$  we have and the number  $j$  add one, but add it to this "inverted" notation. In a typical binary system add one - so remove all the units standing on the end of the number (ie, younger group of units), and put the unit in front of them. Accordingly, in the "inverted" system, we have to go bit number, starting with the oldest, and as long as there are units, remove them and move on to the next bit; when will meet the first zero bit, put the unit into it and stop.

So, we get the following implementation:

```

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

We give a list of other optimizations, which together can significantly accelerate the preceding "improved" implementation:

- **Predposchitat reverse bits** for all numbers in a global table. It is especially easy when the size  $n$  is the same for all calls.

This optimization becomes appreciable when a large number of calls  $\text{fft}()$ . However, the effect of it can be seen even with three calls (three calls - the most common situation, ie when it is required once multiply two polynomials).

- Refuse to use `vector` ([go to regular arrays](#)).

The effect of this depends upon the particular compiler, but typically it is present and approximately 10% -20%.

- **Predposchitat all degrees** of  $wlen$ . In fact, in this cycle of the algorithm repeatedly made the passage in all degrees of  $wlen$  on  $0$  to  $\log_2 len - 1$ :

```

        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 in some array all the required extent, and thus get rid of unnecessary multiplications in the nested loop.

Tentative acceleration - 5-10%.

- Get rid of the **array accesses in the indices**, instead use a pointer to the array of promoting their right to 1 at each iteration.

At first glance, optimizing compilers should be able to cope with this, but in practice it turns out that the replacement of references to arrays  $a[i + j]$  and  $a[i + j + \log_2 len]$  pointers to speed up the program in popular compilers. Winning is 5-10%.

- **Abandon the standard type of complex numbers** `complex`, put it into your own implementation.

Again, this may seem surprising, but even in modern compilers gain from such rewriting can be up to several tens of percent! This indirectly confirms a widespread claim that compilers perform worse with formulaic data types, optimizing work with them much worse than a non-formulaic types.

- Another useful optimization is the **cut-off length**, when the length of the working unit becomes small (say, 4) to calculate the DFT for it "by hand". If paint these cases in the form of explicit formulas for length equal to  $4/2$ , the values of the sine-cosine take integer values due to which you can get a speed boost for a few more tens of percent.

Here we present the realization of the described improvements (except for the last two points, which lead to the proliferation 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, this implementation faster than the previous "improved" version of 2-3.

## The discrete Fourier transform in modular arithmetic

At the heart of the discrete Fourier transform are complex numbers, the roots of  $n$ th roots of unity. To effectively calculate it used features such roots as the existence of  $n$ different roots, forming a group (ie, the degree of the same root - the root is always the other, among them there is one element - the generator of the group, called a primitive root).

But the same is true of the roots of  $n$ th roots of unity in modular arithmetic. More precisely, not for any module  $p$  exists a  $n$  distinct roots of unity, however, these modules do exist. Is 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 other  $n - 1$ roots  $n$ th roots of unity in modulus  $p$  can be obtained as the degree of the primitive root  $w_n$ (as in the complex case).

For use in the fast Fourier transform algorithm we needed to root primivny existed for some  $n$ , a power of two, as well as all the smaller degrees. And if in the complex case, there was a primitive root for all  $n$ , in the case of modular arithmetic is generally not the case. However, note that if  $n = 2^k$ , that  $k$ Star power of two, the modulo  $m = 2^{k-1}$ have:

$$(w_n^2)^m = (w_n)^n = 1 \pmod{p}, \\ (w_n^2)^k = w_n^{2k} \neq 1 \pmod{p}, \quad 1 \leq k < m.$$

Thus, if  $w_n$ - a primitive root  $n = 2^k$ th roots of unity, then  $w_n^2$ - primitive root  $2^{k-1}$ th roots of unity. Therefore, all powers of two smaller  $n$ , the desired degree of primitive roots also exist and can be calculated as the corresponding extent  $w_n$ .

The final touch - for inverse DFT we used instead of the  $w_n$ inverse element:  $w_n^{-1}$ . But modulo  $p$ inverse is also always a.

Thus, all the required properties are observed in the case of modular arithmetic, provided that we have chosen some rather large unit  $p$ , and found in it a primitive root  $n$ th roots of unity.

For example, you can take the following values: module  $p = 7340033$ ,  $w_{220} = 5$ . If this module is not enough to find another pair, you can use the fact that for the modules of the form  $c2^k + 1$ (but still necessarily simple) there will always be a primitive root of degree  $2^k$ of unity.

```

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) {
            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+j], v = int (a[i+j+len/2] * 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 `n` element in absolute value `mod` (see. [The inverse element in the mod](#)). Constants `mod`, define a module and a primitive root, and - the inverse of the element modulo .`root` `root_pw` `root_1` `root_mod`

As practice shows, the implementation of integer DFT works even slower implementation of complex numbers (because of the huge number of transactions modulo), but it has advantages such as less memory usage and lack of rounding errors.

## Some applications

In addition to the direct application to multiply polynomials or long numbers, we describe here are some other applications of the discrete Fourier transform.

### All sums

Problem: given two arrays `a` and `b`. You want to find all sorts of species  $a[i] + b[j]$ , and for each such number of ways to withdraw get it.

For example, for  $a = (1, 2, 3)$  and  $b = (2, 4)$  obtain the number of 3 can be prepared 1 by 4 - one and 5 - 2, 6 - 1 7 - 1.

We construct for arrays `a` and `b` two polynomials `A` and `B`. As the degrees of the polynomial will be performing numbers themselves, ie, values  $a[i]$  ( $b[i]$ ), and as the coefficients in them - this is the number of how many times encountered in the array  $a$  ( $b$ ).

Then, multiply the two polynomials over  $O(n \log n)$ , we obtain a polynomial `C`, where the powers are all sorts of species  $a[i] + b[j]$ , and their coefficients are just the required number of

### All kinds of scalar products

Given two array `a` and `b` the same length  $n$ . You want to display the values of each of the scalar product of the vector `a` for the next cyclic shift vector `b`.

Invert the array `a` and assign it to the end of the `nzeros`, and the array `b` - just assign yourself. Then multiply them as polynomials. Now consider the coefficients of the product  $c[n \dots 2n - 1]$  (as always, all the indices in the 0-indexed). We have:

$$c[k] = \sum_{i+j=k} a[i]b[j].$$

Since all the elements  $a[i] = 0$ ,  $i = n \dots 2n - 1$ , we obtain:

$$c[k] = \sum_{i=0}^{n-1} a[i]b[k-i].$$

It is easy to see in this sum, that this is the scalar product  $a$  on  $k - n - 1$ th cyclic shift. Thus, these coefficients (since  $n - 1$ th and pumping  $2n - 2$  of th) - is the answer to the problem.

The solution is obtained with the asymptotic behavior  $O(n \log n)$ .

## Two strips

Given two strips defined as two boolean (ie, numerical value of 0 or 1) of the array  $a[]$  and  $b[]$ . You want to find all of these positions on the first strip that if you make starting from 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 of the strip, in the form of 0/1 - you can wake up in the cage or not, and given some figure as a template (in the form of an array in which 0 - no cells, 1 - is) required find all the 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 arrays 0/1 - a number of elements in which both were unity. Our goal is to find all the cyclic shifts of the second strip, so that there was not a single element in which at least two strips were unity. Ie we must find all the cyclic shifts of the second array, in which the inner product is zero.

Thus, this problem we decided for  $O(n \log n)$ .

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:54  
EDIT: 14 Jan 2013 17:58

## Sqrt-decomposition

Sqrt-decomposition - is a method or a data structure that allows some typical operations (sum of subarray elements, to find the minimum / maximum, etc.) for  $O(\sqrt{n})$ , significantly faster than  $O(n)$  for the trivial algorithm.

First, we describe a data structure for one of the simplest applications of this idea, then show how to generalize it to solve some other problems, and, finally, look at some other use of this idea: the partition of the input requests sqrt-blocks.

### The data structure on the basis of decomposition sqrt-

We pose the problem . Given an array  $a[0 \dots n - 1]$ . Required to implement such a data structure that will be able to find the sum of the elements  $a[l \dots r]$  for arbitrary  $l$  and  $r$  for  $O(\sqrt{n})$  operations.

#### Description

The basic idea sqrt-decomposition is what to do next **predposchët** : divide the array  $a$  into blocks of length approximately  $\sqrt{n}$ , and in each block  $i$  in advance predposchitaem sum  $b[i]$  of elements in it.

We can assume that the length of one unit and the number of units equal to one and the same number - the root of  $n$ , rounded up:

$$s = \lceil \sqrt{n} \rceil,$$

then the array  $a$  is divided into blocks like this:

$$\underbrace{a[0] \ a[1] \ \dots \ a[s-1]}_{b[0]} \ \underbrace{a[s] \ a[s+1] \ \dots \ a[2 \cdot s-1]}_{b[1]} \ \dots \ \underbrace{a[(s-1) \cdot s] \ \dots \ a[n]}_{b[s-1]}.$$

Although the latter unit may comprise less than  $s$ , the elements (if  $n$  not divisible  $s$ ) - is not essential.

Thus, for each block  $k$ , we know the amount on it  $b[k]$ :

$$b[k] = \sum_{i=k \cdot s}^{\min(n-1, (k+1) \cdot s-1)} a[i].$$

So, let the values  $b_k$  previously calculated (that is, obviously,  $O(n)$  operations). That they can give the calculation of the response to another request  $(l, r)$ ? Note that if the interval  $[l; r]$  is long, then it will contain several blocks as a whole, and these blocks, we can find out the amount on them in a single operation. As a result, the total length  $[l; r]$  will be only two blocks falling into it is only partially, and these pieces we have to sum trivial algorithm.

Illustration (here  $k$  designated block number, wherein the lie  $l$ , and by  $p$ - the number of the block in which lies  $r$ ):

$$\dots \underbrace{a[l] \ \dots \ a[(k+1) \cdot s-1]}_{b[k+1]} \underbrace{a[(k+1) \cdot s] \ \dots \ a[(k+2) \cdot s-1]}_{\text{sum=?}} \ \dots \underbrace{a[(p-1) \cdot s] \ \dots \ a[p \cdot s-1]}_{b[p]} \ a[p \cdot s] \ \dots \ a_r \ \dots$$

This figure shows that in order to calculate the amount of the segment  $[l \dots r]$ , it is necessary to sum the elements of only two "tails":  $[l \dots (k+1) \cdot s-1]$  and  $[p \cdot s \dots r]$ , and sum the values  $b[i]$  in all the blocks, starting from  $k+1$  and ending with  $p-1$ :

$$\sum_{i=l}^r a[i] = \sum_{i=l}^{(k+1) \cdot s-1} a[i] + \sum_{i=k+1}^{p-1} b[i] + \sum_{i=p \cdot s}^r a[i]$$

(Note: this is incorrect formula where  $k = p$ : in this case, some elements are added together twice and in this case it is necessary to sum the elements simply  $\text{by } r$ )

Thus we ekonomim significant number of operations. Indeed, the size of each of the "tails" clearly does not exceed the block length  $s$  and the number of blocks not exceed  $s$ . Since we have chosen  $\approx \sqrt{n}$ , the total amount for the calculation of the interval  $[l \dots r]$ , we need only  $O(\sqrt{n})$  operations.

#### Implementation

We give first a simple realization:

```
// входные данные
int n;
```

#### Contents [hide]

- Sqrt-decomposition
  - The data structure on the basis of decomposition sqrt-
    - Description
    - Implementation
    - Other objects
  - Sqrt-decomposition of the input query
    - An example of the problem: adding the interval
    - Example problem: disjoint-set-union division
  - Off-line tasks to requests for subsegments and versatile array sqrt-heuristics for them

```

vector<int> a (n);

// предпосчёт
int len = (int) sqrt (n + .0) + 1; // и размер блока, и количество блоков
vector<int> b (len);
for (int i=0; i<n; ++i)
    b[i / len] += a[i];

// ответ на запросы
for (;;) {
    int l, r; // считываем входные данные - очередной запрос
    int sum = 0;
    for (int i=l; i<=r; )
        if (i % len == 0 && i + len - 1 <= r) {
            // если i указывает на начало блока, целиком лежащего в [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 other operations). Instead, the block numbers can be counted  $c_l$  and  $c_r$  where the boundaries are  $l$  and  $r$  respectively, and then making a loop in the blocks  $c_l + 1$  by  $c_r - 1$  separately treating "tails" in the blocks  $c_l$  and  $c_r$ . Furthermore, in the case of such an implementation  $c_l = c_r$  becomes special and require 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=c_l+1; i<=c_r-1; ++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 objects

We consider the problem of finding the sum of array elements in some of its subsegments. This task can expand a little: solve and **change** individual elements of the array  $A$ . Indeed, if some change element  $a_i$ , it suffices to update the value  $b[k]$  in the block, wherein the member is ( $k = i/\text{len}$ )

$b[k] += a[i] - \text{old\_}a[i]$ .

On the other hand, instead of the problem of the amount can be similarly solved the problem of **minimum**, **maximum** elements in the interval. If these problems prevent changes in individual elements, too, will have to recalculate the value  $b_k$  of the unit that owns the variable element, but recalculate already fully pass on all elements of the unit  $O(\text{len}) = O(\sqrt{n})$  operations.

Similarly sqrt-decomposition can be applied for a variety of **other** similar problems: finding the number of zero elements, the first non-zero element, count the number of certain elements, etc.

There is also a class of problems, when there are **changes in the elements on the whole subsegments**: adding or appropriation of elements at some subsegments of the array  $A$ .

For example, you need to perform the following two types of queries added to all elements of a certain length  $[l; r]$  value  $\delta$ , and recognize the value of an individual element  $a_i$ . Then, as  $b_k$  the set value that must be added to all elements of  $k$  the  $i$ th block (for example, initially all  $b_k = 0$ ); Then when you run the query "addition" will need to perform addition to all the elements  $a_i$  of "tails", and then perform addition to all the elements  $b_i$  for blocks lying entirely in the segment  $[l \dots r]$ . A response to the second request is likely to be just  $a_i + b_k$  where  $k = i/\text{len}$ . Thus, the addition of the segment will run 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 an interval and response to requests for the same interval. Both types of operations will be carried out for  $O(\sqrt{n})$ . To do this already will have to do two "block" of the array  $b$  and  $c$  - to ensure changes in the segment, the other - to answer queries.

Can give an example, and other tasks, which can be applied sqrt-decomposition. For example, you can solve the problem of **maintaining a set of numbers** with the ability to add / remove numbers, check numbers belonging to the set, the search  $k$ -th order number. To solve this problem it is necessary to store numbers in sorted order, separated by a few blocks of  $\sqrt{n}$  numbers in each. When you add or remove numbers will have to produce a "rebalancing" of blocks, throwing the number of start / end of one block in the beginning / end of the neighboring blocks.

## Sqrt-decomposition of the input query

Consider now a very different ideas about the use sqrt-decomposition.

Suppose that we have some problem in which we are given some input data, and then enter  $k$  the commands / queries, each of which we have to give to process and issue a response. We consider the case when requests are as requested (do not change the state of the system, but only asks for some information) and modifying (ie affecting the state of the system is initially set to the input data).

Specific task can be quite complex, and "honor" her decision (which reads one request, processes it, changing the state of the system, and returns a response) may be technically difficult or even be beyond the power of deciding. On the other hand, the solution of "off-line" version of the task, i.e. when there is no modifying operation, and there are only requesting queries - often much easier. Suppose that we know how to solve "offline" version of the problem, ie, building for some time  $B(n)$  some data structure that can answer queries, but does not know how to handle modifying queries.

Then we will divide the input requests for blocks (how long - is not specified, this length is denoted by  $s$ ). At the start of processing for each block will  $B(n)$  construct a data structure for "off-line" version of the task of data at the beginning of this block.

Now we will in turn take requests from the current block and each handle. If the current request - modifying, then skip it. If the current request - request, please refer to the data structure for the off-line version of the problem, but before taking into account all modifying queries in the current block . Take account of such modifying queries is not always possible, and it should be fast enough - for the time  $O(s)$  or a little worse; denote this time through  $Q(s)$ .

Thus, if we all  $m$  requests that need to process them  $B(m)\frac{m}{s} + mQ(s)$  time. Value  $s$  should be chosen based on the specific form of the functions  $B()$  and  $Q()$ . For instance, if  $B(m) = O(m)$  and  $Q(s) = O(s)$ , then the best choice would be  $s \approx \sqrt{m}$ , and the asymptotic behavior of the final turn  $O(m\sqrt{m})$ .

Since the arguments given above is too abstract, we present a few examples of problems to which this applies sqrt-decomposition.

### An example of the problem: adding the interval

Condition of the problem: given an array of numbers  $a[1 \dots n]$  and received two types of queries: find value in the  $i$ th element of the array, and add a number  $x$  to all elements of the array in some interval  $a[l \dots r]$ .

Although this problem can be solved without this technique with the partition of requests for blocks, we present it here - as simple and obvious application of this method.

So, let us divide the input requests for blocks  $\sqrt{m}$  (where  $m$ - the number of requests). At the beginning of the first block any requests to build structures is not necessary, just store the array  $a[]$ . Come on now at the request of the first block. If the current request - a request the addition, it is still miss him. If the current request - request for reading values in some positions  $i$ , the first just take as a response value  $a[i]$ . Then loop through all missed in this block requests the addition, and for those who gets in  $i$ , we apply them to increase the current account.

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 of queries we must use all modifying queries of this block to the array  $a[]$ . But it is easy to do for  $O(n)$ - enough for each request adding  $(l, r, x)$  noted in the auxiliary array at  $l$  the number  $x$ , and the point  $r + 1$ - the number  $-x$ , and then go through this array, adding a running total of the array  $a[]$ .

Thus, the final decision will be asymptotic behavior  $O(\sqrt{m}(n + m))$ .

### Example problem: disjoint-set-union division

There is an undirected graph with  $n$  vertices and  $m$  edges. Receives requests of three kinds: add an edge  $(x_i, y_i)$ , remove the edge  $(x_i, y_i)$ , and check whether or not related peaks  $x_i$  and  $y_i$  through.

If no removal requests, then the solution of the problem would have been well-known data structure disjoint-set-union (a system of disjoint sets) . However, in the presence of deletions task becomes much more complicated.

Do as follows. At the beginning of each block requests see what the ribs in this block will be removed, and immediately remove them from the graph. Now we construct a system of disjoint sets (dsu) on the resulting graph.

As we now have to respond to the next request from the current block? Our system of disjoint sets "knows" all the edges, except for those that are added / removed in the current block. However, removal of dsu we do not have to - we have removed all such advance edges of the graph. Thus, all that can be - it's extra, add ribs, which can be a maximum of  $\sqrt{m}$  pieces.

Consequently, in response to the current request requesting we can just let the preorder traversal on the connected components dsu, that will work for  $O(\sqrt{m})$ , as we have in the examination will only  $O(\sqrt{m})$  edges.

## 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 received requesting queries of the form  $(l, r)$  to learn something about the subsegments  $a[l \dots r]$ . We believe that the request is modified, and we are known in advance, ie, task - off-line.

Finally, we introduce the latest restriction : we believe that we can quickly recalculate the answer to the query when you change the left or right border of the unit. Ie if we knew the answer to a query  $(l, r)$ , then quickly be able to find an answer to your inquiry  $(l + 1, r)$  or

$(l - 1, r)$  or  $(l, r + 1)$  or  $(l, r - 1)$ .

We now describe the **universal heuristics** for all these problems. Sort the requests for the pair:  $(l \text{ div } \sqrt{n}, r)$ . Ie we sorted requests to the number sqrt-block, which lies in the left end, and at equality - at the right end.

Consider now the group queries with the same value  $l \text{ div } \sqrt{n}$ , and will process all requests of the group. The answer to the first query count trivial way. Each subsequent request will be considered on the basis of the previous response: ie to move left and right boundaries of the previous request to the borders of the next request, while maintaining the current response. We estimate the asymptotic behavior: left margin every time could move no more  $\sqrt{n}$  time, and the right - no more than  $n$  once in the sum over all the demands of the current group. Total, if the current group consisted of  $k$  requests will be made in the amount of not more than  $n + k \cdot \sqrt{n}$  recount. In total, around the algorithm work -  $O((n + m) \cdot \sqrt{n})$  recount.

A simple **example of** this heuristic is on such a task: to find out the number of different numbers in the segment of the array  $[l; r]$ .

A little more of sophistication version of this problem is a [problem with one of the rounds Codeforces](#).

---

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 11 Jun 2008 10:54  
EDIT: 11 Jun 2008 10:56

## Fenwick tree

**Fenwick tree** - a data structure tree on the array with the following properties:

1) allows us to calculate the value of a reversible operation of G on any interval  $[L; R]$  for time  $O(\log N)$  ;

I "jump" 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 of the interval  $[F(F(R)-1); F(R)-1]$ , and so on, until the bar reaches zero.

Function inc moves in the opposite direction - in the direction of increasing index, updating the sum value  $T_j$  for only those positions for which it is needed, i.e. for all  $j$ , for which  $F(j) \leq i \leq j$ .

Obviously, the choice of  $F$  depends on both the speed of the operations. Now we consider the function that allows to achieve a logarithmic performance in both cases.

**Define the value of  $F(X)$**  as follows. Consider the binary representation of the numbers and look at its least significant bit. If it is zero, then  $F(X) = X$ . Otherwise, the binary representation of  $X$  ends in a group of one or more units. Replace all the units in the group with zeros, and assign that number value of the function  $F(X)$ .

This rather complicated description corresponds to a very simple formula:

$$F(X) = X \& (X + 1),$$

where  $\&$  - a 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 these numbers  $j$ , for which  $F(j) \leq i \leq j$ .

However, it is not hard to make sure that all these numbers are obtained from  $i$  successive replacements of the rightmost (least significant) zero in the binary representation. For example, for  $i = 10$ , we find that  $j = 11, 15, 31, 63$ , etc.

### Contents [hide]

- Fenwick tree
  - Implementation Fenwick tree for the sum of one-dimensional case
  - Implementation Fenwick tree for a minimum of one-dimensional case
  - Implementation Fenwick tree for the sum of two-dimensional case

Ironically, such an operation (replacement of the youngest zero per unit) also corresponds to a very simple formula:

$$H(X) = X \mid (X + 1),$$

where  $\mid$  - is bitwise logical "OR".

## Implementation Fenwick tree for the sum of one-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)
        T = result + [R];
    Return result;
}

void inc (int i, int Delta)
{
    for (; i <n; i = (i | (i +1)))
        T [i] = Delta +;
}

int sum (L int, 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]);
}

```

## Implementation Fenwick tree for a minimum of

## one-dimensional case

It should be noted immediately that, as Fenwick tree allows you to find the value of the function in an arbitrary interval  $[0; R]$ , we will not be able to find at least on the interval  $[L; R]$ , where  $L > 0$ . Then, all value changes should only occur downward (again, because does not work to turn the function min). This is a significant limitation.

```
vector <int> T;
int n;

const int INF = 1000 * 1000 * 10
```

## Implementation Fenwick tree for the sum of 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;
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

## The system of disjoint sets

This article describes the data structure "**a system of disjoint sets**" (in English "disjoint-set-union", or simply "DSU").

This data structure provides the following features. Initially, there are several elements, each of which is a separate (his own) set. In one operation, you can **combine any two sets**, and you can **request, in which the set** is now specified item. Also, in the classic version, introduced another operation - the creation of a new element, which is placed in a separate set.

Thus, the basic data structure of the interface consists of only three operations:

- **make\_set( $x$ )**- **adds** a new element  $x$  by placing it in a new set consisting of one it.
- **union\_sets( $x, y$ )**- **unites** these two sets (set in which there is an element  $x$ , and set in which there is an element  $y$ ).
- **find\_set( $x$ )**- **returns in which the set** is specified item  $x$ . In fact, at the same time returns one of the elements of the set (called **representative** or **leader** (in English literature "leader")). This representative is selected in each set of the data structure (and may vary over time, namely after the call `union_sets()`).

For example, if a call `find_set()` for any one of the two elements has returned to the same value, this means that these elements are in one and the same set, and otherwise - to the different sets.

Added: 11 Jun 2008 10:56  
EDIT: 24 Jan 2013 12:47

### Contents [hide]

- The system of disjoint sets
  - Building an efficient data structure
    - Naive implementation
    - Heuristics compression path
      - Evaluation of the asymptotic behavior of the application of heuristics compression path
    - Heuristics union by rank
      - Evaluation of the asymptotic behavior of the application of heuristics rank
      - Combining heuristics: path compression plus rank heuristic
  - Use in various tasks and improve
    - Support for the connected components of the graph
    - Search the connected components in the image
    - Support for additional information for each set
    - Application DSU compression "jumps" over the interval. The problem of painting subsegments Offline
    - Support distances to the leader
    - Support for parity path length and the problem of verification of bipartite graphs online
    - Algorithm for finding the RMQ (at least in the interval) for  $O(\alpha(n))$  an average Offline
    - Algorithm for finding the LCA (lowest common ancestor in the tree) for  $O(\alpha(n))$  an average Offline
    - Storage DSU as an explicit list of sets. The application of this idea at the confluence of various data structures
    - Storage DSU preserving explicit tree structures. Pereopdoveshivanie. Search algorithm bridges in the graph of  $O(\alpha(n))$  the average online
  - Historical retrospective
  - Tasks in the online judges
  - Literature

Described below is a data structure allows each of these operations in nearly  $O(1)$ the average (for more details see the asymptotic behavior. below after the description of the algorithm).

Also in one of the sub-article describes an alternative embodiment of the DSU, which allows to achieve the asymptotic behavior of  $O(\log n)$ an average of one request at  $m \geq n$ ; and when  $m \gg n$ (i.e.,  $m$ much more  $n$ ) - and at the time  $O(1)$ average of the request (see. "Storage DSU as 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 be stored in the form of **trees** : one tree corresponds to one set. The root of the tree - a representative (leader) of the set.

When implemented, this means that we lead an array **parent**in which each element we store a reference to its parent in the tree. For the roots of trees, we assume that their ancestors - they (ie, link to fixate at this point).

### Naive implementation

We can already write the first implementation of a system of disjoint sets. It would be quite inefficient, but then we will improve it with the help of two methods, eventually getting almost constant during operation.

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 with the root at the top *v*, noting that her father - she herself.

To combine the two sets of (operation **union\_sets(*a*, *b*)**), we first find the leaders of the set, which is *a*, and sets in which there is *b*. If the leaders of the match, then do not do anything - it means that the sets already been merged. Otherwise, you can simply indicate that the ancestor vertex *b*is *a*(or vice versa) - thus joining one tree to another.

Finally, the implementation of the search operation leader ( **find\_set(*v*)**) is simple: we ascend the ancestors from the top *v*, until we reach the root, ie while a reference to the parent does not keep to himself. This operation is more convenient to implement recursively (especially it will be convenient later, in relation to added optimizations).

```

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 a system of disjoint sets is very **inefficient**. It is easy to construct an example where after several unions of sets get a situation that a lot - it's a tree, degenerated into a long chain. As a result, each call `find_set()` will work on this test during the order of the depth of the tree, ie, for  $O(n)$ .

This is very far from that of the asymptotics we were going to get (constant time). Therefore, we consider two optimizations that allow (applied even separately) significantly improved performance.

## Heuristics compression path

This heuristic is designed to accelerate the work `find_set()`.

It lies in the fact that when after the call `find_set(v)` we find the desired leader  $p$  sets, then remember that at the apex  $v$ , and all passed the path peaks - this is the leader  $p$ . The easiest way to do it by forwarding them `parent[]` to this vertex  $p$ .

Thus, an array of ancestors `parent[]` meaning varies slightly: now it is **compressed array of ancestors**, ie for each vertex there may be stored not in the immediate ancestor, and the ancestor ancestor ancestor ancestor, etc.

On the other hand, it is clear that you can do to these pointers `parent` always point to the leader: otherwise, in operation `union_sets()` would have to update the leaders in  $O(n)$  the elements.

Thus, the array `parent[]` should be treated just as an array of ancestors, possibly partially compressed.

The new implementation of the operation `find_set()` is as follows:

```

int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
}

```

This simple realization is doing everything that was intended, first by the

recursive call is the leader of the set, and then, in the process of promotion of the stack, this leader is assigned to links `parent` for all of the passed elements.

To implement this operation and can not recursive, but then you 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, non-recursive implementation does not provide a significant benefit.

## Evaluation of the asymptotic behavior of the application of heuristics compression path

We show that the use of a heuristic path compression **achieves logarithmic asymptotics** :  $O(\log n)$  a single request on average.

Note that since the operation `union_sets()` is a call operation, two `find_set()` and more  $O(1)$  operations, we can focus only on the proof time of the evaluation  $O(m)$  operations `find_set()`.

Let's call **the weight of**  $w[v]$  the top of  $v$  the number of descendants of this node (including its itself). Vertex weights, obviously, can only increase during the algorithm.

We call **span rib**  $(a, b)$  difference weights of all edges:  $|w[a] - w[b]|$  (apparent ancestor vertices in weight is always greater than the descendent vertices). It can be noted that the range of a fixed rib  $(a, b)$  can only increase during the algorithm.

In addition, we will divide the edges in the **classes** : we say that the edge has a 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  $x$  and will monitor how the edge in her ancestor: first, it is not (until the top  $x$  is the leader), then held an edge of  $x$  some sort in the top (where the set with a vertex  $x$  joins another set), and then may change compression paths during call `find_path`. It is clear that we are interested in the asymptotic behavior of only the latter case (with path compression): All other cases require  $O(1)$  time on a single request.

Consider the work of a call operation `find_set`: it takes place in a tree along a **path**, erasing all edges of this path and redirecting them to the leader.

Consider this path 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 excluded the  $O(\log n)$  edges of each request.

Let us now consider all **the other** edges of this path. For each such edge, if it has class  $k$ , it turns out that in this way there is one edge class  $k$  (otherwise we would be obliged to exclude the current edge, as the sole representative of the class  $k$ ). Thus, after path compression rib is replaced by at least an edge class  $k + 1$ . Given that reduces the weight of the edges 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 work  $m$  requests:

$O((n + m) \log n)$  that (if  $m \geq n$ ) is logarithmic while working on a single query on average.

## Heuristics union by rank

We consider here the other heuristics, which in itself can accelerate the time of the algorithm, and in combination with compression heuristic ways and at all capable of achieving almost constant running time per request, on average.

This heuristic is a slight change in the work `union_sets`: if a naive implementation of it, a tree will be appended to what is determined by chance, but now we will do it on the basis of **rank**.

There are two variants of the rank heuristic: In one embodiment, the rank of a tree is **the number of vertices** in it, in the other - **the depth of the tree** (or rather, the upper limit to the depth of the tree, as the joint application of heuristics compression ways the real depth of the tree can be reduced).

In both cases the essence of heuristics is the same: when the `union_sets` tree will be attached to the lower rank to a tree with a higher rank.

We present the implementation of **the rank heuristic based on the size of the 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];
    }
}
```

We present the implementation of **the rank heuristic 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[a];
    }
}

```

Both options are equivalent to rank a heuristic perspective asymptotic however in practice possible to use any of them.

## Evaluation of the asymptotic behavior of the application of heuristics rank

We show that the asymptotic behavior of the system of disjoint sets using only heuristics rank, without compression heuristics paths will **slide** to one inquiry on average:  $O(\log n)$ .

Here we show that for any of the two options rank heuristic depth of each tree will be the value of  $O(\log n)$  that will automatically mean logarithmic asymptotics for the request `find_set`, and therefore the request `union_sets`.

Consider **the rank heuristic in the depth of the tree**. We show that if the rank of the tree is  $k$ , the tree contains at least  $2^k$  vertices (here will automatically follow that the rank and, hence, the depth of the tree, is the value  $O(\log n)$ ). Will prove by induction: for  $k = 0$  this is obvious. When compressing ways depth can only decrease. Rank tree increases  $k - 1$  to  $k$  where it is joined by a tree of rank  $k - 1$ ; applying to these two trees the size of  $k - 1$  the induction hypothesis, we get that new tree rank  $k$  really will have a minimum of  $2^k$  vertices as required.

Let us now consider **the rank heuristic size trees**. We show that if the size of the tree is  $k$ , its height is not more  $\lfloor \log k \rfloor$ . Will prove by induction: for  $k = 1$  the statement is true. When compressing ways depth can only decrease, so path compression nothing breaks. Suppose now that merges two tree size  $k_1$  and  $k_2$ ; then by the induction of their height is less than or equal to, respectively,  $\lfloor \log k_1 \rfloor$  and  $\lfloor \log k_2 \rfloor$ . Without loss of generality, we assume that the first tree - small ( $k_1 \geq k_2$ ), so after the merger of the depth of the tree resulting from the  $k_1 + k_2$  peaks will be equal to:

$$h = \max(\lfloor \log k_1 \rfloor, 1 + \lfloor \log k_2 \rfloor).$$

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 heuristic

As mentioned above, the combined use of these heuristics gives the best results particularly in the end reaching almost constant operating time.

We do not give here the proof of the asymptotic behavior, as it is very volumetric (see., Eg, feed, Leiserson, Rivest, Stein "Algorithms. Design and Analysis"). For the first time this evidence was conducted Tarjanne (1975).

The final result is that the joint application of heuristics path compression and union by rank while working on a request is received  $O(\alpha(n))$ , on average, where  $\alpha(n)$ - the inverse Ackermann function , which grows very slowly, so slowly that for all reasonable limits  $n$ , it **does not exceed 4** (about to  $n \leq 10^{600}$ ) .

It is therefore about the asymptotic behavior of the system of disjoint sets is appropriate to say "almost constant during operation."

We give here **the final implementation of the system of disjoint sets** that implements both of these heuristics (heuristics used rank the relative depth of 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];
    }
}

```

## Use in various tasks and improve

In this section we consider some applications of the data structure "a system of

disjoint sets", as trivial, and use some improvements to the data structure.

## Support for the connected components of the graph

This is one of the most obvious applications of the data structure "a system of disjoint sets", which, apparently, and stimulated the study of the structure.

**Formally, the** problem can be formulated as follows: initially given a blank graph gradually in this graph can be added to the top and undirected edges, as well as requests come  $(a, b)$ - "in the same whether the connected components are peaks  $a$  and  $b$ ?".

Directly applying here the above data structure, we get a solution that processes a request to add the vertex / edge or request a review of the two peaks - in almost constant time on average.

Given that almost exactly the same problem is posed using **Kruskal's algorithm for finding the minimum spanning tree**, we immediately obtain the **improved version of the algorithm**, which works almost in linear time.

Sometimes encountered in practice **inverted version of this problem**: originally a graph with some vertices and edges, and receives requests to remove the ribs. If the task is given to offline, ie we can know in advance all the requests, to solve this problem as follows: perevernëm problem backwards: we assume that we have an empty graph, which can be added to the edges (first add the edge of the last request, then the penultimate, etc.). Thus, as a result of the inversion of this task, we came to the usual problem whose solution described above.

## Search the connected components in the image

One of the underlying surface applications DSU is to solve the following problem: there is the image  $n \times m$  pixels. Initially, the entire image is white, but then it draws some black pixels. Is required to determine the size of each "white" connected components in the final image.

To solve, we just iterate through all the white cells of the image, for each cell iterate through its four neighbors, and if a neighbor is also white - then call `union_sets()` these two vertices. Thus, we will have a DSU with  $nm$  vertices corresponding to the pixels of the image. Receive as a result trees DSU - is sought connected components.

This problem can be solved easily using the **crawl depth** (or **bypass wide**), but the method described here has a definite advantage: it can handle a matrix line (operating only with the current line, the previous line, and the system of disjoint sets built for the elements of one line), i.e. using the procedure  $O(\min(n, m))$  memory.

## Support for additional information for each set

"The system of disjoint sets" makes it easy to store any additional information pertaining to the sets.

A simple example - it's **the size of sets** : how to store them, it was described in the description of the rank of heuristics (where information was recorded for the current leader of the set).

Thus, together with the leader of each set can store any additional required information in a specific task.

## Application DSU compression "jumps" over the interval. The problem of painting subsegments Offline

One of the most common applications of DSU is that if there is a set of items, and each item comes from one edge, then we can quickly (in almost constant time) to find the end point, which do we get if we move along the edges of a given starting point.

A good example of this application is the **task of painting subsegments** : a segment length  $L$ , each cell of which (ie the length of each piece 1) has zero color. Receives requests species  $(l, r, c)$ - repaint segment  $[l; r]$  in color  $c$ .

Required to find the final color of each cell. Request assumed to be known in advance, ie, task - Offline.

For solutions we can make DSU-structure that for each cell will keep a reference to the near right unpainted cell. Thus, initially, each cell indicates herself, and after painting the first subsegment - cell before subsegment will point to the cell after the end of the subsegment.

Now, to solve the problem, we consider repainting requests **in reverse order**, from last to first. To execute the query each time we just using our DSU find the most left unpainted cell inside the segment, repaint it, and we move the pointer out of it right the next empty cell.

Thus, we are actually using the DSU with heuristic compression ways, but without rank heuristics (because it is important, who will be the leader after the merger). Consequently, the asymptotic behavior of the final amount  $O(\log n)$  on request (however, small compared with 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, it is possible to implement this solution **with the rank heuristic**: we will store for each set in a array `end[]`, which is the set of ends (ie, the rightmost point). Then it will be possible to combine two sets to one in their ranking heuristics by placing them together to get a lot of 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 heuristics compression paths, any difficulties would not arise - the distance to the root just equal to the number of recursive calls that did function `find_set`.

However, as a result of compression paths several ribs path could squeeze into a single edge. Thus, with each vertex will have to store additional information: **length of the current edges from the vertex to the ancestor**.

When implementing convenient to represent the array `parent[]` function `find_set` now return more than one number, and a pair of numbers: the top of the 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, 1);
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

```

## Support for parity path length and the problem of verification of bipartite graphs 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 in a separate paragraph?

The fact is that usually demand parity storage path emerges in connection with the next **task** : given initially empty graph, it can be added to the edges, and do queries of the form "if the connected component containing a given vertex **bipartite** ?".

To solve this problem, we can start a system of disjoint sets of connected components for storage, and store each vertex parity path length to its leader. Thus, we can quickly check whether the addition of lead to a violation of the specified rib bipartite graph or not: namely, if the ends of the ribs lie in the same connected component, and thus have the same parity path length to the leader, the addition of the edge lead to the formation of a cycle of odd length and evolution of the current components in non-bipartite.

Home **complexity** with which we are confronted at the same time - this is what we must carefully considering parity, producing the union of two trees in the function `union_sets`.

If we add an edge  $(a, b)$  connecting the two components into one, then when joining one tree to another, we need to tell him this parity to result in peaks  $a$  and  $b$  would receive different parity path length.

Vyvedëm **formula** on who should receive this parity, we expose the leader of one of the set when connecting it to the leader of another set. We denote by  $x$ -parity path length from the top  $a$  to the leader it sets through  $y$ - parity path length from the top  $b$  to the leader of her set, and through  $t$ - the required parity, we have to put attachable leader. If the set with a vertex  $a$  joins with the top of the set  $b$ , becoming a subtree, then after joining at the top of  $b$  its parity does not change and remains equal  $y$ , and at the top of  $a$  the parity becomes equal to  $x \oplus t$ (a symbol  $\oplus$  here denotes the operation XOR (symmetric difference)). We require that these two differed parity, ie their XOR is unity. Ie we obtain the equation for  $t$ :

$$x \oplus t \oplus y = 1,$$

deciding which find:

$$t = x \oplus y \oplus 1.$$

Thus, regardless of which set is attached to which, the said formula should be used to set parity ribs conducted from one leader to the next.

We present the **implementation of DSU** with support parity. As in the previous paragraph, for convenience, we use a pair of storage ancestors and results of operations `find_set`. 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);
        bipartite[a] &= bipartite[b];
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

bool is_bipartite (int v) {
    return bipartite[ find_set(v) .first ];
}

```

## Algorithm for finding the RMQ (at least in the interval) for $O(\alpha(n))$ an average Offline

Formally, the problem is formulated as follows: it is necessary to implement a data structure that supports two types of queries: the addition of a specified number  $\text{insert}(i)$  ( $i = 1 \dots n$ ), and search and retrieval of the current minimum

number `extract_min()`. We assume that each number is added only once.

In addition, we believe that the entire sequence of requests is known to us in advance, ie, task - Offline.

**The idea of the solution** is as follows. Rather than take turns responding to each request, brute over the number  $i = 1 \dots n$  and determine the answer to a query, this number should be. To do this, we need to find the first unanswered request, coming after the addition `insert(i)` of this number - it is easy to understand that this is the request, the answer to that is a number  $i$ .

Thus, the idea here is obtained, similar to the **task of painting the segments**.

You can obtain a solution for  $O(\log n)$  an average of inquiry if we abandon the rank heuristic and will simply be stored in each element of the closest link to the right of the query `extract_min()`, and use path compression to maintain those links after the merger.

You can also get a solution and  $O(\alpha(n))$ , if we use the rank heuristic and will be stored in each set position number where it ends (that in the previous embodiment solutions achieved automatically due to the fact that the links were always just right - now it will be necessary stored explicitly).

## Algorithm for finding the LCA (lowest common ancestor in the tree) for $O(\alpha(n))$ an average Offline

Tarjan algorithm for finding the LCA for  $O(1)$  an average online is described in the relevant article . This algorithm compares favorably with other search algorithms LCA in its simplicity (especially compared to the optimal algorithm Farah-Colton-Bender ).

## Storage DSU as an explicit list of sets. The application of this idea at the confluence of various data structures

One of the alternative methods of storing DSU is to maintain each set as **explicitly stored list of its elements** . In this case, each element is also saved a reference to the representative (leader) of his set.

At first glance it seems that this 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 **heuristics weight** similar to that described above, can significantly reduce the asymptotic behavior of the operation: up  $O(m + n \log n)$  to perform  $m$  queries over the  $n$  elements.

Under the weight heuristic is understood that we are always **going to add the lesser of two more sets in** . Adding `union_sets()` one set to another is easy to implement in the time order of the size of the set to be added, and the search for a leader `find_set()` - a time  $O(1)$  with this method of storage.

We prove the **asymptotic behavior**  $O(m + n \log n)$  for performing  $m$  queries.

We fix an arbitrary element  $x$  and examine how it impacted the merge operation `union_sets`. When an element  $x$  exposed for the first time, we can say that the size of his new set will be at least 2. When  $x$  exposed a second time - it can be argued that he gets into a lot of size at least 4(as we add more to the smaller set). And so on - we see that an element  $x$  could affect up to  $\lceil \log n \rceil$  merge operations. Thus, in the sum over all vertices of this  $O(n \log n)$ , plus  $O(1)$  on every request - as required.

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 of more can be used outside of the DSU, to solve other problems.

For example, consider the following **problem** : given a tree, each leaf of which is attributed to any number (the same number may occur more than once in different leaves). Is required for each node of the tree to find out the number of different numbers in its subtree.

Applying for this problem the same idea, we can obtain a solution: Let `bypass_in_depth` on the tree, which will return a pointer to the `setnumbers` - a list of all the numbers in the subtree of this node. Then, to get an answer to the current node (unless, of course, it is not a leaf) - bypassing the need to call in the depth of all the children of this top and combine all received `setone`, the size of which will be the answer to the current node. To effectively combine multiple `setone` just

applies the above method: will combine the two sets by simply adding one element in a larger smaller set. As a result, we obtain a solution for  $O(n \log^2 n)$ , since the addition of one element set is made of  $O(\log n)$ .

## **Storage DSU preserving explicit tree structures. Perepodveshivanie. Search algorithm bridges in the graph of $O(\alpha(n))$ the average online**

One of the most powerful applications of data structures "of disjoint sets" is that it allows you to store the same time **as the compressed and uncompressed tree structure**. Compressed structure can be used for rapid association of trees and check for two vertices belonging to one tree, and uncompressed - for example, to search for the path between two given vertices, or other rounds of the tree structure.

When implemented, this means that in addition to the usual array of compressed DSU ancestors `parent[]` we have a normal, uncompressed, ancestors `real_parent[]`. It is clear that maintaining such an array does not worsen the asymptotic behavior: changes in it occur only when the two trees, and only one element.

On the other hand, in practical applications often requires two joining tree to learn said rib is not necessarily exiting from 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 the tree to the next, making the root of the tree of child nodes to be added to the second end edge.

At first glance, it appears that the operation perepodveshivaniya - very expensive and greatly worsen the asymptotic behavior. Indeed, for perepodveshivaniya tree for the top  $v$  we have to go from the vertex to the 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 to get the asymptotic behavior of one association, equal to  $O(\log n)$ the average.

For more details (including evidence asymptotics) cm. [search algorithm bridges in the graph of  \$O\(\log n\)\$ the average online](#).

## **Historical retrospective**

The data structure "a system of disjoint sets" was known relatively long time.

Way to store 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 combining rank, apparently developed

McIlroy (McIlroy) and Morris (Morris), and, independently, Tritter (Tritter).

Some time was known only score  $O(\log^* n)$  on a single operation, on average, Hopcroft and Ullman given in 1973 (Hopcroft, Ullman "Set-merging algomthms") - here  $\log^* n$ - **iterated logarithm** (it is slowly increasing function, but still not as slow as inverse Ackermann function).

First assessment of  $O(\alpha(n))$  where  $\alpha(n)$ - **inverse Ackermann function** - Tarjan got in his article in 1975 (Tarjan "Efficiency of a Good But Not Linear Set Union Algorithm"). Later in 1985, he, along with Lyuvenom got this evaluation time for several different heuristics rank and path compression methods (Tarjan, Leeuwen "Worst-Case Analysis of Set Union Algorithms").

Finally, Fredman and Saks in 1989 proved that the model adopted in the calculations **of any** algorithm for a system of disjoint sets must operate at least  $O(\alpha(n))$  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 temporary assessment and argues that a system of disjoint sets with compression heuristics ways and associations working in the rank of  $O(1)$  the average: Zhang "The Union-Find Problem Is Linear", Wu, Otoo " A Simpler Proof of the Average Case Complexity of Union-Find with Path Compression ".

## Tasks in the online judges

A list of tasks that can be solved by a system of disjoint sets:

- [TIMUS # 1671 "Web Anansi"](#) [Difficulty: Low]
- [Codeforces 25D "Roads not only in Berland"](#) [Difficulty: Medium]
- [TIMUS # 1003 "Parity"](#) [Difficulty: Medium]
- [SPOJ # 1442 "Chain"](#) [Difficulty: Medium]

## Literature

- Thomas feed, Charles Leiserson, Ronald Rivest, Clifford Stein. [Introduction to Algorithms](#) [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 Algorithms Set Union](#) [1985]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:00  
EDIT: 25 Oct 2011 21:31

## Wood pieces

Segment tree - a data structure that allows efficient (ie, the asymptotic behavior  $O(\log n)$ ) to implement the operation of the following form: finding the amount / minimum of array elements in a given interval ( $a[l \dots r]$  where  $l$  and  $r$  are input to the algorithm), thus further possible to change the elements of the array: as a change value for one element or elements on the change of the whole array subsegments (i.e., all elements are allowed to assign  $a[l \dots r]$  a value, or add all the elements of the array any number).

In general, the segment tree - a very flexible structure, and the number of problems solved it theoretically unlimited. In addition to the forms shown above operations with trees segments as possible and much more complex operations (see. Section "versions of sophistication 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 a given matrix podpryamougochnike (but only for the time already  $O(\log^2 n)$ ).

An important feature of the segment tree is that they consume a linear memory: the standard segment tree requires about  $4n$  memory elements to work on the array size  $n$ .

### Description of wood pieces in the base case

First, consider the simplest case of the tree segments - segment tree for sums. If you put the problem formally, then we have an array  $a[0..n - 1]$ , and our tree segments should be able to find the sum of elements from  $l$ th to  $r$ th (this request amount), as well as handle the change in the value of the specified element in the array, ie, actually respond to the assignment  $a[i] = x$ (this modification request). Once again, the tree segments should handle both of these queries over time  $O(\log n)$ .

### The tree structure of segments

So, what is a segment tree?

Count and remember somewhere sum of all elements of the array, ie, segment  $a[0 \dots n - 1]$ . Also calculate the sum 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 will divide in half, calculate and store the sum of them, then we will divide in half again, and so on until it reaches the current segment length 1. In other words, we start from the segment  $[0; n - 1]$ , and each time we divide the current segment in half (if it has not yet become a segment of unit length), then causing the same procedure on both halves; for each such segment we store the sum of numbers on it.

We can say that these segments in which we considered the amount, form a tree: the root of the tree - the segment  $[0 \dots n - 1]$ , and each vertex has exactly two sons (except the vertex-leaf, which has a length of the segment 1). Hence the name - "segment tree" (although the implementation is usually no tree is clearly not built, but more on that below in the implementation section).

So, we have described the structure of the tree segments. Immediately, we note that it has a **linear dimension**, namely, contains less  $2n$  vertices. This can be understood as follows: the first level of the tree segments contains one node (segment  $[0 \dots n - 1]$ ), the second layer - in the worst case, the two peaks, the third level in the worst case there will be four peaks, and so on, until the number of vertices is reached  $n$ . Thus, the number of vertices in the worst case estimated amount  $n + n/2 + n/4 + n/8 + \dots + 1 < 2n$ .

It is worth noting that other than powers of two, not all levels of the tree segments are completely filled. For example, when  $n = 3$  the left son of the root is a segment  $[0 \dots 1]$  having two children, while the right son of the root - the segment  $[2 \dots 2]$  being a sheet. No particular difficulties in implementing it is not, but nevertheless it should be borne in mind.

**The height of the tree** is of the segments  $O(\log n)$ - for example, because the length of the segment is at the root of the tree  $n$ , and when you go to one level down the length of the segments is reduced by about half.

### Building

### Contents [hide]

- Wood pieces
  - Description of wood pieces in the base case
    - The tree structure of segments
    - Building
    - Request amount
    - Update request
    - Implementation
  - Version of sophistication wood segments
    - More advanced features and inquiries
      - Minimum / maximum
      - Minimum / maximum value and the number of times it occurs
      - Search greatest common divisor / least common multiple
      - Counting the number of zeros, search  $k$ th zero
      - Search prefix array with a given sum
      - Search subsegment with the maximum amount
    - Saving the entire subarray in each node of the tree segments
    - Search the smallest integer greater than or equal to the specified value, the specified interval. No modification request
    - Search the smallest integer greater than or equal to the specified value, the specified interval. Permitted modification request
    - Search the smallest integer greater than or equal to the specified value, the specified interval. Acceleration using the technique of "partial cascading"
    - Other possible areas of
  - Update on the interval
    - The addition of the segment
    - Assigning the interval
    - The addition of the interval, the maximum request
    - Other Destinations
  - The generalization to higher dimensions
    - The two-dimensional segment tree in its simplest form
    - Compression of two-dimensional pieces of wood
    - Segment tree while preserving the history of its values (better to persistent-data structure)

The process of constructing the tree segments for a given array  $a$  can be done effectively as follows, from bottom to top: first write down the values of the elements  $a[i]$  in the corresponding leaves of the tree, then on the basis of these count values for the vertices of the previous level as the sum of the two leaves, then similarly calculate values for another level, etc. Convenient to describe this operation recursively: we run a procedure for constructing from the root segments, and the procedure of construction, if it is not caused by leaf, causing themselves from each of the two sons and summarizes the calculated values, and if it is caused by the sheet - it simply writes themselves the value of this element of the array.

Asymptotics tree construction segments will be so  $O(n)$ .

## Request amount

We now consider the request amount. Are input two numbers  $l$  and  $r$ , and we have for the time  $O(\log n)$  to calculate the sum of the numbers on the interval  $a[l \dots r]$ .

To do this, we will build a tree down the segments using to calculate the response previously counted amounts on each node of the tree. Initially, we get up in the root of the tree segments. Let's see in which of his two sons gets cut request  $[l \dots r]$  (recall that the sons of the root segments - it stretches  $[0 \dots n/2]$  and  $[n/2 + 1 \dots n - 1]$ ). There are two possibilities: that the segment  $[l \dots r]$  gets only one son of the root, and that, on the contrary, the segment intersects with two sons.

The first case is simple: just move on to that son, which is the length of our request, and we apply the algorithm described here to the current node.

In the second case we can not remain other choice but to go first to the left child and find the answer to the query in it, and then - go to the right child, find the answer in it and add it to our account. In other words, if the left son was cut  $[l_1 \dots r_1]$ , and the right - a segment  $[l_2 \dots r_2]$  (note that  $l_2 = r_1 + 1$ ), then we move on to the left child request  $[l \dots r_1]$ , and the right - with the request  $[l_2 \dots r]$ .

Thus, the amount of processing of the request is a **recursive function** that calls itself every time a son from the left or from the right (without changing the boundaries of the query in both cases), or by both at once (at the same time sharing our request for two relevant sub-query). However, recursive calls are not always going to do that if the current request coincided with the boundaries of the segment in the current top of the tree segments, then as a response will return the precomputed value of the sum on this segment recorded in the tree segments.

In other words, the calculation of the query is a descent of the tree segments, which is distributed to all the necessary branches of a tree, and for fast work has been used to count the sum of each segment in the segment tree.

Why is the **asymptotic behavior** of this algorithm will be  $O(\log n)$ ? To do this, look at each level of the tree segments as maximum lengths could visit our recursive function in the processing of a request. It is argued that such segments could not be more than four; then, given the estimate  $O(\log n)$  for the height of the tree, we obtain the required asymptotic behavior of the running time.

We show that this estimate of the four segments is true. In fact, at the zero level of the tree query affected only vertex - the root of the tree. Next on the first level recursive call in the worst case is divided into two recursive calls, but it is important here is that these two queries calls will coexist, ie the number of  $l$ 'query in the second recursive call is one more than the number of  $r$ 'query in the first recursive call. It follows that at the next level, each of the two calls could produce two more recursive call, but in this case, half of the non-recursive queries will work, taking the required value from the top of the tree segments. Thus, every time we will have no more than two really working branches recursion (we can say that one branch is close to the left edge of the request, and the second branch - to the right), but only the number of affected segments could not exceed the height of the tree segments multiplied by four, i.e. it is a number  $O(\log n)$ .

Finally, you can lead a working understanding of the amount of the request: the input segment  $[l \dots r]$  is divided into several subsegments, the answer to each of which has already calculated and stored in the tree. If you do it the right way partition, thanks to the tree structure of segments required number of subsegments will always be  $O(\log n)$  that and gives the efficiency of the wood pieces.

## Update request

Recall that the update request is received at the input index  $i$  value and  $x$ , and rearranges the tree segments so as to conform to the new value  $a[i] = x$ . The request must also be performed during  $O(\log n)$ .

This is more than a simple query compared with the request counting amount. The fact that the element  $a[i]$  involves only a relatively small number of vertices of the segments: namely in  $O(\log n)$  the tops - one on each level.

Then it is clear that the update request can be implemented as a recursive function: it is passed to the current top of the tree lines, and this function performs a recursive call from one of his two sons (of which contains the position  $i$  in its segment), and after that - counts the value of the sum in the current node in the same way as we did in the construction of wood segments (i.e., the sum of the values for both the current node sons).

## Implementation

The main follow-up time - is how to keep the tree in memory segments. For simplicity, we will not keep the tree in an explicit form, and we use this trick: we say that the root of the tree has a number of 1 his sons - the numbers 2 and 3 their sons - rooms 4 on 7, and so on. It is easy to understand the correctness of the following formula: if the vertex is numbered  $i$ , 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 of tree segments - now we do not need to be stored in the memory tree structure segments, and only make any array for sums on each segment of the tree segments.

One has only to note that the size of the array in such a numbering is not necessary to put  $2n$  and  $4n$ . The fact that this numbering does not work perfectly when  $n$  not a power of two - then there are missing numbers, which does not correspond to any tree nodes (actually numbering behaves just as if  $n$  would be rounded up to the nearest power of two). This does not create any difficulties in implementation, but leads to the fact that it is necessary to increase the size of the array to  $4n$ .

Thus, the segment tree, we **keep** just as an array  $t[]$ , the size of four times the size  $n$  of the input data:

```
int n, t[4*MAXN];
```

The procedure for **constructing the tree segments** for a given array  $a[]$  is as follows: it is a recursive function, it is passed to the array  $a[]$ , the number of  $v$  the current top of the tree, and the boundaries  $tl$  and  $tr$  the segment corresponding to the current top of the tree. The main program must call this function with 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 to **query the amount** also represents a recursive function in the same way that the information is transferred to the current top of the tree (ie, the number of  $v, tl, tr$  which in the main program should pass  $1, 0, n - 1$  respectively), and in addition to this - also border  $l$  and  $r$  the current request. In order to simplify this code fukntsya always makes two recursive calls, even if actually need one - just extra recursive call will request, in which  $l > r$  it is easy to cut off additional check at 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, a **modification request**. He just passed information about the current top of the tree segments, and additionally, 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 noting that the function `update` is easy to make non-recursive, because it tail recursion, ie branching never occurs: one call can generate only one recursive call. When non-recursive implementation, speed can grow several times.

Other **optimizations** it is worth mentioning that the multiplication and division by two is necessary to replace Boolean operations - it is also slightly improves the performance of the tree segments.

## Version of sophistication wood segments

Segment tree - a very flexible structure, and allows you to make generalizations in many different directions. Let's try to classify them below.

## More advanced features and inquiries

Improvements wood segments in this direction may be quite obvious (as in the case of the minimum / maximum value instead of the amount) and a very, very nontrivial.

### Minimum / maximum

Little to change the terms of the problem described above: instead of requesting the sum will produce now request the minimum / maximum on the interval.

Then the segment tree for such a task does not differ from the tree segments described above. Just need to change the method of calculating  $t[v]$  in functions `build` and `update`, as well as the calculation of the returned response function `sum` (to replace the summation on the minimum / maximum).

### Minimum / maximum value and the number of times it occurs

The task is similar to the previous one, but now in addition to the maximum required and return the number of its occurrences. This problem arises in a natural way, for example, in the solution of using wood pieces such task: to find the number of the longest increasing subsequence in the specified array.

To solve this problem at each node of the tree segments will store a pair of numbers: in addition to the maximum number of its occurrences in the corresponding interval. Then the construction of the tree we should just two such pairs obtained from the sons of the current node, get a pair for the current node.

The union of two such pairs one stands out in a separate function because this operation will have to produce and modify the query, and the query search for a 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]);
    }
}

```

## Search greatest common divisor / least common multiple

If we want to learn to look for NOD / NOC all the numbers in a given segment of the array.

It's pretty interesting generalization tree segments obtained in exactly the same way as trees segments for the amount / minimum / maximum: simply stored in each node of the tree NOD / NOC all the numbers in the corresponding segment of the array.

## Counting the number of zeros, search $k$ -th zero

In this task, we want to learn to respond to the request a predetermined number of zeros in the segment array, and finding a request  $k$ -th element zero.

Again slightly modify the data stored in the tree segments: will now be stored in an array of  $t$  the number of zeros that occur in their respective segments of the array. Understand how to maintain and use the data in the 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  $k$  the zero-th entry in the array. To do this we go down the tree segments, starting with the root and each time moving in a left or right child, depending on which of the segments is required  $k$  th zero. In fact, to understand what we need to pass a son, just look at the value recorded in the left son: if it is greater than or equal to  $k$ , the need to move to the left son (because it has at least a segment  $k$  of zeros), but otherwise - go to the right child.

In the case of the implementation can be cut off when  $k$  there is no zero-th, even when entering a 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

The challenge is this: is required by this value  $x$  quickly find such  $i$  that the sum of the first  $i$  element of the array  $a$  is greater than or equal to  $x$  (assuming that the array  $a$  contains only non-negative numbers).

This problem can be solved by binary search, calculating every once in a sum for a particular prefix array, but it will lead to the solution of the time  $O(\log^2 n)$ .

Instead, you can use the same idea as in the previous paragraph, and look for the desired position of a descent on the wood turning every once in a left or right child, depending on the value of the sum in the left son. Then the response to the search request would be one such descent of the tree, and therefore, will be carried over  $O(\log n)$ .

## Search subsegment with the maximum amount

Is still given to the input array  $a[0 \dots n - 1]$ , and receives requests  $(l, r)$ , which means: to find a subsegment  $a[l' \dots r']$  that  $l \leq l'$ ,  $r' \leq r$  and the sum of the length of  $a[l' \dots r']$  the maximum. Request for modification of individual elements of the array are allowed. Array elements may be negative (and, for example, if all the numbers are negative, the best subsegments will be empty - on it is zero sum).

This is a very non-trivial generalization of tree lengths obtained as follows. Will be stored in each node of the tree segments of four values: the amount on this interval, the maximum amount of all the prefixes of this segment, the maximum amount of all suffixes, as well as the maximum amount subsegment on it. In other words, for each segment of the tree segments response to it already predposchitan, as well as additional responses counted among all segments abutting against the left boundary of the segment, as well as among all the segments that are limited to the right border.

How do you build a tree segments such data? Again, get to it with a recursive point of view: if the current node for all four values in the left son and son in law had already counted, count them now for the summit. Note that the answer is in the very top:

- a response in the left son, which means that the best subsegment in the current top fits into the segment of the left son,
- a response in the right son, which means that the best subsegment in the current top fits into the segment of the right son,
- or the maximum amount of the suffix in the left son and the maximum prefix in the right son, which means that the best subsegment is its beginning in the left son, and the end - to the right.

So, the answer to the current node is the maximum of these three variables. Recalculate the maximum amount for the same prefix and suffix even easier. We present the implementation of a function `combine` that, given two structures `data`, containing the data on the left and right sons, and that returns the data in the current top.

```

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;
}

```

Thus, we learned how to build a tree segments. Hence it is easy to obtain and implement query modification: as in the simple tree segments, we do recalculation of values in all segments of changed the tree tops, which all use the same function `combine`. To calculate the values in the tree leaves as an auxiliary function `make_data`, which returns a structure `data` as calculated by a single number `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 also, as before, we go down the tree, thereby breaking the query interval  $[l \dots r]$  for a few subsegments, coincides with the segment tree segments, and merge the answers to them in a single answer to the whole problem. Then it is clear that the work is no different from ordinary wood work pieces, rather than just have a simple summation / minimum / maximum values use the function `combine`. The below implementation is slightly different from the implementation of the query `sum`: it does not allow for cases where the left boundary of the query exceeds the right border  $r$  (otherwise any unpleasant incidents - what structure `data` comes back when the queue is empty segment? ..).

```

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)
    );
}

```

## Saving the entire subarray in each node of the tree segments

This is a separate subsection, standing apart from the rest, since at each node of the tree segments, we will not keep some concise information on this subsegments (sum, minimum, maximum, etc.), and all elements of the array, which lie in the subsegments. Thus, the root of the tree segments will store all elements of the array, the left son of the root - the first half of the array, the right son 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 the numbers found in the corresponding interval. In more complex versions are not stored lists, and any data structures 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 that has a memory size of the order of the corresponding segment.

The first natural question arising when considering trees segments of this class - this is the amount of memory consumed . It is argued that if each node of the tree contains a list of all segments found on this segment of numbers, or any other data structure size of the same order, the sum of all segment tree will occupy  $O(n \log n)$  memory. Why is this so? Because each number  $a[i]$  falls into  $O(\log n)$  pieces of wood segments (not least because the height of the tree segments there  $O(\log n)$ ).

So, despite the apparent extravagance of the tree segments, it consumes memory is not much more than the usual wood segments.

Below described are some typical uses such a data structure. It should be noted immediately clear analogy trees segments of this type with **two-dimensional data structures** (in fact, in a sense, this is a two-dimensional data structure, but with a rather limited possibilities).

### Search the smallest integer greater than or equal to the specified value, the specified interval. No modification request

Required to respond to requests from the following:  $(l, r, x)$  that means finding the minimum number in the interval  $a[l \dots r]$  that is greater than or equal to  $x$ .

We construct a segment tree, in which each vertex will store the sorted list of all the numbers appearing on the corresponding interval. For example, the root will contain an array  $a[]$  in sorted order. How to build a segment tree as efficiently as possible? To do this, come to the problem, as usual, in terms of recursion: if the left and right children of the current node, these lists have already been built, 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 through it with two pointers. C ++ users even easier, because the 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(),
               back_inserter (t[v]));
    }
}
```

We already know that the constructed thus segment tree will take up  $O(n \log n)$  memory. And with such an implementation time of its construction also has value  $O(n \log n)$  - because every list is constructed in linear time with respect to its size. (By the way, here there is an obvious analogy with the algorithm **merge sort** : but here we keep the information from all stages of the algorithm, not just the outcome.)

Now consider the **response to the request** . Let's go down the tree, as does the standard response to a request in the tree lines, breaking our segment  $a[l \dots r]$  on several subsegments (some  $O(\log n)$  units). It is clear that the answer to the whole problem is minimized among the responses to each of these subsegments. Let us understand now how to respond to a request for one such subsegments, coinciding with a vertex of the tree.

So we came to some vertex of the tree segments and want to find the answer to it, ie, find the smallest number greater than or equal to this  $x$ . To do this we just need to perform a **binary search** on the list, count in this top of the tree and return the first number on the list, is greater than or equal to  $x$ .

Thus, the answer to the query in one subsegments takes place  $O(\log n)$ , and the entire request is being processed at the 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` equal to some large number, certainly more than any number in the array. It carries the meaning of "response in a given segment does not exist."

### Search the smallest integer greater than or equal to the specified value, the specified interval. Permitted modification request

The task is similar to the previous one, but now resolved modification requests: assignment process  $a[i] = y$ .

The decision is also similar to the solution of the previous problem, but instead of simple lists at each node of the tree segments, we will keep a balanced list, which allows you to quickly search for the required number, delete it and insert a new number. Given that the general number in the input array may be repeated, the best choice is a data structure STL `multiset`.

**The construction** of such a tree segments occurs about the same as in the previous problem, but now we must unite not sorted lists, and `multiset` that will lead to the fact that the asymptotic behavior of building to deteriorate  $n \log^2 n$  (although, apparently, red-black trees allow to merge the two trees in linear time, but the STL does not guarantee).

Response to a **search request** has practically equivalent to the code above, but now `lower_bound` need to call on `t[v]`.

Finally, a **modification request**. To handle it, we have to go down the tree by making changes to all  $O(\log n)$  lists containing affect a component. We simply remove the old value of this element (not forgetting that we do not need to remove it all together with repetitions of this number) 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;
}
}

```

Processing this request also occurs during  $O(\log^2 n)$ .

### Search the smallest integer greater than or equal to the specified value, the specified interval. Acceleration using the technique of "partial cascading"

Improve response time to the search time  $O(\log n)$  by applying the technique of "partial cascading" ("fractional Cascading").

Partial cascading - this is a simple technique that helps to improve the work of several binary search 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 by the number of binary search  $x$ . Partial cascading allows you to replace all the binary search for one.

The simplest and most obvious example is the partial cascading **following problem**: there are several sorted lists of numbers, and we should find in each list the first number is greater than or equal to the specified value.

If we solve the problem "head" that would have to run a binary search on each of these lists, that if a lot of these lists, it becomes a very important factor: if the entire list  $k$ , the asymptotic behavior happens  $O(k \log(n/k))$  where  $n$  - the total size of all the lists (asymptotic behavior is because the worst case - when all lists are approximately equal in length to each other, ie equal  $n/k$ ).

Instead, we could combine all of these lists into one sorted list in which each number  $n$  will keep a list of position: the position in the first list, the first number is greater than or equal to  $n$ , a similar position in the second list, and so on. In other words, for each occurrence of the number we store at the same time the number of results binary search on it in each list. In this case, the asymptotic behavior of the answer to the query is obtained  $O(\log n + k)$ , which is much better, but we are forced to pay a large consumption of memory: namely, we need  $O(nk)$  memory.

Appliances partial cascading goes further in solving this problem and is working memory consumption  $O(n)$  at the same time respond to the request  $O(\log n + k)$ . (To do this, we keep not one big list length  $n$ , and come back to  $k$  the list, but with each list contains every second element from the following list, we again have with each number to record its position in both lists (current and next), but it will continue to effectively respond to the request: we do a binary search on the first list, and then go

to these lists in order, each time passing in the following list using predposchitannyy pointers, and taking one step to the left, thereby taking into account that half the following list of numbers was not taken into account).

But we in our application to the tree line segments **do not need** the full power of this technique. The fact that the current list contains all of the top, 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 the number of its position in the list of left and right sons (more precisely, the position of the first day, less than or equal to the current).

Thus, instead of the usual list of all the numbers we keep a list of triples: the number itself, the position in the list of the left son, the position in the list, right son. This will allow us for  $O(1)$  the list to determine the position of the left or right child, instead of doing a binary list on it.

The easiest way to apply this technique to the problem when there is no modification request - then these positions are simply numbers and counting them in the construction of the tree is very easy within the algorithm merge two sorted sequences.

In the event that allowed modification requests, everything becomes more complicated: these positions now be stored in the form of iterators inside multiset, and when you request an update - the right to decrease / increase for those items for which it is required.

Anyway, the problem has already been reduced to net realizable subtleties, but the basic idea - replacing  $O(\log n)$  a binary search binary search through the list in the root of the tree - fully described.

### Other possible areas of

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 applications using `vector` and `multiset`, while generally used may be any other compact data structure: Other segment tree (this is a little discussed below in the section on multivariate trees segments), [Fenwick tree](#) , [Treap](#) etc.

## Update on the interval

We have considered only a problem when the modification request affects only one element of the array. In fact, the segment tree allows queries that apply to whole segments of contiguous elements, with fulfill these requests during the same time  $O(\log n)$ .

### The addition of the segment

To begin consideration of such trees segments with the simplest case: modification request is the addition of all the numbers in some subsegments  $a[l \dots r]$  of a number  $x$ . Read request - continue reading the value of a number  $a[i]$ .

To make a request for the addition of efficient 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, "added to the entire array  $a[0 \dots n - 1]$  number 2", we will deliver at the root number 2. Thus we will be able to process a request for the addition of any subsegments effectively, rather than to change all  $O(n)$  values.

If a request comes now read the value of a number, it is sufficient to go down the tree, summing all encountered on the way the values recorded 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 modification request is assigned to all the elements of a certain length of  $a[l \dots r]$  some value  $p$ . As a second request will be considered read the array values  $a[i]$ .

To make modifications to the whole segment will have at each node of the tree segments stored, whether painted this piece 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 of changing the values in the set of vertices of segments, changing only some of them, leaving flags "painted" for the other segments, which means that this whole segment together with its subsegments to be painted in this color.

So, after the query modification segment tree becomes, generally speaking, irrelevant - there were shortfalls in some modifications.

For example, if the request arrived modification "to assign the entire set  $a[0 \dots n - 1]$  some number ", the tree segments we will only change - mark the root of the tree that he painted entirely in that number. The rest of the top of the tree will remain unaltered, even though it is actually the tree should 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  $a[0 \dots n/2]$  at any other number. To handle such a request, we need to paint the entire left child of the root in this new color, but before you do that, we must deal with the root of the tree. Subtlety here is that should remain in the tree that the right half is colored in the old color, and currently no tree in the right-half data is not stored.

Output is as follows: make **pushing** information from the root, ie if the root of the tree was painted in any number, the color in the number of its right and left a son, and from the root to remove this mark. After that, we can safely paint left child of the root, without losing any relevant information.

Summing up, we obtain for any queries with the tree (request modification or reading) while descending the tree, we should always do the pushing of information from the current node in both of her sons. You can understand it so that when descending the tree we use lagging modification, but only as much as necessary (not to worsen with the asymptotic behavior  $O(\log n)$ ).

When implemented, this means that we need to make a function **push** that will be transmitted top of the tree lines, and it will produce pushing information from this vertex in both her sons. Call this function should be at the very beginning of the function **query** (but do not call it from the leaves, because of the push plate information is not necessary, and nowhere else).

```

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, l, 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 not do it delayed the update, and immediately return a response as soon as it enters the top of the tree segments, entirely painted in a particular color.

## The addition of the interval, the maximum request

Now let modification request will again request the addition of all numbers a subsegment of the same number, and read request is to find the maximum in some subsegments.

Then at each node of the tree segments will have to additionally store up to all this subsegments. But subtlety here is how to recalculate the values.

For example, suppose there was a request "added to the entire first half, i.e.  $a[0 \dots n/2]$ , number 2 ". Then it will be reflected in the tree record number 2 in the left child of the root. How to calculate the new value is now high on the left, and his son at the root? Here it becomes important not to get confused - what the maximum is stored in the top of the tree: maximum without considering adding on top of all this, or considering it. You can choose any of these approaches, but the main thing - to consistently use it anywhere. For example, if the first approach, the maximum will be obtained at the root of a maximum of two numbers: the maximum in the left son, plus the addition of the left son, and the son of a maximum in the right plus the addition in it. In the second approach is at the root of the maximum will be obtained as the addition of a root plus a maximum of the peaks in the left and right sons.

## Other Destinations

There were considered only the basic application segments trees in problems with modifications on the segment. The remaining tasks are obtained based on the same ideas that are described here.

It is only important to be very careful when dealing with pending modifications: it must be remembered that even if the current top we have "pushed" a modification is pending, then the left and right sons, most likely, have not done so. Therefore it is often necessary to call `is_pushalso` on the left and right children of the current node, or else to carefully consider the pending modifications to them.

## The generalization to higher dimensions

Segment tree is generalized quite naturally on the two-dimensional and multi-dimensional case at all. If the one-dimensional case we broke array indexes into segments, the two-dimensional case will now first break all on the first index, and for each segment on the first index - to build an ordinary tree segments for the second index. Thus, the basic idea of the solution - a segment tree for insertion of the second index into the wood pieces on the first index.

Let us illustrate this idea by the example of a specific task.

### The two-dimensional segment tree in its simplest form

Dana rectangular matrix  $a[0 \dots n - 1, 0 \dots m - 1]$ , and enter search queries amount (or minimum / maximum) on some podpryamougnikah  $a[x_1 \dots x_2, y_1 \dots y_2]$  and requests modifications 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 ( $x$ ), and then - for the second ( $y$ ).

To the process of building more understandable, it is possible to forget that the original two-dimensional array has been, and leave only the first coordinate. We will construct the usual one-dimensional segment tree, working only with the first coordinate. But as the value of each segment, we will not record a certain number, as in the one-dimensional case, and the whole tree segments: ie at this point we remember that we have more and the second coordinate; but since at this point already recorded that the first coordinate is a segment  $[l \dots r]$ , we actually work with the band  $a[l \dots r, 0 \dots m - 1]$ , and it builds 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 in the coordinate  $x$  (`build_x`) and the coordinate  $y$  (`build_y`). If the first function is almost no different from the usual one-dimensional tree, the latter is forced to deal separately with the two cases: when the current segment in the first coordinate ( $[tlx \dots trx]$ ) has unit length, and when - a length greater than one. In the first case, we simply take the required value from the matrix  $a[][],$  and the 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, lx, mx);
        build_x (vx, mx+1, rx);
        t[vx] = t[vx][mx] + t[vx][mx+1];
    }
}

```

```

        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 space, but with a more constant:  $16nm$  memory cells. It is clear that it is constructed in the above procedure `build_x` also in linear time.

We now turn to the **query processing**. Respond to a two-dimensional query will on the same principle: first break request to the first coordinate, and then when we got to the top of some wood pieces in the first coordinate - initiate a request from the relevant sections of the tree on 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  $O(\log n \log m)$  since she first down on a tree in the first coordinate, and each passed the top of the tree - makes a request of a conventional wood pieces on the second coordinate.

Finally, consider the **modification request**. We want to learn how to modify the tree segments in accordance with the change of the value of any element  $a[x][y] = p$ . It is clear that changes will occur only in the top of the first wood segments that cover the coordinate  $x$  (and they will  $O(\log n)$ ), and for trees segments corresponding to them - the changes will be only those tops that cover the coordinate  $y$  (and there is  $O(\log m)$ ). Therefore, the implementation of the modification request will not be much different from the one-dimensional case, but now we will 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 pieces of wood

Suppose that the problem is as follows: there are  $n$  points in the plane defined by its coordinates  $(x_i, y_i)$ , and get requests like "count the number of points lying in a box  $((x_1, y_1), (x_2, y_2))$ ". It is clear that in the case of such a task becomes unnecessarily wasteful to build a two-dimensional segment tree with  $O(n^2)$  elements. Much of this memory will be wasted, because every single point can be reached only in the  $O(\log n)$  segments of wood pieces in the first coordinate and, therefore, the total "useful" size of all segments of the trees on the second coordinate is the value  $O(n \log n)$ .

Then proceed as follows: at each node of the tree segments in the first coordinate will be stored segment tree built only on the second coordinates, which are found in the current segment of the first frame. In other words, the construction of the tree segments within some vertex with the number  $vx$  and boundaries  $tlx, trx$ , we consider only those points that fall in this segment  $x \in [tlx; trx]$ , and build a segment tree just above them.

In this way we will achieve that every tree segments the second coordinate will occupy as much memory as it should. As a result, the total **amount of memory** decreases to  $O(n \log n)$ . **Responding to a request**, we will continue for  $O(\log^2 n)$  just now in the call request from the wood pieces on the 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 will be impossible to make an arbitrary **modification request**: in fact, if a new point, then it will lead to the fact that we will have in any tree segments the second coordinate add a new element into the middle, impossible to do that effectively.

In conclusion, we note that a concise manner described two-dimensional segment tree is practically **equivalent to** the above-described modification of the one-dimensional tree segments (see. "Saving the entire subarray in each node of the tree segments"). In particular, it turns out that the herein described two-dimensional segment tree - it's just a special case of the subarray conservation at each vertex of the tree where the subarray is stored in a tree segments. It follows that if you have to abandon the two-dimensional pieces of wood because of impossibility of performance of a query, it makes sense to try to replace the embedded tree segments in any more powerful data structure, for example, [the Cartesian tree](#).

## Segment tree while preserving the history of its values (better to persistent-data structure)

Persistent-data structure called such a data structure that stores every modification of 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 the data structures that can be converted into a persistent-data structure (of course, we consider the persistent-efficient structure, and not one that copies all himself entirely before each update).

In fact, any change in the query tree segments leads to a change in data  $O(\log n)$  peaks, moreover along a path starting from the root. So, if we keep the tree segments in the index (ie pointers to the left and right sons do pointers stored at the top), you can view the update we should just change instead of having a vertex to create new vertices, links of which are directed to the old top. Thus, when requesting an update will create  $O(\log n)$  new peaks, including will create a new root of the tree segments, and all prev version tree, suspended for the old root, will remain unchanged.

Here is an example implementation of the simplest pieces of wood when there is only a request for calculation of 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;
    return get_sum (t->l, tl, tm, l, min(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)
        );
}

```

With this approach can be turned into persistent-data structure virtually any segment tree.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:01  
EDIT: 9 Mar 2013 1:37

## Treap (treap, deramida)

Treap - a data structure that combines binary search tree and a 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  $Y$ . Assuming that all  $X$  and  $Y$  are all different, we see that if the tree contains an element ( $X_0, Y_0$ ), then all elements in the left subtree of  $X < X_0$ , all the elements in the right subtree  $X > X_0$ , and also in the left and right subtree have:  $Y < Y_0$ .

Deramidy been proposed Seidel (Siedel) and Aragon (Aragon) in 1989

### The advantages of such an organization of data

In the application, we consider (we consider deramidy as Cartesian tree - it is actually more general data structure),  $X$ s are the keys (and also the values stored in the data structure), and  $Y$ 'i - called **priorities**. If priority was not, it would be normal binary search tree in  $X$ , and a given set of  $X$  starts up could correspond to a lot of trees, some of which are degenerate (eg, in the form of a chain), and therefore is extremely slow (the main operations would be carried out for  $O(N)$ ).

At the same time, **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 by the corresponding theorem). Now obviously, if **accidentally choose priorities**, then we will achieve this construct **degenerate** trees in the average case, providing the asymptotic behavior of  $O(\log N)$  on average. Hence it is clear another name of this data structure - **a randomized binary search tree**.

### Operations

Thus, treap provides the following operations:

- **Insert ( $X, Y$ )** - for  $O(\log N)$  on average  
Performs adding a new element to the tree.  
The variant in which the priority value  $Y$  is not passed to the function, and randomly selected (but bear in mind that it should not coincide with any any other tree  $Y$ ).
- **Search ( $X$ )** - for  $O(\log N)$  on average  
Searches for an item with the specified key value  $X$ . implemented in exactly the same way as for an ordinary binary search tree.
- **Erase ( $X$ )** - for  $O(\log N)$  on average  
Seeks element and removes it from the tree.
- **Build ( $X_1, \dots, X_N$ )** - for the  $O(N)$   
tree is constructed from the list of values. This operation can be implemented in linear time (assuming that the values of  $X_1, \dots, X_N$  sorted), but here, this implementation will not be considered.  
It will be used only simple realization - in the form of successive calls Insert, ie for  $O(N \log N)$ .
- **Union ( $T_1, T_2$ )** - for the  $O(M \log(N/M))$  in average  
Merges two wood, assuming that all elements are different (though this operation can be realized with the same asymptotic behavior when combined if necessary to remove recurring items).
- **Intersect ( $T_1, T_2$ )** - for the  $O(M \log(N/M))$  on average  
finds the intersection of the two trees (i.e. they are common elements). Here is the implementation of this operation will not be considered.

### Contents [hide]

- Treap (treap, deramida)
  - The advantages of such an organization of data
  - Operations
  - Description of the implementation of
  - Implementation
  - Support sizes of subtrees
  - Construction of Cartesian tree for  $O(N)$  in the offline
  - Implicit Treap

Furthermore, due to the fact that wood is a Cartesian binary search tree and their meanings, are applicable thereto operations such as finding the K-th largest element on the other hand, number of the element definition.

## Description of the implementation of

From the point of view of implementation, each element contains X, Y, and pointers to the left L and right R son.

To implement the operations needed to implement two additional operations: Split and Merge.

**Split (T, X)** - separates two wood tree T L and R (which are returned) so that L contains all elements smaller key X, R and contains all the elements, large X. This operation is performed for the O ( $\log N$ ). Implementation of it is quite simple - the obvious recursion.

**Merge (T<sub>1</sub>, T<sub>2</sub>)** - combines two subtrees T<sub>1</sub> and T<sub>2</sub>, and returns the new tree. This operation is also implemented for the O ( $\log N$ ). It works under the assumption that T<sub>1</sub> and T<sub>2</sub> have the appropriate order (all the values of X smaller than the first value in the second X). Thus, we need to combine them so as not to disturb the order of priorities for Y. To do this, simply select it as the root of a tree, wherein Y is fundamentally more and recursively calls itself from another tree and a corresponding son selected tree.

Now the obvious realization **Insert (X, Y)**. First down on a tree (as in a normal binary search tree on X), but stops at the first element, in which the priority value was less than Y. We found a position where we will insert our element. Now call Split (X) from the found element (the element together with all its subtree) and returns its L and R recorded as left and right child element to be added.

It is also understood and implementation **Erase (X)**. Go down the tree (as in a normal binary search tree on X), seeking entry to be removed. Find the items, we simply call the Merge from his left and right sons, and the return value put in place the item to remove.

Operation **Build** sell for O ( $N \log N$ ) simply by successive calls Insert.

Finally, the operation **Union (T<sub>1</sub>, T<sub>2</sub>)**. Theoretically, its asymptotic behavior of O ( $M \log (N / M)$ ), but in practice it works very well, probably with a very small hidden constant. Suppose, without loss of generality, T<sub>1</sub> -> Y > T<sub>2</sub> -> Y, ie, root of T<sub>1</sub> is the root of the result. To get results, we need to combine trees T<sub>1</sub> -> L, T<sub>1</sub> -> R and T<sub>2</sub> in two of wood, that they could do the sons of T<sub>1</sub>. To do this, call the Split (T<sub>2</sub>, T<sub>1</sub> -> X), thus we razobëm T<sub>2</sub> into two halves L and R, which are then recursively combine sons T<sub>1</sub> : Union (T<sub>1</sub> -> L, L) and Union (T<sub>1</sub> -> R, R), thus we will build the left and right subtrees result.

## Implementation

We sell all the operations described above. Here, for convenience, we introduce 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 nodes in its subtree - a certain field in the structure int cnt item. For example, it can easily be found for the O ( $\log N$ ) K-th largest element of wood, or, conversely, for the same asymptotics know the item number in the sorted list (the implementation of these operations will not differ from their realization for conventional binary search trees).

If you change the tree (adding or removing items, etc.) must vary accordingly and cnt some vertices. Define two functions - function cnt () will return the current value of cnt, or 0 if the node does not exist, and the 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, enough to add function calls upd\_cnt () at the end of each function, insert, erase, split, merge, in order 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 for O (N) in the offline

TODO

### Implicit Treap

Implicit Cartesian tree - a simple modification of the conventional Cartesian tree, which, however, is a very powerful data structure. In fact, implicit Cartesian tree can be thought of as an array, on which you can implement the following (all of O ( $\log N$ ) online):

- Inserting an element in an array in any position
- Removal of any element
- Sum, minimum / maximum on an arbitrary interval, etc.
- Addition, painting on an interval
- Coup (permutation of the elements in reverse order) on the interval

The key idea is that the key as keys to be used **indexes** in the array elements. However, these values are explicitly stored key, we will not (otherwise, for example, when you insert the element would have to change the key in O ( $N$ ) vertices of the tree).

Note that in this case is actually the key to some peaks - is the number of vertices less than it. It should be noted that the vertices of the smaller, not only are its left subtree, and possibly in the left subtree of its ancestors. More strictly, **the implicit key** top for a number of vertices equal to t cnt ( $t \rightarrow l$ ) in the left subtree of this node plus the analogous values cnt ( $p \rightarrow l$ ) +1 for each ancestor of the vertex p, provided that t is a the right subtree of p.

Clearly, as I quickly calculated for the current top of her implicit key. As in all operations, we arrive at any vertex down the tree, we can simply accumulate this amount, transferring its functions. If we go to the left subtree - accumulated amount does not change, and if we go to the right - increases by cnt ( $t \rightarrow l$ ) +1.

We give 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 additional operations on implicit Treap:

- **Insert** element.

Let us have to insert the element in position pos. We divide the Cartesian tree into two halves: the corresponding 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 wood with a new vertex  $t_1$ ; it is sufficient to cause the merge ( $t_1, t_1, new\_item$ ) (not hard to see that all the preconditions for the merge performed). Finally, combine the two trees  $t_1$  and  $t_2$  back into a tree  $t$  - calling merge ( $t, t_1, t_2$ ).

- **Removing** the element.

There is still easier: just find a removable element, and then perform a merge to his sons, l and r, and put in place the result of combining the top of t. In fact, the removal of the implicit Treap not differ from the removal of a conventional Cartesian tree.

- **Sum / low**, etc. on the segment.

First, for each vertex will create an additional field f in the structure item, that will store the value of the objective function for the subtree of this node. Such a field is easy to maintain, it is necessary to act similarly supported size cnt (create a function that computes the value of this field, taking advantage of its significance 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 be responsible a request for an arbitrary interval [A; B]. Learn how to allocate a portion of the tree corresponding to the interval [A; B]. It is not difficult to understand that it is enough to cause a first split (t, t1, t2, A), and then split (t2, t2, t3, B-A + 1). As a result of the tree and t2 will consist of all the elements in the interval [A; B], and only them. Therefore, the answer to the query would be a top field f t2. After the response to the request to restore the necessary call tree merge (t, t1, t2) and merge (t, t, t3).

- **Addition / painting** on the segment.

Here we are doing in the previous paragraph, but instead of the field f field will store add, which will contain Adds value (or value, in which color all the vertices of the subtree). Before performing any operation it is necessary to add this value to "push" - ie, amended accordingly tl-> add and t-> r-> add, and add value at take off. In this way we will achieve that in any changes in the tree information will not be lost.

- **Coup** on the segment.

This item is almost similar to the previous - you need to enter the field bool rev, which put to true, when you want to make a revolution in the subtree of the current node. "Push" field rev is that we exchange places sons current node, and set this flag for them.

**Realization**. Let us give an example for the full realization of implicit Cartesian tree with the coup on the segment. Here each vertex is also stored field value - the actual value of the item standing in the array at the current position. Also shows the implementation of the function output (), which takes an array that corresponds to the current state of the 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);
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 30 Oct 2008 23:24  
EDIT: 20 Aug 2012 23:55

## Modification of the stack and queue for finding the minimum in O (1)

Here we look at three things: the modification of the stack with the addition of finding the smallest element in O (1), a similar modification of the queue, as well as their application to the problem of finding the minimum in all subsegments of fixed-length array of O (N).

### Contents [hide]

- Modification of the stack and queue for finding the minimum in O (1)
  - Modification of the stack
  - Modification of the queue. Method 1
  - Modification of the queue. Method 2
  - The problem of finding a minimum in all subsegments of fixed length of the array

### Modification of the stack

Want to add the possibility of finding the minimum of the stack in O (1), while maintaining the same asymptotic behavior of adding and removing elements from the stack.

To do this, we will not be stored on the stack elements themselves, and couples: element and at least on the stack, since this element and below. In other words, if the stack is present as an array of pairs,

```
stack [i] .second = min {stack [j] .first}
                      j = 0..i
```

It is clear that while finding a minimum throughout the stack would be simply to take the value of 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). Deleting an element from the stack is no different from the usual removal of the stack 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 way to modify the queue, but has the disadvantage that the modified queue can actually store, not all elements (ie, when 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 can be applied to the problem of finding a minimum in subsegments (see. Below).

The key idea is to actually store queue elements, not all, but only to determine the needed minimum. Namely, let turn a non-decreasing sequence of numbers (ie, the head is kept the lowest value), and, of course, is not arbitrary, but always contain a minimum. Then at least the entire queue will always be the first of its elements. Before adding a new element to the queue is sufficient to produce a "cut-off": while in the tail of the queue

element is greater new element will remove this element from the queue; then add a new element to the end of the queue. Thus we have, on one side, not disturb the order, and on the other hand, do not lose the current element if at any subsequent step will be a minimum. But when you remove 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 extracted element - if the item with that value is in the head of the queue, then extract it; or simply do nothing.

Consider the implementation of the above operations:

```
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 understood that the average execution time of all these operations is O (1).

## Modification of the queue. Method 2

Here we consider another way of modifying the queue for finding the minimum of O (1), which is slightly more complicated to implement, but devoid of the main drawbacks of the previous method: all the actual queue elements are stored therein and, in particular, when removing the element it is not necessary 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 how to model queue using two stacks.

Zavedem two stacks: s1 and s2; of course, refers to the stacks, modified for finding the minimum in O (1). Add new elements will always be on the stack s1, and removing elements - only from the stack s2. At the same time, if you try to retrieve the item from the stack s2 it was empty, just transfer the all elements from stack to stack s1 s2 (with the elements in the stack s2 are obtained in the reverse order that we need to extract the elements; s1 stack will become blank). Finally, finding 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 1 time added to the stack s1, 1 time transferred to the stack s2, and 1 time 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 of fixed length of the array

Suppose we are given an array A of length N, and given the number M  $\leq$  N. Required to find the minimum in each subsegments of length M of the array, ie, find:

$$\min_{\substack{0 \leq i \leq M-1 \\ 1 \leq i \leq M \\ 2 \leq i \leq M+1 \\ \dots \\ NM \leq i \leq N-1}} A[i]$$

Solve this problem in linear time, ie O (N).

It's enough to make all modified for finding the minimum in O (1), which was considered by us above, and in this problem, you can use any of the two methods of implementing such a queue. Next, the solution is clear: to add to the queue first M of array elements, we find in it at least, and remove it, then add the next item in the queue, and izvlechëm from it the first element of the array, again derive the minimum, etc. Since all operations performed on the average queue in constant time, and the asymptotic behavior of the algorithm just get O (N).

It is worth noting that the implementation of the modified queue first method is easier, however, for it will probably need to store the entire array (as in the i-th step need to know the i-th and (iM) th element of the array). When implementing the second stage of the method of array A store obviously do not need - only to find the next, i-th element of the array.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 17 Jul 2009 23:00  
Edited: 17 Jul 2009 23:00

## Randomized heap

Bunch of randomized (randomized heap) - it's a bunch that through the use of a random number generator lets you perform all necessary operations in logarithmic expected time.

Heap is a binary tree for each vertex is true that the value of this top is less than or equal to the value in all of its descendants (it's a bunch of minimum; of course, you can define a bunch of symmetrically to the maximum). Thus, the root is always the minimum heap.

A standard set of operations defined for piles, as follows:

- Adding an item
- Finding the minimum
- Removing the minimum (removing it from the tree and return its value)
- The merger of the two piles (returns a bunch containing elements of both heaps; duplicates are not removed)
- Removal of any element (at a known position in the tree)

Randomized heap lets you perform all the operations for the expected time  $O(\log n)$  with a very simple implementation.

### Contents [hide]

- Randomized heap
  - Data structure
  - Performing operations
  - Asymptotics
    - Expected value
    - Exceeding the expected value
    - Asymptotic behavior of the algorithm

## Data structure

Immediately describe the data structure describing the binary heap:

```
struct tree {
    T value;
    tree * l, * r;
};
```

In the top of the tree contains the value of `value` a certain type `T`, which is defined for comparison operator (`operator <`). Additionally, pointers to the stored left and right sons (which is equal to 0 if there is no appropriate child).

## Performing operations

It is easy to see that all operations are reduced to a heap of a single operation: **merging** two piles into one. Indeed, the addition of an item in a pile equivalent to merge this heap with a bunch consisting of a single element to be added. Finding the minimum does not require any action - just the minimum is the root of the heap. Removing the minimum

equivalent to the heap is replaced by the result of the merger of the left and right subtrees of the root. Finally, the removal of any element is similar to removing a minimum: everything subtree rooted at this vertex is replaced by the merger of two subtrees, the sons of this vertex.

So, we actually need to implement the merge operation only two piles, all other operations are reduced to trivial this operation.

Suppose we are given two piles  $T_1$  and  $T_2$  required to return their union. It is clear that the root of each of these piles minima are blocked, so the resultant heap root will be a minimum of these two values. So, we compare the root of some of the piles is less important, it is placed in the root of the result, and now we must unite the sons of the selected vertices with the rest of the heap. If we are on some grounds choose one of the two sons, then we will have to simply combine the root subtree with this son of a heap. Thus, we again come to the merge operation. Sooner or later, this process stops (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 hard to guess that make this choice, we will **randomly**, so 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;
}
```

There is first checked if at least one of the piles is empty, no action mergers produce is not necessary. Otherwise, we do to heap  $t_1$  was a bunch with a lower value in the root (which exchange the  $t_1$  and  $t_2$ , if necessary). Finally, we believe that the second heap  $t_2$  will merge with the left child of the root pile  $t_1$ , so we randomly exchange the left and right sons, and then merges the left and the second son of the heap.

## Asymptotics

We introduce the random variable  $h(T)$  denoting **the length of the random path** from root to leaf (length in the number of edges). It is understood that the algorithm `merge` is executed for  $O(h(T_1) + h(T_2))$  operations. Therefore, to study the asymptotic behavior of the algorithm is necessary to investigate the random variable  $h(T)$ .

## Expected value

It is argued that the expectation  $h(T)$  is bounded above by the logarithm of the number *n* of vertices in this pile:

$$Eh(T) \leq \log(n + 1)$$

This can be proved easily by induction. Let  $L$  and  $R$ - respectively the left and right subtrees of the root pile  $T$ , and  $n_L$  and  $n_R$ - 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  $c$ .

Denote  $P$  multiple paths from the root to the pile of leaves, the length of which exceeds  $(c + 1) \log n$ . Note that for any path  $p$  length  $|p|$  probability as a random path will be selected because it is  $2^{-|p|}$ . Then we obtain:

$$P\{h(T) > (c + 1) \log n\} = \sum_{p \in P} 2^{-|p|} < \sum_{p \in P} 2^{-(c+1) \log n} = |P|n^{-(c+1)} \leq n^{-c}$$

QED.

## Asymptotic behavior of the algorithm

Thus, the algorithm `merge`, which means that all other expressed through the operation is performed in  $O(\log n)$  an average.

Moreover, for any positive constants  $\epsilon$  there exists a positive constant  $c$  that the likelihood that the operation would require more than  $c \log n$  the operations less  $n^{-\epsilon}$  (in some sense describes the worst behavior of the algorithm).

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:17  
EDIT: 25 May 2012 14:18

## Breadth-first search

Breadth-first search (bypassing the width, breadth-first search) - is one of the basic algorithms on graphs.

As a result of searching the shortest path width is the length in unweighted graph, that is, path that contains the smallest number of edges.

The algorithm works for  $O(n + m)$ , where  $n$ - the number of vertices  $m$ - number of edges.

### Contents [hide]

- Breadth-first search
  - Description of the algorithm
  - Implementation
  - The application of the algorithm
  - Tasks in the online judges

## Description of the algorithm

The input to the algorithm is fed given graph (unweighted), and number of starting vertex  $s$ . A graph can be either oriented or unoriented, this algorithm is not important.

The algorithm itself can be understood as a process of "ignition" of the graph: at the zero step ignite only the tip  $s$ . At each step the fire already burning with each vertex spreads to all its neighbors; ie in one iteration of the algorithm is an expansion of the "ring of fire" in width per unit (hence the name of the algorithm).

More rigorously, this can be represented as follows. Let's create a place  $q$  in which to fit the top of the burning, and zavedëm Boolean array  $used[]$  in which each vertex will celebrate, lit it already or not (in other words, whether she had).

Initially placed in a queue is just the tip  $s$ , and  $used[s] = true$ , as for all other vertices  $used[] = false$ . Then the algorithm is a loop: while the queue is not empty, get out of her head one vertex, view all edges emanating from that vertex, and if some of the peaks is not viewed burn, then set them on fire and put it in the queue.

As a result, when the queue is empty, bypassing the width will bypass all reachable from  $s$  the vertex, with up to each reaches the shortest way. You can also calculate the length of the shortest paths (which just need to have an array of lengths of paths  $d[]$ ), and compact to save information sufficient to recover all of these shortest paths (you have to have an array of "ancestors"  $p[]$ , in which each vertex to store vertex number for which we got into this vertex).

## Implementation

Implement the above algorithm in C ++.

Input data:

```
vector < vector<int> > g; // граф
int n; // число вершин
int s; // стартовая вершина (вершины везде нумеруются с нуля)

// чтение графа
...
```

Himself bypass:

```
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 now we have to restore and display the shortest path to the top of some `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 << " ";
```

## The application of the algorithm

- Search **shortest path** in unweighted graph.
- Search **the connected components** in the graph for  $O(n + m)$ .

To do this, we simply run a preorder traversal of each node, with the exception of vertices visited remaining (`used = true`) after the previous runs. Thus, we perform normal start in width from each vertex, but do not reset each time the array `used[]`, due to which we have every time we get a new connected component, and the total time of the algorithm will continue  $O(n + m)$  (such multiple runs on a graph traversal without zeroing array `used` called a series of rounds in width).

- Finding a solution to any problem (the game) **with the least number of moves**, if each state of the system can be represented by a vertex of the graph, and the 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 is required for the fewest number of moves to move the boxes to the desired position. Solved this preorder traversal through the graph, where the state (top) is a set of coordinates: the coordinates of the robot, and the coordinates of all the boxes.

- Finding the shortest path in the **graph of 0-1** (ie, weighted graph, but with weights equal only 0 or 1): it is enough to slightly modify the breadth-first search: if the current edge of zero weight, and an improvement of the distance to some vertex then add this vertex is not the end but the beginning of the queue.
- Finding **the shortest cycle** in a directed unweighted graph: search the width of each vertex; as soon as we are in the process of trying to crawl out of the current node on some edge to an already visited vertex, then it means that we have found the shortest cycle and stop bypassing wide; found among all such cycles (one from each run bypass) choose the shortest.
- Find all the edges that lie **on any shortest path** between a given pair of vertices  $(a, b)$ . To do this, start the search in two widths: from  $a$ , to and from  $b$ . Denote  $d_a[]$  - the array shortest distances obtained from the first bypass and through  $d_b[]$  - in a second bypass. Now, for any edge  $(u, v)$  is easy to check whether it is on any fast track: the criterion is the condition  $d_a[u] + 1 + d_b[v] = d_a[b]$ .
- Find all the vertices **on any shortest path** between a given pair of vertices  $(a, b)$ . To do this, start the search in two widths: from  $a$ , to and from  $b$ . Denote  $d_a[]$  - the array shortest distances obtained from the first bypass and through  $d_b[]$  - in a second bypass. Now, for each vertex  $v$  is easy to check whether it is on any fast track: the criterion is the condition  $d_a[v] + d_b[v] = d_a[b]$ .
- Find **the shortest way to an even** in the graph (ie, the path of even length). For this we need to build an auxiliary graph, whose vertices are the state  $(v, c)$ , where  $v$  - the number of the current node,  $c = 0 \dots 1$  - the current parity. Each edge  $(a, b)$  of the original graph in this new column will turn into two ribs  $((u, 0), (v, 1))$  and  $((u, 1), (v, 0))$ . After that, it is necessary to bypass the

column width to find the shortest path from the starting vertex to the end, with parity equal to 0.

## Tasks in the online judges

List of tasks that can be taken using a wide detour:

- SGU # 213 "Strong Defence" [Difficulty: Medium]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:19  
EDIT: 10 Jun 2008 19:21

## Dfs

This is one of the basic algorithms on graphs.

As a result, depth-first search is lexicographically first path in the graph.

The algorithm works for  $O(N + M)$ .

### Contents [hide]

- Dfs
  - Applying the algorithm
  - Implementation

## Applying the algorithm

- Search any path in the graph.
- Search lexicographically first path in the graph.
- Checking whether a tree node ancestor another:

At the beginning and end of the iteration depth-first search will remember the "time" coming and going at each vertex. Now in  $O(1)$  can find the answer: vertex  $i$  is an ancestor node  $j$  if and only if the start $_i < \text{start}_j$  and the end $_i > \text{end}_j$ .

- The task of LCA (lowest common ancestor).
- Topological sorting :

Run a series of depth-first search to traverse all vertices. Let's sort out the top of the time descending - this will be the answer.

- Checking on the graph is acyclic and finding cycle
- Search strongly connected components :

First, do a topological sort, count and then transpose again hold a series of depth-first search in a manner determined by the topological sorting. Each tree search - strongly connected component.

- Search bridges :
- first converted into a directed graph, making a series of depth-first search, and orienting each edge as we tried to pass him. Then we find strong-components. Bridges are those edges whose endpoints belong to different strongly connected components.

## Implementation

```
vector<vector<int>> g; // Count
int n; // The number of vertices

vector<int> color; // Vertex color (0, 1, or 2)

vector<int> time_in, time_out; // "Times" coming and going from the top
int dfs_timer = 0; // "Timer" to determine the times

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 are not important, as well as the vertex colors are not important (but then you have to enter the same in the sense of a Boolean array used). Here are the most simple implementation:

```
vector <vector <int>> g; // Count
int n; // The 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);
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:24  
EDIT: 23 Aug 2011 12:42

## Topological sorting

Dan directed graph with  $n$  vertices and  $m$  edges. Need to renumber the vertices so that each edge led from the top with a smaller number in the top with a large.

In other words, find a permutation of vertices ( **topological order** ) corresponds to the order given by all edges of the graph.

Topological sort can be **not only** (for example, if the graph - blank, or if there are three such vertices  $a, b, c$  that because  $a$  there is a way in  $b$  and in  $c$ , but none of the  $b$  in  $c$  or out  $c$  to  $b$  get you can not).

Topological sort can **not exist** at all - if the graph contains cycles (as there is a contradiction: there is a path from one vertex to another, and vice versa).

**A common problem** on the topological sort - the next. There are  $n$  variables that are unknown to us. We only know about some pairs of variables that one variable is smaller than the other. Need to check whether these are not contradictory inequality, and if not, to give the variables in ascending order (if there are several solutions - any issue). Easy to see that this is exactly is the problem of finding a topological sorting of the graph  $n$  vertices.

### Contents [hide]

- Topological sorting
  - Algorithm
  - Implementation
  - Tasks in the online judges

## Algorithm

Solutions for use [in bypass depth](#).

Suppose that the graph is acyclic, ie, solution exists. What makes a detour into the depths? When you run out of some tops  $v$  it tries to run along all edges emanating from  $v$ . Along the edges of the ends of which have already been visited before, it does not pass, and along all others - calls itself and passes on their ends.

Thus, by the time of the call,  $\text{dfs}(v)$  all the vertices reachable from  $v$  either directly (one edge) and indirectly (by the way) - all such vertices already visited bypass. Therefore, if we are at the moment out of the  $\text{dfs}(v)$  top of our to add to the top of a list, in the end of this list will **topological sorting**.

These explanations can be presented in a slightly different way, using the concept of "**time out**" in the crawl depth. Time out for each vertex  $v$  - a point in

time at which the call ended up working  $\text{dfs}(v)$  in the crawl depth of it (retention times can be numbered from 1 to  $n$ ). It is easy to understand that in going into the depth of time to any vertex  $v$  is always greater than the time out of all vertices reachable from it (as they have been visited or to call  $\text{dfs}(v)$ , or during it). Thus, the desired topological sorting - sorting in descending order of the time of release.

## Implementation

We present the implementation, it is assumed that the graph is acyclic, ie, Seeking topological sorting exists. If necessary check on the graph is acyclic easily inserted into the bypass in depth, as described in the [article of circumvention in depth](#).

```

int n; // число вершин
vector<int> g[MAXN]; // граф
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 constants **MAXN** should be set equal to the maximum possible number of vertices in the graph.

The main function of the solution - it `topological_sort`, it initializes tagging bypass in depth, runs it, and end up with a response vector `ans`.

## Tasks in the online judges

A list of tasks that need to search for topological sorting:

- [UVA # 10305 "Ordering Tasks"](#) [Difficulty: Low]
- [UVA # 124 "Following Orders"](#) [Difficulty: Low]
- [UVA # 200 "Rare Order"](#) [Difficulty: Low]

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 10 Jun 2008 19:25  
 EDIT: 24 Aug 2011 12:31

## The search algorithm of connected components in a graph

### Contents [\[hide\]](#)

- The search algorithm of connected components in a graph
  - Algorithm for solving
  - Implementation

Given an undirected graph  $G$  with  $n$  vertices and  $m$  edges. You want to find in it all the connected components, ie vertices divided into several groups so that within a group can be reached from any one node to another, and between different groups, - the path does not exist.

## Algorithm for solving

Alternatively you can use as a [bypass in depth](#) and [in breadth traversal](#).

In fact, we will produce **a series of rounds** : first round of the launch of the first vertex and all vertices that in doing so he walked - form the first connected component. Then we 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  $O(n + m)$ : in fact, this algorithm will not run on the same vertex twice, which means that each edge will be seen exactly twice (at one end and the other end).

## Implementation

To implement a little more convenient to [bypass in 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 (int i = 0; i < g[v].size(); i++)
        if (!used[g[v][i]])
            dfs (g[v][i]);
```

```

        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 for the call - `find_comps()` she finds and displays the connected components of a graph.

We believe that the graph of a given adjacency list, that  $g[i]$  contains a list of vertices, in which there are edges from the vertex  $i$ . Constant `MAXN` should be set equal to the maximum possible number of vertices in the graph.

The vector `comp` contains a list of vertices in the current connected component.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:27  
EDIT: 9 Jul 2009 18:24

## Search strongly connected component, the construction of the condensation graph

### Contents [hide]

- Search strongly connected component, the construction of the condensation graph
  - Definitions, formulation of the problem
  - Algorithm
  - Implementation
  - Literature

## Definitions, formulation of the problem

Dan directed graph  $G$  whose vertex set  $V$  and the set of edges -  $E$ . Loops and multiple edges are allowed. Denoted by  $n$  the number of vertices, through  $m$ - the number of edges.

**Strongly connected component** (strongly connected component) is called a (maximal with respect to inclusion) subset of vertices  $C$  that any two vertices of this subset are reachable from each other, ie, to  $\forall u, v \in C$ :

$$u \mapsto v, v \mapsto u$$

where the symbol  $\mapsto$  hereafter we denote reachability, ie, existence of a path from the first vertex to the second.

It is understood that the components are strongly connected to a given graph does not intersect, ie in fact it is a partition of all vertices of the graph. Hence the logical definition of **condensation**  $G^{SCC}$  as the graph obtained from the graph of the compression of each component of a strong connection in one vertex. Each vertex of the graph corresponds to the condensation component strongly connected graph  $G$  and a directed edge between two vertices  $C_i$  and  $C_j$  graph condensation is carried out, if there exists a pair of vertices  $u \in C_i, v \in C_j$ , between which there was an edge in the original graph, ie  $(u, v) \in E$ .

The most important property of the graph condensation is that it is **acyclic**. Indeed, suppose that  $C \mapsto C'$ , prove that  $C' \not\mapsto C$ . From the definition of condensation obtain that there exist two vertices  $u \in C$  and  $v \in C'$  that  $u \mapsto v$ . Will prove by contradiction, ie, suppose that  $C' \mapsto C$ , then there are two vertices  $u' \in C$  and  $v' \in C'$  that  $v' \mapsto u'$ . But since  $u$  and  $u'$  are in the same strongly connected component, then there is a path between them; similarly for  $v$  and  $v'$ . As a result, the path integrating, we obtain  $v \mapsto u$ , simultaneously  $u \mapsto v$ .

Consequently,  $u$  and  $v$  must belong to strongly connected component, ie, a contradiction, as required.

Algorithm described below are shown in this graph all components of strong connectivity. Count on them to build a condensation is not difficult.

## Algorithm

Algorithm described here has been proposed independently Kosaraju (Kosaraju) and Sharir (Sharir) in 1979. This is a very easy-to-implement algorithm based on two series of [depth-first search](#), and because working at the time  $O(n + m)$ .

**In the first step** of the algorithm, a series of detours in depth attending the entire graph. To do this, we go through all the vertices of the graph and from each vertex is not being visited call bypass in depth. In this case, for each vertex  $v$  remember **time to tout**[ $v$ ]. These retention times play a key role in the algorithm, and this role is expressed in the theorem below.

First, we introduce the notation: time- tout[ $C$ ]out components  $C$ of strong connectivity is defined as the maximum of the values tout[ $v$ ]for all  $v \in C$ . In addition, the proof of the theorem will be referred to and the time of entry in each vertex tin[ $v$ ], and similarly determine the time of entry tin[ $C$ ]for each strongly connected components of a minimum of values tin[ $v$ ]for all  $v \in C$ .

**Theorem** . Let  $C$ and  $C'$ - two different strongly connected components, and let the condensation in the graph is an edge between them ( $C, C'$ ). Then tout[ $C$ ] > tout[ $C'$ ].

In the proof there are two fundamentally different cases, depending on which of the components of the first round will go in depth, ie depending on the relationship between tin[ $C$ ]and tin[ $C'$ ]:

- The first component was reached  $C$ . This means that at some point in time in the round comes to a depth of a vertex  $v$ components  $C$ , while all the other vertices component  $C$ and  $C'$ not yet visited. However, since by condition in box condensations is an edge ( $C, C'$ ), then from the top  $v$ will be achieved not only all components  $C$ but the entire component  $C'$ . This means that when you start from the top of  $v$ bypassing deep pass over all vertices of the component  $C$ and  $C'$ , and, therefore, they will be in relation to the descendants  $u$ in the tree traversal in depth, ie for each vertex  $u \in C \cup C'$ ,  $u \neq v$ will be performed tout[ $v$ ] > tout[ $u$ ], QED
- The first component was reached  $C'$ . Again, at some point in time in the round comes to a depth of a vertex  $v \in C'$ , with all the other vertices component  $C$ and  $C'$ not visited. As for the condition in box condensations edge there ( $C, C'$ ), then, as a result of condensation acyclic graph, there is no way back  $C' \not\rightarrow C$ , that is, bypass in depth from the top  $v$ you reach the top  $C$ . This means that they will be visited bypass in depth later, which implies tout[ $C$ ] > tout[ $C'$ ], QED

The above theorem is **the basis for the algorithm** search of strongly connected

component. From it follows that every edge  $(C, C')$  in the graph condensation comes from components with greater magnitude  $\text{tout}$  in the component with a lower value.

If we sort all the vertices  $v \in V$  in order of exit time  $\text{tout}[v]$ , the first would be some vertex  $u$  belonging to the "root" strongly connected component, ie, which is not included none of the edges in the graph condensation. Now we would like to start a tour of this peak  $u$ , which would be visited only this component strongly connected and did not go to any other; learning how to do it, we can gradually highlight all strongly connected components: removing from the graph vertex of the first selected component, again, we find among the remaining top with the highest value  $\text{tout}$ , re-run of it this round, etc.

To learn how to do this tour, we consider the **transposed graph**  $G^T$ , ie, the graph obtained from  $G$  each change in direction of the opposite edge. It is not difficult to understand that in this graph are the same strongly connected component, as in the original graph. Moreover, the graph condensation  $(G^T)^{\text{SCC}}$  for him to be transposed graph is condensation of the original graph  $G^{\text{SCC}}$ . This means that now because we are considering the "root" component will no longer go to the edges of the other components.

So, to get around the whole "root" strongly connected component containing a vertex  $v$ , simply run the crawl from the top of  $v$  the graph  $G^T$ . This tour will visit all the vertices of the components of strong connectivity and only them. As already mentioned, then we can mentally remove these vertices in the graph, find another vertex with the maximum value  $\text{tout}[v]$  and run round the transposed column out of it, etc.

Thus we have constructed the following **algorithm** selection component of strong connectivity:

1 step. Launch a series of rounds into the depths of the graph  $G$ , which returns the top in order to increase output of time  $\text{tout}$ , ie, some list  $\text{order}$ .

Step 2. Build transposed graph  $G^T$ . Launch a series of detours in depth / width of the graph in the manner determined by the list  $\text{order}$  (namely, in reverse order, ie, in order to reduce the time out). Each set of nodes reached as a result of the next start crawling, and there will be another component of the strong connectivity.

**Asymptotic behavior** of the algorithm, obviously, is  $O(n + m)$  because it is only two bypass in depth / width.

Finally, it is appropriate to note the relationship with the notion of **topological sorting**. First, step 1 of the algorithm is not nothing but a topological sort of the graph  $G$  (in fact, this is what means sorting the vertices time out). Secondly, the scheme of the algorithm is that the strongly connected components and generates it in order to reduce their output times, thus it produces components - in the condensation of a vertex of 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;
    ... чтение n ...
    for (;;) {
        int a, b;
        ... чтение очередного ребра (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);
            ... вывод очередной component ...
            component.clear();
        }
    }
}

```

Here, in `g` the graph itself is stored, and `gr`- transpose graph. Function `dfs1` crawls in depth on the graph  $G$ , the function `dfs2`- the transposed  $G^T$ . Function `dfs1` populates a list of `order` vertices in order of increasing time out (in fact, makes a topological sort). The function `dfs2` stores all reach the top of the list `component`

, which after each run will contain another component of the strong connectivity.

## Literature

- Thomas H. Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Introduction to Algorithms** [2005]
- M. Sharir. **A strong-Connectivity algorithm and ITS applications in Data-Flow analysis** [1979]

# MAXimal

[home](#)

[algorithms](#)

[bookz](#)

[forum](#)

[about](#)

Added: 10 Jun 2008 19:28  
EDIT: 23 Aug 2011 11:23

## Search bridges

Suppose we are given an undirected graph. A bridge is an edge whose removal makes the graph disconnected (or, more precisely, increases the number of connected components). You want to find all the bridges in a given graph.

Informally, this problem is formulated as follows: you want to find on a given road map all the "important" roads, ie road such that removal of any of them will lead to the disappearance of the path between any pair of cities.

Below we describe an algorithm based on [DFS](#) and work during  $O(n + m)$  which  $n$ - the number of vertices  $m$ - edges in the graph.

Note that the site also describes [an online search algorithm bridges](#) - in contrast to the algorithm described here, an online algorithm is able to support all the bridges in the changing graph graph (meaning adding new edges).

### Contents [hide]

- Search bridges
  - Algorithm
  - Implementation
  - Tasks in the online judges

## Algorithm

Start the [tour in depth](#) from an arbitrary vertex of the graph; denote it through [root](#). We note the following **fact** (which is not hard to prove):

- Suppose we are in the bypass in depth, looking now all edges from the vertex  $v$ . Then, if the current edge  $(v, to)$  such that from the top  $to$  and from any of its descendants in the tree traversal is no inverse depth at the top edge  $v$  or any of its parent, then this edge is bridge. Otherwise, it is not a bridge. (In fact, this condition, we check if there is no other way out  $v$  in  $to$ , but to descend along the edge  $(v, to)$  of the tree traversal in depth.)

Now it is necessary to learn how to verify this fact for each vertex effectively. To do this, use the "time of entry into the top," is calculated [depth-first search algorithm](#).

So, let  $tin[v]$ - this time of call DFS at the top  $v$ . Now, we introduce an array  $fup[v]$ , which will allow us to answer the above questions. Time  $fup[v]$  is the minimum time of entering into the very top of  $tin[v]$ , since the approach to each vertex  $p$  is the end of a back edge  $(v, p)$ , as well as of all the values  $fup[to]$  for each vertex  $to$ , which is a direct son of  $v$  a 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 to its ancestor if and only if there is such a son  $to$  that  $fup[to] \leq tin[v]$ . (If  $fup[to] = tin[v]$  it means that there exists an opposite edge coming exactly  $v$ , but if  $fup[to] < tin[v]$  it means the presence of reverse edges in any ancestor vertices  $v$ ).

Thus, if the current edges  $(v,to)$  (belonging to the search tree) is satisfied  $fup[to] > tin[v]$ , then this edge is a bridge; otherwise it is not a bridge.

## Implementation

If we talk about the actual implementation, here we need to be able to distinguish three cases: when we are on the edge of the tree depth-first search, when we go to the opposite edge, and when we try to take the edge of the tree in the opposite direction. It is, accordingly, the cases:

- $used[to] = false$  - Criterion ribs search tree;
- $used[to] = true \&& to \neq parent$  - Criterion Back Ribs;
- $to = parent$  - Criterion pass along the edge of the search tree in the opposite direction.

Thus, for the implementation of these criteria, we need to pass in the search function in the top of the depth of the parent of the current node.

```

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) {
        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])
                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 `find_bridges`- it produces the necessary initialization and start crawling in depth for each connected component of the graph.

At the same time `IS_BRIDGE(a,b)`- this is a function that will respond to the fact that the edge  $(a, b)$ is a bridge, for example, to display it on the screen edge.

Constant `MAXN`in the beginning of the code should be set equal to the maximum possible number of vertices in the input graph.

It is worth noting that this implementation does not work correctly in the presence of a graph **multiple edges** : it actually does not pay attention, if multiple edge or it is unique. Of course, multiple ribs should not be included in the response, so when you call `IS_BRIDGE`, you can check further, if not a multiple edge we want to add to the answer. Another way - a more accurate work with the ancestors, ie transmit to `dfs`the top is not the parent, and the number of edges on which we went to the top (you will need to additionally store the numbers of all edges).

## Tasks in the online judges

A list of tasks that need to search for bridges:

- [UVA # 796 "Critical Links"](#) [Difficulty: Low]
- [UVA # 610 "Street Directions"](#) [Difficulty: Medium]

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: 5 Jul 2008 22:26  
EDIT: 6 Dec 2012 1:41

## Find points of articulation

Suppose we are given a connected undirected graph. **articulation point** (or points of articulation, Eng. "cut vertex" or "articulation point") is called a vertex whose removal makes the graph disconnected.

We describe an algorithm based on DFS working for  $O(n + m)$ , where  $n$ - the number of vertices  $m$ - edges.

### Contents [hide]

- Find points of articulation
  - Algorithm
  - Implementation
  - Tasks in the online judges

## Algorithm

Start the tour in depth from an arbitrary vertex of the graph; denote it through **root**. We note the following **fact** (which is not hard to prove):

- Suppose we are in the bypass in depth, looking now all edges from the vertex  $v \neq \text{root}$ . Then, if the current edge  $(v, to)$  is such that from the top  $to$  and from any of its descendants in the tree traversal depth no reverse edges in any ancestor vertex  $v$ , the vertex  $v$  is an articulation point. Otherwise, i.e. if bypassing deep through all the edges from the top  $v$ , and did not find satisfying the above conditions ribs, then the vertex  $v$  is not a point of articulation. (In fact, this condition, we check if there is no other way of  $v$  to  $to$ )
- Let us 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 depth. (In fact, this means that, after passing out **root** on any edge, we were not able to bypass the entire graph, which immediately implies that **root**- the point of articulation).

(Wed wording of this criterion with the wording of the criteria for the search **algorithm bridges**.)

Now it is necessary to learn how to verify this fact for each vertex effectively. To do this, use the "time of entry into the top," is calculated **depth-first search algorithm**.

So, let  $tin[v]$ - this time of call DFS at the top  $v$ . Now, we introduce an array  $fup[v]$ , which will allow us to answer the above questions. Time  $fup[v]$  is the minimum time of entering into the very top of  $tin[v]$ , since the approach to each vertex  $p$  is the end of a back edge  $(v, p)$ , as well as of all the values  $fup[to]$  for

each vertex  $to$ , which is a direct son of  $v$  a 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 to its ancestor if and only if there is such a son  $to$  that  $fup[to] < tin[v]$ .

Thus, if the current edge  $(v,to)$  (the search tree belongs) is satisfied

$fup[to] \geq tin[v]$ , then the vertex  $v$  is a point of articulation. For the initial vertex of  $v = \text{root}$  another criterion: for this it is necessary to count the number of vertices immediate sons in the tree traversal depth.

## Implementation

If we talk about the actual implementation, here we need to be able to distinguish three cases: when we are on the edge of the tree depth-first search, when we go to the opposite edge, and when we try to take the edge of the tree in the opposite direction. It is, accordingly, cases  $used[to] = false$ ,  $used[to] = true \&& to \neq parent$  and  $to = parent$ . Thus, we need to pass in the search function in the top of the depth of the parent of the current node.

```

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;
    ... чтение n и 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( $v$ )** in the code - is a function that will respond to the fact that the vertex  $v$  is an articulation point, for example, to display the top of this screen (it should be borne in mind that for the same vertex, this function can be called several times).

## Tasks in the online judges

A list of tasks that need to search for points of articulation:

- UVA # 10199 "**Tourist Guide**" [Difficulty: Low]
- UVA # 315 "**Network**" [Difficulty: Low]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 5 Aug 2011 1:55  
EDIT: 5 Aug 2011 1:55

## Search bridges online

Suppose we are given an undirected graph. A bridge is an edge whose removal makes the graph disconnected (or, more precisely, increases the number of connected components). You want to find all the bridges in a given graph.

Informally, this problem is formulated as follows: you want to find on a given road map all the "important" roads, ie road such that removal of any of them will lead to the disappearance of the path between any pair of cities.

Described herein is an algorithm **online**, that means the input graph is not known in advance, and the edges are added to it one by one, and after adding each such algorithm recalculates all bridges in the current column. In other words, the algorithm is designed to work effectively in a dynamic, changing graph.

More strictly, **formulation of the problem** is as follows. Initially empty graph consists of  $n$  vertices. Then receives requests, each of which - a pair of vertices  $(a, b)$  that represent an edge in the graph to be added. Required after each request, ie, 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, and clearly support the components edged doubly linked.)

Described below algorithm works in time  $O(n \log n + m)$ , where  $m$ - the number of requests. The algorithm is based on the [data structure "a system of disjoint sets"](#).

The reduced implementation of the algorithm, however, is working for the time  $O(n \log n + m \log n)$ , as it uses in one place a simplified version [of disjoint sets](#) without ranking heuristics.

### Contents [hide]

- Search bridges online
  - Algorithm
    - Data structure for storing forest
  - Implementation

## Algorithm

It is known that the edges of bridges divide the vertices into components called costal doubly connected components. If each component edged doubly linked to squeeze in one vertex, and leave only the edges of bridges between these components, we obtain an acyclic graph, that is, forest.

Algorithm described below support explicitly the **forest component edged doubly linked**.

It is clear that initially, when the graph is empty, it contains a  $n$  component edged

doubly linked, not bound to each other.

When you add another rib  $(a, b)$  may be three situations:

- Both ends  $a$  and  $b$  are in the same component edged doubly connected - then this edge is not a bridge, and does not change the structure of the forest, so just skip this edge.

Thus, in this case the number of bridges is not changed.

- Tops  $a$  and  $b$  are in different connected components, ie connect the two trees. In this case, the edge  $(a, b)$  becomes the new bridge and the two are combined into a single tree (and all old bridges remain).

Thus, in this case the number of bridges is incremented.

- Tops  $a$  and  $b$  are in the same connected component, but in different components of the costal doubly linked. In this case, this edge defines a ring together with some of the old bridges. All these bridges are no longer bridges, and the resulting cycle must be combined into a new component edged doubly linked.

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 storing forest

All we need from the data structures - a [system of disjoint sets](#). In fact, we need to make two copies of this structure: one is to maintain the **connected components**, the other - to maintain the **doubly-connected component of the rib**.

In addition, the storage structure of the trees in the forest component is doubly connected to each node will store a pointer  $\text{par}[]$  to its parent in the tree.

We now consistently disassemble every operation that we must learn to realize:

- Check whether the two are listed top in the same connected component / doubly linked.** Is the usual request to the structure of the "system of disjoint sets."
- Connecting two trees into one** by an edge  $(a, b)$ . Because it could turn out that no vertex  $a$  or a vertex  $b$  are not roots of their trees, the only way to combine these two trees - **perepodvesit** one of them. For example, you can perepodvesit one tree for the top  $a$ , and then attach it to another tree, making the top of  $a$  the child to  $b$ .

However, there is a question about the effectiveness of the operation perepodveshivaniya: to perepodvesit tree rooted at  $r$  the vertex  $v$ , you have to pass on the way from  $v$  to  $r$  redirecting pointers  $\text{par}[]$  in the opposite direction, as well as changing references to an ancestor in the system of disjoint sets in charge of the connected components.

Thus, the cost of the operation has perepodveshivaniya  $O(h)$  where  $h$  - the height of the tree. It can be estimated even higher, saying that this is the value

$O(\text{size})$  where  $\text{size}$  - the number of vertices in the tree.

We now apply this standard trick: we say that the two trees **will perepodveshivat one in which fewer vertices**. Then intuitively clear that the worst case - when combined two trees of approximately equal size, but then the result is a tree twice the size that does not allow such a situation to occur many times. Formally, this can be written in the form of a recurrence relation:

$$T(n) = \max_{k=1 \dots n-1} \{ T(k) + T(n-k) + O(n) \},$$

where by  $T(n)$  we have denoted the number of operations required to obtain the tree of  $n$  vertices using the operations of union and perepodveshivaniya trees. This is a known recurrence relation, and it has a solution  $T(n) = O(n \log n)$ .

Thus, the total time spent on all perepodveshivaniya, will  $O(n \log n)$ , if we always perepodveshivat lesser of two tree.

We have to keep the size of each connected component, but the data structure "a system of disjoint sets" lets you do this easily.

- **Search loop** formed by adding a new edge  $(a, b)$  to some wood. Practically, this means that we need to find the lowest common ancestor (LCA) of vertices  $a$  and  $b$ .

Note that then we sozhmëm all the vertices of the detected cycle in one vertex, so we want any search algorithm LCA, working during the order of its length.

Since all the information about the structure of the tree, that we have - it links  $\text{par}[]$  to ancestors, the only possibility seems the next search algorithm LCA: mark peaks  $a$  and  $b$  as the visited, then go to their ancestors  $\text{par}[a]$  and  $\text{par}[b]$  and mark them, then to their ancestors, and so on, until it happens that at least one of the two current peaks is already marked. This would mean that the current peak - is the required LCA, and it will be necessary to repeat again the path to it from the top  $a$  and from the top  $b$  - thus we find the desired cycle.

It is obvious that 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  $(a, b)$  to some wood.

We need to create a new component edged doubly linked, which will consist of all the vertices of the detected cycle (of course, he could detect cycles consist of some components of the doubly linked, but it does not change anything). Furthermore, it is necessary to compress so that the tree structure has not been disrupted, and all pointers  $\text{par}[]$  and two sets of disjoint were correct.

The easiest way to do this - **to compress all the peaks found in the cycle of LCA**. In fact, the top-LCA - is the highest peaks of the compressible, ie it  $\text{par}$  remains unchanged. For all other vertices compressible update also did not need to, since these vertices simply cease to exist - in a system of disjoint sets for the components of the doubly linked all these vertices are simply points to the top-LCA.

But then it turns out that a system of disjoint sets for doubly connected component works without heuristics union by rank if we always attach to the top of the cycle of LCA, then this heuristic is no place. In this case, the asymptotic behavior occurs  $O(\log n)$  because without heuristics to rank any operation with a system of disjoint sets of works that for a time.

To achieve the asymptotic behavior of  $O(1)$  a single request is necessary to combine the top of the cycle according to the ranking heuristics, and then assign para new leader par[LCA].

## Implementation

We give here the final realization of the whole algorithm.

In order to ease the system of disjoint sets for doubly connected component written **without rank heuristics**, so the final asymptotic behavior make  $O(\log n)$  the request on average. (For information on how to reach the asymptotic behavior  $O(1)$ , described above in paragraph "compression cycle.")

Also in this implementation is not kept themselves ribs, bridges, and kept only their number - see. Variable **bridges**. However, if you want to not be difficult to have set all of the bridges.

Initially, you should call the function **init()** that initializes the two systems of disjoint sets (releasing each vertex in a separate set and dimensioned equal to unity), marks the ancestors **par**.

The main function - is **add\_edge(a, b)** that processes the request to add a new edge.

Constant **MAXN** should be set equal to the maximum possible number of vertices in the input graph.

More detailed explanations to this sale. Refer 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);
}

```

We comment on the code in more detail.

**The system of disjoint sets for doubly connected component** is stored in an array `bl[]`, and a function that returns a leader doubly connected components - a `get(v)`. This function is used many times in the rest of the code, as it must be remembered that after the compression of several peaks in a single all these vertices cease to exist, and instead there is only their leader, which are stored and correct data (ancestor `par`, ancestor of the system of disjoint sets for connected components, etc.).

**System for disjoint sets of connected components** is stored in the array `comp[]`, there is also an additional array `size[]` to store the size of the component. The

function `get_comp(v)` returns the leader of the connected components (which is actually the root of the tree).

**Function perepodveshivaniya tree make\_root(*v*)** works as described above: it goes from the top *v* to the ancestors to the root, each time redirecting ancestor `par` in the opposite direction (down toward the top *v*). Also updated pointer `comp` in the system for disjoint sets of connected components to point to the new root. After perepodveshivaniya the new root Dimensions `size` connected components. Note that the implementation every time we call the function `get()` to access it to the leader of the components of strong connectivity, and not to some vertex, which may have been compressed.

**The detection and path compression** `merge_path(a, b)`, as described above, looking for peaks LCA *a* and *b*, which rises up from them in parallel until a vertex is encountered a second time. To be effective, passed peaks marked by the technique of "numerical used", that works for the  $O(1)$  application instead `set`. TRIP is stored in the vectors `va` and `vb` then to walk on it a second time to LCA, thereby obtaining all the vertices of the cycle. All vertices of the cycle is compressed, by attaching them to the LCA (here comes the asymptotic behavior  $O(\log n)$ ), as in compression we do not use the rank heuristic). Along the way, is considered to be the number of edges traversed, which is the number of bridges to detect cycles (this amount is subtracted from `bridges`).

Finally, the **query function** `add_edge(a, b)` defines the connected components, which are the vertices *a* and *b*, and if they lie in different connected components, the smaller tree perepodveshivaetsya for the new root, and then attached to a large tree. Otherwise, if the vertices *a* and *b* belong to the same tree, but in different components of the doubly connected, the function is called `merge_path(a, b)`, which detects a cycle, and compresses it into a doubly-linked component.

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:30  
EDIT: 10 Dec 2012 16:05

## Finding the shortest paths from a given vertex to all other vertices Dijkstra's algorithm

### Contents [hide]

- Finding the shortest paths from a given vertex to all other vertices
  - Dijkstra's algorithm
    - Statement of the Problem
    - Algorithm
    - Proof
    - Implementation
    - Literature

### Statement of the Problem

Given directed or undirected weighted graph with  $n$  vertices and  $m$  edges. The weights of all edges are non-negative. Contains some starting vertex  $s$ . Required to find the length of the shortest paths from the 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 that offered Dutch researcher **Dijkstra** (Dijkstra) in 1959

Zaveděm array  $d[]$  in which each vertex  $v$  will store the current length  $d[v]$  of the shortest path  $s$  in  $v$ . Initially  $d[s] = 0$ , and for all the other vertices, this length is equal to infinity (when implemented on a computer usually as infinity simply choose a sufficiently large number, certainly more than possible path length):

$$d[v] = \infty, v \neq s$$

In addition, for each vertex  $v$  we store, it is still labeled or not, i.e. zaveděm boolean array  $u[]$ . Initially, all vertices are labeled, ie

$$u[v] = \text{false}$$

Dijkstra's algorithm itself consists of **iterations**. At the next iteration of the selected vertex with the lowest value among the not yet marked, ie  $\min_{v \in V} d[v]$

$$d[v] = \min_{p: u[p]=\text{false}} d[p]$$

(It is understood that on the first iteration will be selected starting vertex  $s$ .)

Selected so the top  $v$  notes marked. Further, in the current iteration, from the top  $v$  are made of relaxation : Browse all edges  $(v, to)$  emanating from a vertex  $v$ , and for each such vertex  $to$  algorithm tries to improve value  $d[to]$ . Let the length of this edge is  $\text{len}$  then in the form of relaxation code looks like:

$$d[to] = \min(d[to], d[v] + \text{len})$$

At the ends the current iteration, the algorithm proceeds to the next iteration (again selects the lowest peak value  $d$ , the relaxation produced from it, etc.). In the end, after  $n$  iterations, all the vertices of the graph will be marked, and the algorithm completes its work. It is alleged that the obtained values  $d[v]$  are the desired length of the shortest paths from  $s$  to  $v$ .

It is worth noting that, if not all vertices reachable from a vertex  $s$ , then the values  $d[v]$  for them will remain endless. It is clear that the last few iterations of the algorithm will just select these vertices, but no useful work to produce these iterations will not (because an infinite distance can not prorelaksirovat others, too, even infinite distance). Therefore, the algorithm can be immediately stopped as soon as the selected vertex is taken from the top of an infinite distance.

**Recovery paths** . Of course, you usually need to know not only the length of the shortest path, but get yourself way. We show how to maintain sufficient information to restore the shortest path from  $s$  any vertex to. It's enough to so-called **array ancestors** : an array  $p[]$  in which each vertex  $v \neq s$  is stored vertex number  $p[v]$ , 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 the last vertex of this path, you get way, ending a vertex  $p[v]$ , and this will be the shortest path for the top  $p[v]$ . So, if we have this array of ancestors, the shortest path can be restored to him, each time just taking ancestor of the current node until we arrive at the starting vertex  $s$  so we get the desired shortest path, but spelled backwards. Thus, the shortest way  $P$  to the top  $v$  is:

$$P = (s, \dots, p[p[p[v]]], p[p[v]], p[v], v)$$

It remains to understand how to build the array ancestors. However, this is very simple: at every successful relaxation, ie when the selected vertex  $v$  is improving distance to a vertex  $to$ , we record that ancestor vertex  $to$  is a vertex  $v$ :

$$p[to] = v$$

## Proof

**The main assertion** , based on which the correctness of Dijkstra's algorithm is as follows. It is alleged that after some vertex  $v$  becomes marked, the current distance to it  $d[v]$  is already the shortest, and, consequently, more will not change.

**The proof** will be producing by induction. For the first iteration of the justice of his obvious - for the top  $s$ , we have  $d[s] = 0$ , which is the length of the shortest path to it. Now suppose that it holds for all previous iterations, ie, all already labeled vertices; prove that it is not broken after the current iteration. Let  $v$ - top chosen for the current

iteration, ie vertices that are going to mark the algorithm. We will prove that  $d[v]$  indeed is the length of the shortest path to it (we denote this length through  $l[v]$ ).

Consider the shortest way  $P$  to the top  $v$ . Clearly, this path can be divided into two paths:  $P_1$  consisting only of marked vertices (at least the starting vertex  $s$  is in the way), and the rest of the way  $P_2$  (it can also include a marked peak, but begins always with untagged). Denoted by  $p$  the first vertex of the path  $P_2$ , and through  $q$  - the last point of the path  $P_1$ .

We first prove the assertion for the top  $p$ , ie, prove equality  $d[p] = l[p]$ . However, it is almost obvious: after all, one of the previous iterations we chose the top  $q$  and perform the relaxation of it. Since (by virtue of the choice of the vertex  $p$ ) to the shortest path  $p$  is the shortest path to the  $q$  plus edge  $(p, q)$ , then if the relaxation of the  $q$  value  $d[p]$  really set the desired value.

Due to the non-negativity values ribs length of the shortest path  $l[p]$  (and she just proved equal  $d[p]$ ) does not exceed the length of  $l[v]$  the shortest path to the top  $v$ . Given that  $l[v] \leq d[v]$  (after all, Dijkstra's algorithm could not find a shorter path than it is at all possible), as a result we obtain the relations:

$$d[p] = l[p] \leq l[v] \leq d[v]$$

On the other hand, and as  $p$  and  $v$  - unmarked vertices, so that both the current iteration vertex been chosen  $v$  instead of a vertex  $p$ , then obtain another inequality:

$$d[p] \geq d[v]$$

From these two inequalities we conclude equality  $d[p] = d[v]$ , and then found out before we receive and relations:

$$d[v] = l[v]$$

QED.

## Implementation

Thus, the Dijkstra's algorithm is  $n$  iterative, each of which is selected unlabeled vertex with the lowest value  $d[v]$ , this vertex is marked, and then played all edges emanating from that vertex, and along each edge is an attempt to improve the value  $d[]$  at the other end of the edge.

Time of the algorithm consists of:

- $n$  Just search the top with the lowest value  $d[v]$  among all unlabelled vertices, ie, among  $O(n)$  vertices
- $m$  relaxation time an attempt is made

At the simplest implementation of these operations on the top of the search will be spent  $O(n)$  operations and one relaxation -  $O(1)$  operations, and the resulting **asymptotic behavior of the algorithm** is as follows:

$$O(n^2 + m)$$

## Implementation :

```

const int INF = 1000000000;

int main() {
    int n;
    ... чтение n ...
    vector < vector < pair<int,int> > > g (n);
    ... чтение графа ...
    int s = ...; // стартовая вершина

    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  $g$  is stored in the form of adjacency lists: for each vertex  $v$  list  $g[v]$  contains a list of edges emanating from the vertex, ie, a list of pairs  $\text{pair} < \text{int}, \text{int} >$ , where the first element of the pair - the vertex which leads the edge, and the second element - the weight of the edge.

After reading infest arrays distances  $d[]$ , marks  $u[]$  and ancestors  $p[]$ . Then executed  $n$  iterations. At each iteration, first is the vertex  $v$  having the smallest distance  $d[]$  among 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 viewed all edges emanating from that vertex, and along each edge run relaxation. If relaxation is successful (ie, the distance  $d[to]$  varies), the distance is converted  $d[to]$  and stored ancestor  $p[]$ .

After all the iterations in the array  $d[]$  are the length of the shortest paths to all vertices in the array  $p[]$  - the ancestors of all vertices (except home page  $s$ ). Restore the path to any node  $t$  can be 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 feed, Charles Leiserson, Ronald Rivest, Clifford Stein. **Introduction to Algorithms** [2005]
- Edsger Dijkstra. **A Note on two Problems in connexion with graphs** [1959]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:32  
EDIT: 9 Jul 2009 23:51

## Finding the shortest paths from a given vertex to all other vertices Dijkstra algorithm for sparse graphs

Statement of the problem, the algorithm and its proof can be found. In the article on the general Dijkstra's algorithm .

### Contents [hide]

- Finding the shortest paths from a given vertex to all other vertices  
Dijkstra algorithm for sparse graphs
  - Algorithm
  - Implementation
    - set
    - priority\_queue
    - Getting rid of the pair

## Algorithm

Recall that the complexity of Dijkstra's algorithm consists of two basic operations: Time Spent vertex with the lowest distance  $d[v]$  and time of relaxation, ie while changing the value  $d[to]$ .

In a simple implementation of these operations require, respectively  $O(n)$ , and  $O(1)$  time. Given that the first operation is performed just  $O(n)$  once, and the second -  $O(m)$ , we obtain the asymptotic behavior of the simplest implementation of Dijkstra's algorithm:  $O(n^2 + m)$ .

It is clear that this asymptotic behavior is optimal for dense graphs, ie, when  $m \approx n^2$ . The more sparse graph (ie less  $m$  than the maximum number of edges  $n^2$ ), the less optimal becomes this estimate, and the fault of the first term. Thus, it is necessary to improve the operating times of the first type are not strongly compromising the operating times of the second type.

To do this, use a variety of auxiliary data structures. The most attractive are the **Fibonacci heap**, which allows operation of the first kind  $O(\log n)$  and the second - for  $O(1)$ .

Therefore, when using Fibonacci heaps time of Dijkstra's algorithm will be  $O(n \log n + m)$ , which is almost the theoretical minimum for the algorithm to find the shortest path. By the way, this estimate is optimal for algorithms based on Dijkstra's algorithm, ie, Fibonacci heap are optimal from this point of view (this is about the optimality actually based on the impossibility of the existence of such an "ideal" data structure - if it existed, it would be possible to perform sorting in linear time, which, as you know, in the general case 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 completely different idea than Dijkstra's algorithm, so no contradiction here). However, Fibonacci heap rather difficult to implement (and, it should be noted, have a considerable constant hidden in the asymptotic).

As a compromise, you can use the data structures that enable you to **both types of operations** (in fact, is the removal of the minimum and update element) for  $O(\log n)$ .

Then the time of Dijkstra's algorithm is:

$$O(n \log n + m \log n) = O(m \log n)$$

In the data structure such as C ++ programmers conveniently take a standard container `set` or `priority_queue`. The first is based on a red-black tree, the second - on the binary heap. Therefore `priority_queue` has a smaller constant hidden in the asymptotic behavior, but it has a drawback: it does not support the delete operation element, because of what is necessary to do "workaround" that actually leads to the substitution in the asymptotics  $\log n$  for  $\log m$  (in terms of the asymptotic behavior of this really really does not change anything, but the hidden constant increases).

## Implementation

### set

Let's start with the container `set`. Since the container we need to keep the top, sorted by their values `d[]`, it is convenient to put in a container couples: the first member of the pair - the distance, and the second - the number of vertices. As a result, `set` the pair will be stored automatically sorted by the distance that we need.

```
const int INF = 1000000000;

int main() {
    int n;
    ... чтение n ...
    vector < vector < pair<int,int> > > g (n);
    ... чтение графа ...
    int s = ...; // стартовая вершина

    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's algorithm, the array becomes unnecessary  $u[]$ . His role as the function of finding the vertex 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 the queue until there is at least one vertex. Removed from the queue vertex with the smallest distance, and then run it from the relaxation. Before performing each successful relaxation we first removed from `set` an old pair, and then, after relaxation, add back a new pair (with a new distance  $d[v]$ ).

## priority\_queue

Fundamentally different from here `set` there is, except for the point that removed from the `priority_queue` arbitrary elements is not possible (although theoretically heap support such an operation, in the standard library is not implemented). Therefore it is necessary to make "workaround": the relaxation just will not remove the old couple from the queue. As a result, the queue can be simultaneously several pairs of the same vertices (but with different distances). Among these pairs we are interested in only one for which the element `first` is  $d[v]$ , and all the rest are fictitious. Therefore it is necessary to make a slight modification: the beginning of each iteration, as we learn from the queue the next pair, will check fictitious or not (it is enough to compare `first` and  $d[v]$ ). It should be noted that this is an important modification: if you do not make it, it will lead to a significant deterioration of the asymptotics (up  $O(nm)$ ).

More must be remembered that `priority_queue` organizes the elements in descending order, rather than ascending, as usual. The easiest way to overcome this feature is not an indication of its comparison operator, but simply putting as elements of `first` distance with a minus sign. As a result, at the root of the heap will be provided with the smallest elements of the distance that we need.

```
const int INF = 1000000000;

int main() {
    int n;
    ... чтение n ...
    vector < vector < pair<int,int> > > g (n);
    ... чтение графа ...
    int s = ...; // стартовая вершина

    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 the pair

You can still slightly improve performance if the container is still not keep couples, and only the numbers of vertices. At the same time, of course, you have to reboot the comparison operator for the vertices: compare two vertices must be over distances up to them  $d[]$ .

As a result of the relaxation of the magnitude of the distance from the top of some changes, it should be understood that "in itself" data structure is not rebuilt. Therefore, although it may seem that remove / add items in a 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 remove items from `set`, but not of `priority_queue`, it turns out that this technique is only applicable to `set`. In practice, it significantly increases performance, especially when the distances are used for storing large data types (such as `long long` or `double`).

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 10 Jun 2008 19:37  
EDIT: 24 Aug 2011 15:45

## Bellman-Ford algorithm

Let a directed weighted graph  $G$  with  $n$  vertices and  $m$  edges, and contains some vertex  $v$ . You want to find the length of the shortest path from the top  $v$  to all other vertices.

Unlike Dijkstra's algorithm, the algorithm can also be applied to graphs containing the edges of negative weight. However, if the graph contains a negative cycle, then, of course, some of the shortest path to the vertices might not exist (due to the fact that the weight of a shortest path to be equal to minus infinity); however, this algorithm can be modified to signal the presence of the negative cycle of the weight, or even He concludes the cycle.

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 sub-task 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 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.

### Contents [hide]

- Bellman-Ford algorithm
  - Description of the algorithm
  - Implementation
    - The simplest implementation
    - Improved implementation
    - Restoring ways
  - The proof of the algorithm
  - Case of a negative cycle
  - Tasks in the online judges

## Description of the algorithm

We believe that the graph contains no cycles of negative weight. The case of the presence of a negative cycle will be discussed below in a separate section.

Zavedem array of distances  $d[0 \dots n - 1]$  that after execution of the algorithm will contain the answer to the problem. At the beginning, we fill it as follows:  $d[v] = 0$ , and all other elements  $d[]$  equal to infinity  $\infty$ .

The algorithm itself Bellman-Ford is a few phases. At each phase to view all edges of the graph and the algorithm tries to produce relaxation (relax, easing) along each edge  $(a, b)$  cost  $c$ . Relaxation along the edge - is an attempt to improve the value of  $d[b]$  value  $d[a] + c$ . In fact, it means that we are trying to improve the response for the top  $b$ , using the edge  $(a, b)$  and the current response for the top  $a$ .

It is argued that sufficient  $n - 1$  phase algorithm to correctly calculate the length of the shortest paths in the graph (again, we believe that the negative weight cycles available). For inaccessible peaks distance  $d[]$  will be equal to infinity  $\infty$ .

## Implementation

Algorithm Bellman-Ford, unlike many other graph algorithms, it is more convenient to represent the graph in the form of a list of all the edges (not  $n$  lists of edges - the edges of each vertex). The table of realization of the data structure edge for the edge. The inputs to the algorithm are the number  $n, m$ , a list  $e$  of edges and the number of the starting vertex  $v$ . All rooms vertices are numbered from  $0$  to  $n - 1$ .

### The simplest implementation

The constant  $INF$  is the number of "infinity" - it must be chosen in such a way that it is certainly superior to all possible lengths of the paths.

```

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);
// вывод d, например, на экран
}

```

Checking the "if ( $d[e[j].a] < INF$ )" is needed only if the graph contains edges of negative weight: without such a test would be a relaxation of the vertices to which the path is not found, and would appear incorrect type of distance  $\infty - 1$ ,  $\infty - 2$ etc.

## Improved implementation

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 in vain to view all edges. Therefore, we will keep the flag that something has changed in the current phase or not, and if at some stage nothing has happened, then the algorithm can be stopped. (This optimization does not improve the asymptotic behavior, ie on some graphs will still need all the  $n - 1$ phase, but significantly accelerates the behavior of the algorithm "on average", ie random graphs.)

With this enhancement, it becomes unnecessary to restrict all by hand 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;
    }
    // вывод d, например, на экран
}

```

## Restoring ways

We now consider how the algorithm can be modified Bellman-Ford so that he not only found the length of the shortest paths, but also allows you to recover themselves shortest paths.

For this zavedëm another array  $p[0 \dots n - 1]$  in which each vertex will keep its "ancestor", ie penultimate vertex in the shortest path that leads to it. In fact, the shortest path to some vertex  $a$  is the shortest way to some vertex  $p[a]$ , which attributed to the end of the top  $a$ .

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 the top of the other. Consequently, at the moment we just need to improve in the store  $p[]$ , of which the vertex is improvement occurred.

We present the implementation of the Bellman-Ford with the restoration path to any 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());
            cout << "Path from " << v << " to " << t << ": ";
            for (size_t i=0; i<path.size(); ++i)
                cout << path[i] << ' ';
        }
    }
}

```

Here we first passed by the ancestors, starting from the top  $t$ , and keep all the traversed path in the list  $\text{path}$ . This list is obtained from the shortest path  $v$  to  $t$ , but in reverse order, so we call `reverse` him, and then outputs.

## The proof of the algorithm

Firstly, once we note that for unreachable from  $v$  the vertex algorithm will work correctly: they label  $d[t]$  will remain equal to infinity (as Bellman-Ford algorithm finds some way to all reachable from  $s$  vertex and relaxation at all other vertices will not happen even once).

We now prove the following **statement**: After a  $i$  phase-Bellman Ford algorithm correctly finds all shortest path length (number of edges) does not exceed  $i$ .

In other words, for every vertex  $a$  is denoted by  $k$  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  $k$  this phase the shortest path is found guaranteed.

**Proof**. Consider an arbitrary vertex  $a$  to which there is a path from the starting vertex  $v$ , and consider the shortest path to it:  $(p_0 = v, p_1, \dots, p_k = a)$ . Before the first phase of the shortest path to the summit  $p_0 = v$  found correctly. During the first phase of the edge  $(p_0, p_1)$  has been viewed Bellman-Ford algorithm, therefore, the distance to the summit  $p_1$  was correctly counted after the first phase. Repeating these statements  $k$  again, we see that after  $k$  phase of the distance to the top of the  $p_k = a$  counted correctly, as required.

The last thing to note - this is something that every shortest path can not have more  $n - 1$  ribs. Therefore, the algorithm is sufficient to produce only  $n - 1$  phase. After that, no one is guaranteed to relaxation can not be completed improvement distance to some vertex.

## Case of a negative cycle

Above all, we felt that the negative cycle in the graph does not contain (check 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 vertices of this 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 of Bellman-Ford can **do infinitely relaxation** among all vertices of this cycle and the vertices reachable from it. Consequently, if the numbers do not limit the number of phases  $n - 1$ , then the algorithm will operate indefinitely, continuously improving the distance to these vertices.

Hence we have a **criterion of existence attainable negative weight cycle**: if after  $n - 1$  the phase we perform one more phase, and it happens at least one relaxation, then the graph contains a cycle of negative weight, attainable from  $v$ ; Otherwise, no such cycle.

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's enough to remember the number of the vertex  $x$ , in which there was a relaxation on  $n$ th phase. This node will either be on a cycle of negative weight, or it is reachable from it. To obtain a vertex which lies on the cycle guaranteed sufficiently, for example,  $n$  just once by ancestors, from top  $x$ . Received a number of  $y$  vertices lying on the cycle, you have to walk from the vertex to the ancestors until we get back into the same vertex  $y$  (and it certainly 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 if there is a negative cycle of  $n$  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 a 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  $d[i]$  equal to zero, but not indefinitely - as if we are looking for the shortest path from all the vertices at the same time; for correct detection of the negative cycle will not be affected.

In addition to the topic of this problem - see. Separate article "["Finding the negative cycle in the graph"](#)" .

## Tasks in the online judges

A list of tasks that can be solved using the algorithm of Bellman-Ford:

- E-OLIMP # 1453 "[Ford-Bellman](#)" [Difficulty: Low]
- UVA # 423 "[MPI Maelstrom](#)" [Difficulty: Low]
- UVA # 534 "[Frogger](#)" [Difficulty: Medium]
- UVA # 10099 "[The Tourist Guide](#)" [Difficulty: Medium]
- UVA # 515 "[King](#)" [Difficulty: Medium]

See. Also a list of tasks in the article "["Finding the negative cycle"](#)" .

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:39  
EDIT: 2 Oct 2010 1:35

## Leviticus algorithm of finding the shortest paths from a given vertex to all other vertices

Suppose we are given a graph with  $N$  vertices and  $M$  edges, each of which indicated its weight  $L_i$ .

Also, given the starting vertex  $V_0$ . You want 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 speed of work. See below).

### Description

Let the array  $D[1..N]$  will contain the current shortest path lengths, ie,  $D_i$  - is the current length of the shortest path from vertex  $V_0$  to vertex  $i$ . Originally  $D$  array populated with "infinity", except  $D_{V_0} = 0$ . At the end of the algorithm, this array will contain the final shortest distance.

Let an array  $P[1..N]$  contains current ancestors, ie  $P_i$  - is the pinnacle preceding vertex  $i$  in the shortest path from vertex  $V_0$  to  $i$ . As well as an array  $D$ , array  $P$  gradually varies in the course of the algorithm, and it receives the end of the final value.

Now actually the algorithm Leviticus. At each step, supported by three sets of vertices:

- $M_0$  - vertex distance which has already been calculated (but perhaps not entirely);
- $M_1$  - vertex distance are calculated;
- $M_2$  - vertex distance are not yet calculated.

Vertices in the set  $M_1$  is stored in the form of a bi-directional queue (deque).

Initially, all the vertices are placed in the set  $M_2$ , apart from the vertex  $V_0$ , which is placed in the set  $M_1$ .

At each step of the algorithm we take the top of the set  $M_1$  (We reach the top element of the queue). Let  $V$  - is the selected vertex. Translate this vertex in the set  $M_0$ . Then view all edges emanating from that vertex. Let  $T$  - this is the second end edge (i.e., not equal to  $V$ ), and  $L$  - the length of this edge.

- If  $T$  belongs to  $M_2$ , then  $T$  is transferred to the set  $M_1$  at the end of the queue.  $D_T$  is set equal to  $V + L$ .
- If  $T$  belongs to  $M_1$ , we try to improve the value of  $D_T$ :  $D_T = \min(D_T, D_V + L)$ . The very top  $T$  is never moves in the queue.
- If  $T$  belongs to  $M_0$ , and if  $D_T$  can be improved ( $D_T > D_V + L$ ), then improve  $D_T$ , and  $T$  return to the top of the set  $M_1$ , placing it in the top of the queue.

### Contents [hide]

- Leviticus algorithm of finding the shortest paths from a given vertex to all other vertices
  - Description
  - Implementation Details
  - Asymptotics
  - Implementation

Of course, whenever you update the array D should be updated and the value in the array P.

## Implementation Details

Create an array ID [1..N], in which each vertex will be stored, which set it belongs: 0 - if M<sub>2</sub> (ie, the distance is infinite), 1 - if M<sub>1</sub> (ie, the vertex is queue), and 2 - when M<sub>0</sub> (a path has already been found, the distance is less than infinity).

Queue processing can be realized by a standard data structure deque. However, a more efficient way. First, obviously, a queue at any one time will be stored a maximum of N elements. But secondly, we can add elements and beginning and end of the queue. Therefore, we can arrange a place on the array size N, but you have to loop it. Ie do array Q [1..N], pointers (int) to the first element QH and after the last element of QT. The queue is empty when QH == QT. Adding to the end - just post in Q [QT] and an increase in QT 1; if QT then went beyond the queue (QT == N), then do QT = 0. Adding to the top of the queue - reduce QH 1 if it has moved beyond the stage of (QH == -1), then do QH = N - 1.

The algorithm itself realize exactly according to the description above.

## Asymptotics

I do not know more or less good asymptotic estimate of this algorithm. I have seen only the estimate O (NM) have a similar algorithm.

However, in practice the algorithm has proven itself very well: while it is running, I appreciate as O (M log N) , although, again, this is only **an experimental evaluation**.

## 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);

    ... Read Count ...

    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 ...

}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 19:41  
EDIT: 24 Aug 2011 1:19

## Algorithm Floyd-Uorshella finding the shortest paths between all pairs of vertices

Dan directed or undirected weighted graph  $G$  with  $n$  vertices. You want to find the values of all variables  $d_{ij}$ - the length of the shortest path from vertex  $i$  to vertex  $j$ .

It is assumed that the graph contains no cycles of negative weight (if the response between some pairs of vertices may not exist - it will be infinitely small).

This algorithm was simultaneously published in the article by Robert Floyd (Robert Floyd) and Steven Uorshella (Warshall) (Stephen Warshall) in 1962, on behalf of which the algorithm is called now. However, in 1959, Bernard Roy (Bernard Roy) has published almost the same algorithm, but its publication went unnoticed.

### Contents [hide]

- Algorithm Floyd-Uorshella finding the shortest paths between all pairs of vertices
  - Description of the algorithm
  - Implementation
  - Restoring themselves ways
  - The case of real weights
  - The case of negative cycles

## Description of the algorithm

The key idea of the algorithm - the process of finding a partition of the shortest paths in the **phase**.

Before  $k$ th phase ( $k = 1 \dots n$ ) is considered that the distances matrix  $d$  stored length of the shortest paths, which contain as the only internal vertices of a plurality of vertices  $\{1, 2, \dots, k-1\}$  (vertices we number from unity).

In other words, before  $k$ th phase value  $d[i][j]$  equal to the length of the shortest path from vertex  $i$  to vertex  $j$ , if this path is allowed to go only in vertices with indices less than  $k$  (the beginning and the end of the road do not count).

Easy to see that this property holds for the first phase, it is sufficient in the distance matrix  $d$  to record the adjacency matrix:  $d[i][j] = g[i][j]$  - the cost of an edge from vertex  $i$  to vertex  $j$ . Thus, if between any vertex of not, write down the following values "infinity"  $\infty$ . From the vertex to itself, always record the value  $0$ , it is critical for the algorithm.

Suppose now that we are on  $k$ th phase, and we want to **count the** matrix  $d$  so that it meets the requirements for the already  $k+1$ -th phase. We fix some peaks  $i$  and  $j$ . We there are two fundamentally different cases:

- The shortest path from vertex  $i$  to vertex  $j$ , which is allowed to pass through an additional vertex  $\{1, 2, \dots, k\}$ , **coincides** with the shortest path, which is allowed to pass through the vertices of the set  $\{1, 2, \dots, k-1\}$ .

In this case, the value  $d[i][j]$  does not change when switching from  $k$ th to  $k+1$ -th phase.

- "New" was the shortest way **better than** the "old" way.

This means that the "new" shortest path passes through the top  $k$ . Just note that we do not lose generality, further considering only simple paths (ie paths that do not pass on some vertex twice).

Then we note that if we will divide this "new" way of the apex  $k$  into two halves (one running  $i \Rightarrow k$ , and the other -  $k \Rightarrow j$ ), each of these halves are no longer comes to the top  $k$ . But then it turns out that the length of each of the halves was deemed more on  $k-1$ th phase or even earlier, and it is sufficient to simply take the amount  $d[i][k] + d[k][j]$ , it will give the length of the "new" shortest path.

**Combining** these two cases, we find that on  $k$ th phase is required to recalculate the length of the shortest paths between all pairs of vertices  $i$ , and  $j$  as follows:

```
new_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Thus, all the work that is required to produce  $k$ th phase - is to iterate over all pairs of vertices and recalculate the length of the shortest path between them. As a result, after the  $n$ -th phase in the matrix of distances  $d[i][j]$  will be recorded between the length of the shortest path  $i$  and  $j$ , or  $\infty$ , if paths between these nodes does not exist.

The last remark, which should be done - something that can **not create a separate matrix**  $new_d[]$  for the temporary matrix of shortest paths on  $k$ th phase: all the changes you can make immediately in the matrix  $d[]$ . In fact, if we have improved (reduced) some value in the matrix of distances, we could not degrading the length of the shortest path for any other pairs of vertices processed later.

**Asymptotic** algorithm obviously is  $O(n^3)$ .

## Implementation

The input to the program served graph given in the form of adjacency matrix - a two-dimensional array  $d[]$  size  $n \times n$ , in which each element specifies the length of the edge between the vertices.

It is required that  $d[i][i] = 0$  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 between two if some vertices **are no ribs**, the adjacency matrix was recorded some large number (large enough that it is greater than the length of any path in this graph); then this edge will always be profitable to take, and the algorithm works correctly.

However, if you do not take special measures, in the presence of edges in the graph **of negative weight**, in the resulting matrix can appear number of species  $\infty - 1, \infty - 2$  etc., which, of course, still means that between the corresponding vertices in general there is no way. Therefore, in the presence of negative edges in the graph algorithm Floyd's better to write so that it did not perform transitions from those states that 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]);
```

## Restoring themselves ways

Easy to maintain additional information - the so-called "ancestors" in which it will be possible to restore himself the shortest path between any two given vertices **as a sequence of vertices**.

It's enough apart distance matrix  $d[]$  also support **matrix ancestors**  $p[]$  that for every pair of vertices will contain a number of phases, which was obtained by the shortest distance between them. It is clear that the phase number is not more than the "average" top title of the shortest path, and now we just need to find the shortest path between the vertices  $i$  and  $p[i][j]$ , and between  $p[i][j]$  and  $j$ . This yields a simple recursive algorithm for the shortest path recovery.

## The case of real weights

If the weight of the edges of the graph is not an integer, and real, we should take into account the errors that inevitably arise when dealing with floating-point types.

With regard to the algorithm Floyd unpleasant special effect of these errors become what distance algorithm results may take much minus because **of accumulated errors**. In fact, if the first phase error has occurred  $\Delta$ , then in the second iteration of this error may have in turn  $2\Delta$  on the third - in  $4\Delta$ , and so on.

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 has cycles of negative weight, then formally algorithm Floyd-Uorshella inapplicable to such a graph.

In fact, for those pairs of vertices  $i$  and  $j$  between which it is impossible to go into a cycle of negative weight, the 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 way between them), Floyd algorithm finds an answer in a certain number of (probably strongly negative, but not necessarily). Nevertheless, we can improve the algorithm Floyd that he carefully handle such pairs 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." Thus, even for a given graph has worked usual algorithm Floyd. Then between the tops  $i$  and  $j$  the shortest path does not exist, and then only if there exists a vertex  $t$  accessible from  $i$  and from which is achievable  $j$  for which the following  $d[t][t] < 0$ .

In addition, when using the Floyd algorithm for graphs with negative cycles should be remembered that arise in the process of distance can go into much less exponentially with each phase. Therefore, you should take action against integer overflow, limiting all distances from the bottom of any value (for example,  $-\text{INF}$ ).

For more information about this task, see. A separate article: "[Finding a negative cycle in the graph](#)".

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Sep 2010 16:13  
EDIT: May 3, 2012 1:35

## Shortest path of fixed length, the number of tracks fixed length

The following describes the solutions of these two problems, built on the same idea: to reduce the problem to the construction of the power matrix (with the usual multiplication, and modified).

### Contents [hide]

- Shortest path of fixed length, the number of tracks fixed length
  - The number of paths of fixed length
  - Shortcuts fixed length
  - Generalization to the case when the required path length, no more than a predetermined length

## The number of paths of fixed length

Given a directed graph is unweighted  $G$  with  $n$  vertices, and contains an integer  $k$ . Is required for each pair of vertices  $i$  and  $j$  the number of ways to find between these vertices consisting of exactly  $k$  edges. Ways at the same time consider arbitrary, not necessarily simple (ie, the vertices can be repeated any number of times).

We assume that the graph is given **adjacency matrix**, ie, matrix  $g$  size  $n \times n$ , where each element  $g[i][j]$  is equal to one if there is between the edge nodes, and zero if there is no edge. Described below, and the algorithm works in the case of multiple edges: if between some vertices  $i$  and  $j$  is at once  $m$  the ribs, the adjacency matrix should record this number  $m$ . Also, the algorithm correctly takes into account the loops in the graph, if any.

Obviously, in this form, **the adjacency matrix** of the graph is the **answer to the problem when  $k = 1$**  - it contains a number of paths of length 1 between each pair of vertices.

Solution will be constructed **iteratively**: let the answer for some  $k$  is found, we show how to build it  $k + 1$ . Denoted by  $d_k$  the matrix found answers to  $k$ , and through  $d_{k+1}$  - a matrix of responses you want to build. Then clear the following formula:

$$d_{k+1}[i][j] = \sum_{p=1}^n d_k[i][p] \cdot g[p][j].$$

Easy to see that the above formulas - nothing but the product of two matrices  $d_k$  and  $g$  in the usual sense:

$$d_{k+1} = d_k \cdot g.$$

Thus, the **solution** of 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 matrix in the degree can be produced efficiently using the algorithm **binary exponentiation**.

Thus, the resulting solution has the asymptotic behavior  $O(n^3 \log k)$  and is in a binary construction  $k$  of the degree of adjacency matrix of the graph.

## Shortcuts fixed length

Given a directed weighted graph  $G$  with  $n$  vertices, and contains an integer  $k$ . Is required for each pair of vertices  $i$ , and  $j$  find the length of the shortest path between the vertices of exactly  $k$  edges.

We assume that the graph is given **adjacency matrix**, ie, matrix  $g$  [size  $n \times n$ ], where each element  $g[i][j]$  contains the length of the edge from vertex  $i$  to vertex  $j$ . If between some vertices of the edge is not, then the corresponding element of the matrix is taken to be infinity  $\infty$ .

Obviously, in this form, **the adjacency matrix** of the graph is the **answer to the problem when  $k = 1$**  - it contains the length of the shortest paths between each pair of vertices, or  $\infty$  if the length of the path  $1$  does not exist.

Solution will be constructed **iteratively**: let the answer for some  $k$  is found, we show how to build it  $k + 1$ . Denoted by  $d_k$  the matrix found answers to  $k$ , and through  $d_{k+1}$  a matrix of responses you want to build. Then clear the following formula:

$$d_{k+1}[i][j] = \min_{p=1 \dots n} (d_k[i][p] + g[p][j]).$$

Take a good look at this formula, it is easy to draw an analogy with matrix multiplication: in fact, the matrix  $d_k$  is multiplied by the matrix  $g$ , only the operation of multiplication instead of the sum of all  $p$  the minimum is taken over all  $p$ :

$$d_{k+1} = d_k \odot g,$$

where the operation  $\odot$  of 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, the **solution** to 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 exponentiation with this multiplication operation can be performed efficiently using the algorithm **binary exponentiation**, because the only required for a feature - associativity of multiplication - obviously there.

Thus, the resulting solution has the asymptotic behavior  $O(n^3 \log k)$  and is in a binary construction  $k$  of the degree of adjacency matrix of a modified matrix multiplication.

## Generalization to the case when the required path length, no more than a predetermined length

The above solutions solve problems when you need to consider the way a certain, fixed length. However, these solutions can be adapted to solve problems when you need to consider ways of containing **not more** than a specified number of edges.

You can do it, slightly modified input graph. For example, if we are interested only path ending in a particular vertex  $t$ , the graph can **add a loop**  $(t, t)$  of zero weight.

If we are still interested in the answers for all pairs of vertices, then simply adding loops to all vertices spoil the answer. Instead, you can **bifurcate** each node: for each vertex  $v$  to create additional vertex  $v'$ , edge hold  $(v, v')$  and add a loop  $(v', v')$ .

Deciding on a modified graph the problem of finding ways of fixed length, the answers to the original problem will be obtained as the responses between the vertices  $i$  and  $j'$  (ie, additional peaks - it tops-end, in which we can "whirl" necessary number of times).

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 10 Jun 2008 20:55  
 EDIT: 18 Aug 2011 22:50

## The minimum spanning tree. Prim's algorithm

Given a weighted undirected graph  $G$  with  $n$  vertices and  $m$  edges. Required to find a subtree of the 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, with every vertex can reach any other exactly one simple way.

This subtree is called the minimum spanning tree or just **the minimum spanning tree**. It is easy to understand that any frame will necessarily contain  $n - 1$  an edge.

In a **natural setting**, this problem is as follows: there are  $n$  cities, and for each pair of compounds known to cost them dear (or know that they can not connect). Required to connect all of the city so that you can get from any city to another, and with the cost of construction of roads would be minimal.

### Contents [hide]

- The minimum spanning tree. Prim's algorithm
  - Prim's algorithm
    - Description of the algorithm
    - Proof
  - Sale
    - Trivial Pursuit: Algorithms for  $O(nm)$  and  $O(n^2 + m \log n)$
    - The case of dense graphs: an algorithm for  $O(n^2)$
    - The case of sparse graphs: an algorithm for  $O(m \log n)$
    - The analogy with the Dijkstra algorithm
  - Properties of the minimum spanning tree

## Prim's algorithm

This algorithm is named after American mathematician Robert Prima (Robert Prim), which opened this algorithm in 1957. However, even in 1930, this algorithm was discovered by 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 the minimum spanning tree is built gradually, adding he edges one by one. Originally skeleton relies consisting of a single vertex (it can be chosen arbitrarily). Then the minimum weight is selected edge emanating from this vertex and added to the minimum frame. Thereafter skeleton already contains two peaks and looking for an edge, and adds the minimum weight, having one end in one of two selected peaks, and the other - on the contrary, all other than these two. And so on, i.e. whenever sought a minimum weight edge, one end of which is - has already taken in the top of the skeleton, and the other end - is not taken, and this edge is added to the frame (if there are multiple edges, you can take any). This process is repeated until the frame will not yet contain all peaks (or, equivalently,  $n - 1$  a rib).

As a result, the skeleton will be built, is minimal. If the graph was not initially connected, then the skeleton will not be found (the number of selected edges will be less  $n - 1$ ).

## Proof

Suppose that the graph  $G$  was connected, ie, answer there. We denote by  $T$  skeleton found Prim's algorithm, and through  $S$  - the minimum spanning tree. Obviously, that  $T$  is indeed the backbone (ie, a subtree of the graph  $G$ ). We show that the weight  $S$  and  $T$  the same.

Consider the first time when  $T$  the rib are added, is not included in the optimum backbone  $S$ . We denote this edge through  $e$  the ends of it - through  $a$  and  $b$ , and the set included at that time in the skeleton of the vertices - through  $V$  (according to the algorithm  $a \in V$ ,  $b \notin V$  or vice versa). Advantageously the skeleton  $S$  vertices  $a$  and  $b$  connected in some way  $P$ ; in this way we find any edge  $g$ , one end of which lies in  $V$ , and the other - no. Since the Prim algorithm chose the rib  $e$  edges instead  $g$ , it means that the weight of the edge  $g$  is greater than or equal to the weight of the edge  $e$ .

Now remove from the  $S$  edge  $g$ , and add an edge  $e$ . By just said the weight of the core as a result could not have increased (decreased he also could not because  $S$  it was the best). In addition,  $S$  no longer a skeleton (that connection is not broken, it is easy to make sure we closed the way  $P$  to the ring, and then remove them from this cycle, one edge).

Thus we have shown that it is possible to select the optimal framework  $S$  so that it will include an edge  $e$ . Repeating this procedure as many times, we see that we can choose the best frame  $S$  so that it matches with  $T$ . Consequently, the weight of the construction of algorithms Prima  $T$  minimal, as required.

## Sale

Time of the algorithm depends essentially on how we produce search next minimum matching edges among edges. There can be different approaches lead to different asymptotics and different implementations.

### Trivial Pursuit: Algorithms for $O(nm)$ and $O(n^2 + m \log n)$

If you look for an edge every time just browsing among all possible options, then asymptotically be required viewing  $O(m)$  edges to find among all the valid edge with the lowest weight. The total amount to the asymptotic behavior of the algorithm in this case  $O(nm)$ , in the worst case there  $O(n^3)$  - too slow algorithm.

This algorithm can be improved if the view each time, not all the edges, and only one edge of each pre-selected top. For this example, you can sort the edges of each vertex in ascending order of the weights, and store a pointer to the first valid edge (recall permissible only those edges that lead to the set is not selected vertices). Then, if these pointers to recalculate each time you add an edge to the backbone, the total asymptotic behavior of the algorithm will  $O(n^2 + m)$ , but first need to sort all edges for  $O(m \log n)$  that in the worst case (for dense graphs) gives the asymptotic behavior  $O(n^2 \log n)$ .

Below we consider two slightly different algorithm: for dense and sparse graphs, eventually getting much better asymptotic behavior.

### The case of dense graphs: an algorithm for $O(n^2)$

Approaching the matter search smallest rib on the other hand: for each is not selected will be stored a minimum edge going into an already selected vertex.

Then, at the current step to make the choice of the minimum edges, you just have to see these minimum ribs at each vertex is not selected yet - be the asymptotic behavior  $O(n)$ .

But now, when added to the next frame edges and vertices of these pointers must be recalculated. Note that these pointers can only decrease, ie each vertex not yet viewed any need to leave it without changing the pointer or give it the weight of the edge in the newly added vertex. Therefore, this phase can also be done for  $O(n)$ .

Thus, we have an option of Prim's algorithm with asymptotic behavior  $O(n^2)$ .

In particular, such an implementation is particularly useful for solving the so-called **problem of the Euclidean minimum spanning tree** when given  $n$  points in the plane, the distance between which is measured by the standard Euclidean metric, and you want to find the skeleton of minimum weight that connects them all (and adding new vertices anywhere elsewhere is prohibited). This problem is solved by the algorithm described here for the  $O(n^2)$  time and  $O(n)$  memory, which does not turn out to achieve [Kruskal's algorithm](#).

The implementation of Prim's algorithm for a graph specified by adjacency matrix  $g[]$ :

```
// входные данные
int n;
vector < vector<int> > g;
const int INF = 1000000000; // значение "бесконечность"

// алгоритм
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) {
    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  $n$  and the matrix  $g[]$  size  $n \times n$ , which marked the edges of weight, and cost of  $INF$ , if there is no corresponding edge. The algorithm supports three arrays: flag  $used[i] = true$  means that the top  $i$  is included in the frame, the size of  $min\_e[i]$  the smallest allowable weight keeps the edges from a vertex  $i$ , and the element  $sel\_e[i]$  contains the smallest end of the rib (it is necessary to bring the edges of the reply). The algorithm makes  $n$  steps, each of which selects the top of  $v$  the lowest mark

`min_e`, marks it `used`, and then looks at all the edges of this vertex, recounting their labels.

## The case of sparse graphs: an algorithm for $O(m \log n)$

In the above-described algorithm, one can see the operation of finding the minimum standard in the set and change values in the set. These two operations are classic, and perform many data structures, for example, implemented in C++, the red-black tree set.

Within the meaning of the algorithm remains exactly the same, but now we can find the minimum edge over time  $O(\log n)$ . On the other hand, the time for recalculation `n` pointers now be  $O(n \log n)$  even worse than in the above algorithm.

If we consider that there will be  $O(m)$  calculations of the original indexes and  $O(n)$  searches the minimum edge, then the asymptotic behavior of the total amount  $O(m \log n)$  - for sparse graphs is better than both of the above algorithm, but in dense graphs, this algorithm is slower than the previous one.

The implementation of Prim's algorithm for a graph specified by adjacency lists `g[]`:

```
// входные данные
int n;
vector < vector < pair<int,int> > > g;
const int INF = 1000000000; // значение "бесконечность"

// алгоритм
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 `n` and `n` adjacency lists: `g[i]` - a list of all the edges emanating from the vertex `i`, in the form of pairs (second end edge weight of the edge). The algorithm maintains two arrays: the value `min_e[i]` keeps the weight of the smallest

allowable edges from the vertex  $i$  and the element  $\text{sel\_e}[i]$  contains the smallest end of the rib (it is necessary to bring the edges of the reply). Additionally, a queue  $q$  of all the nodes in increasing order of their labels  $\min\_e$ . The algorithm makes  $n$  steps, each of which selects the top of  $v$  the lowest mark  $\min\_e$  (simply 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 we place a new back).

## The analogy with the Dijkstra algorithm

In the two algorithms described just seen quite a clear analogy with the Dijkstra's algorithm : it has the same structure ( $n - 1$  phase, each of which first selects the optimal edge is added to the response, and then recalculated values for all vertices not yet selected). Moreover, Dijkstra's algorithm also has two options for implementation: for  $O(n^2)$  and  $O(m \log n)$  (of course we here do not consider the possibility of using complex data structures to achieve even smaller asymptotics).

If you look at the Prim algorithm and Dijkstra's more formally, it turns out that they are all identical to each other except for **the weighting function** peaks: in Dijkstra's algorithm at each vertex supported length of the shortest path (ie the sum of the weights of some edges), the Prim's algorithm to each vertex is attributed only 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 see all the edges  $(v, to)$  of the vertex, the algorithm Prima pointer  $to$  is updated weight of the edge  $(v, to)$ , and Dijkstra's algorithm - mark distance  $d[to]$  updated sum labels  $d[v]$  and edge weight  $(v, to)$ . The rest of these two algorithms can be considered identical (though they are very different and solve the problem).

## Properties of the minimum spanning tree

- **Maximum frame** can also be found Prim's algorithm (for example, replacing all the weights of edges on the opposite: the algorithm does not require the non-negativity of the weights of edges).
- The minimum spanning tree is **unique**, if the weights of all edges are different. Otherwise, there may be some minimum spanning tree (which one will be selected Prim's algorithm depends on the order in view of edges / vertices with the same weights / pointers)
- The minimum spanning tree is also the core, **the minimum for the product of all edges** (assuming that all weights are positive). In fact, if we replace the weights of all edges to their logarithms, it is easy to notice that in the algorithm will not change anything, and will be found the same edges.
- The minimum spanning tree is a spanning tree with minimum weight **of the heaviest edges**. Most clearly this statement is understandable if we consider the work of **Kruskal's algorithm**.
- **The criterion of minimal** core: core is minimal if and only if for every edge not belonging to the skeleton, the cycle formed by the addition of this edge to the core, contains no harder edges of the rib. In fact, if for some edge turned out that it is easier for some ribs formed loop, it is possible to obtain a lighter skeleton (adding this edge into the core, and removing the heaviest edge of the loop). If this condition is

not met for any of the edges, all the edges do not improve the weight of the core when they are added.

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 10 Jun 2008 22:03  
 EDIT: 10 Jun 2008 22:05

## The minimum spanning tree. Kruskal's algorithm

Given a weighted undirected graph. Required to find a subtree of the graph, which would connect all the vertices, and thus has the lowest weight (ie, the sum of the weights of edges) of all. This subtree is called the minimum spanning tree or just the minimum spanning tree.

It will discuss several important facts related to the minimum spanning tree, then be considered Kruskal's algorithm in its simplest implementation.

### Properties of the minimum spanning tree

- The minimum spanning tree is unique, if the weights of all edges are different . Otherwise, there may be a minimum number of cores (specific algorithms typically receive one of the possible core).
- The minimum spanning tree is also a skeleton with a minimum product of the weights of edges. (proved it's easy enough to replace the weights of all edges to their logarithms)
- The minimum spanning tree is also a skeleton with a minimum weight of the heavy edges . (This follows from the validity of Kruskal's algorithm)
- The skeleton of the maximum weight is sought is similar to the skeleton of minimum weight, enough to change the signs of all the ribs on the opposite 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 his tree, and then gradually brings these trees, combining the two at each iteration of some tree some edge. Before the start of the algorithm, all edges are sorted by weight (in decreasing order). Then begins the process of unification: all edges are moving from first to last (in the sort order), and if the current edges of the ends belong to different subtrees, these subtrees are merged, and the edge is added to the answer. At the end iterate over 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 runs in  $O(M \log N + N^2)$  . Sort ribs require  $O(M \log N)$  operations. Vertices belonging to a particular subtree stored simply by using an array tree\_id - there is stored for each vertex tree number to which it belongs. For each edge we have  $O(1)$  determine whether it belongs to the ends of different trees. Finally, the union of the two trees is carried out for the  $O(N)$  simply passes through the array tree\_id. Given that all the operations of union is  $N-1$ , we obtain the asymptotic behavior of  $O(M \log N + N^2)$  .

```
int m;
vector<pair<int, pair<int, int>>> g(m); // Weight - the top 1 - 2 vertex

int cost = 0;
vector<pair<int, int>> res;
```

### Contents [hide]

- The minimum spanning tree. Kruskal's algorithm
  - Properties of the minimum spanning tree
  - Kruskal's algorithm
  - The simplest implementation
  - Improved implementation

```
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;
    }
}
```

## Improved implementation

Using the data structure "system of disjoint sets" can write faster implementation of [Kruskal's algorithm with the asymptotic O \(M log N\)](#).

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:05  
EDIT: 3 May 2010 23:29

## The minimum spanning tree. Kruskal's algorithm with a system of disjoint sets

A description of the problem and the algorithm of Kruskal see. [here](#).

There will be reviewed implementation using a data structure "a system of disjoint sets" (DSU), which will reach the asymptotic behavior of  $O(M \log N)$ .

### Description

Just as in the simple version of Kruskal's algorithm, we can sort all edges in non-decreasing weight. Then place each vertex in your tree (ie their set) by calling the DSU MakeSet - it will take a total of  $O(N)$ . Loop through all the edges (in sorted order) and for each edge in  $O(1)$  determine whether it belongs to the ends of different trees (with two calls FindSet  $O(1)$ ). Finally, the union of two trees will be calling the Union - also in  $O(1)$ . Total we obtain the asymptotic behavior of  $O(M \log N + N + M) = O(M \log N)$ .

### Implementation

To reduce the volume of code and carry out all operations are not as separate functions, but 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;
}

... In the function main (): ...

int m;
vector <pair <int, pair <int, int>>> g; // Weight - the top 1 - 2 vertex
... Read Count ...

int cost = 0;
vector <pair <int, int>> res;

sort (g.begin (), g.end ());
```

### Contents [hide]

- The minimum spanning tree. Kruskal's algorithm with a system of disjoint sets
  - Description
  - Implementation

```
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 += 1;
        res.push_back (g [i] .second);
        dsu_unite (a, b);
    }
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: 5 Nov 2008 20:17  
EDIT: 23 Aug 2010 22:24

## Kirchhoff matrix theorem. Finding the number of spanning trees

Ask a connected undirected graph of its adjacency matrix. Multiple edges in the graph are allowed. Required to count the number of different spanning trees of the graph.

The below formula belongs Kirchhoff (Kirchhoff), who proved it in 1847

### Kirchhoff matrix theorem

Take the adjacency matrix of the graph G, we replace each element of this matrix to the opposite, and instead of the diagonal element of  $A_{i,i}$  put the degree of the vertex  $i$  (if there are multiple edges, the vertex degree they are considered with its multiplicity). Then, according to the Kirchhoff matrix theorem, all the cofactors of the matrix are equal, and equal to the number of spanning trees of the graph. For example, you can delete the last row and the last column of the matrix, and the modulus of its determinant is equal to the required number.

Determinant of the matrix can be found for the  $O(N^3)$  using [the Gauss method](#) or [the method Kraut](#).

The proof of this theorem is quite difficult and is not presented here (see., Eg, coming VB "The problem of dimers and Kirchhoff theorem").

### Communication with the laws of Kirchhoff in an electrical circuit

Between the Kirchhoff matrix theorem and the laws of Kirchhoff for the electrical circuit has an amazing relationship.

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  $j$ , the circuit is:

$$R_{ij} = |T^{(i,j)}| / |T^j|$$

where the matrix  $T$  is obtained from the matrix  $A$  reverse resistance of the conductor ( $A_{ij}$  - the opposite number to the resistance of the conductor between

### Contents [hide]

- Kirchhoff matrix theorem. Finding the number of spanning trees
  - Kirchhoff matrix theorem
  - Communication with the laws of Kirchhoff in an electrical circuit

points i and j) transformation described in the Kirchhoff matrix theorem, and the notation  $T^{(i)}$  denotes the row and column with cancellation number i, and  $T^{(i, j)}$  - with cancellation of two columns and rows i and j.

Kirchhoff theorem gives this formula geometric meaning.

# MAXimal

home  
algo  
bookz  
forum  
about

## Prüfer code. Cayley formula. The number of ways to make a graph connected

In this article we consider the so-called **code Prüfer**, which is a way of unambiguous labeled tree by a sequence of numbers.

Use the code Prüfer showing proof of **Cayley's formula** (specify the number of spanning trees in a complete graph), as well as solution of the problem of the number of ways to add a given graph edges to turn it into a cohesive.

**Note**. We will not consider a tree consisting of a single vertex - a special case, in which many of the statements degenerate.

Added: 26 Mar 2012 1:00  
EDIT: Mar 26 2012 1:00

### Contents [hide]

- Prüfer code. Cayley formula. The number of ways to make a graph connected
  - Prüfer code
    - Building code Prüfer for the tree
    - Building code Prüfer for this tree in linear time
    - Some properties codes Prüfer
    - Recovering tree by its code Prüfer
    - Restoration of the code tree in linear time Prüfer
    - One correspondence between trees and codes Prüfer
  - Cayley formula
  - The number of ways to make a graph connected
  - Tasks in the online judges

## Prüfer code

Prüfer code - a way to mutually unambiguous labeled trees with  $n$  vertices by a sequence  $n - 2$  of integers in the interval  $[1; n]$ . In other words, the code Prüfer - a **bijection** between all spanning trees of a complete graph and numerical sequences.

Although the use Prüfer code for storing and manipulating trees is impractical because of the specificity of the representations, codes Prüfer 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 Prüfer for the tree

Prüfer code is constructed as follows. Will  $n - 2$  once prodelyvat procedure: choose a tree leaf with the lowest number, delete it from the tree, and add the code number of the vertex Prüfer, which was associated with this sheet. Finally, in the tree there will be only 2the top and this algorithm ends (number of vertices explicitly written in the code).

Thus, for a given code tree Prüfer - a sequence of  $n - 2$  numbers, where each number - the number of vertices associated with the lowest point on the sheet - i.e. this number in the interval  $[1; n]$ .

Algorithm for computing code Prüfer easy to implement with the asymptotic behavior  $O(n \log n)$ , simply maintaining a data structure to extract a minimum (for example, set <> or priority\_queue <> in the language C ++), containing a list of all the current sheet:

```
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 can be realized Prüfer code and linear time, which is described in the next section.

## Building code Priifer for this tree in linear time

We give here a simple algorithm, which has the asymptotic behavior  $O(n)$ .

The essence of the algorithm is to store a **moving pointer**  $ptr$  which will always move only in the direction of increasing numbers of vertices.

At first glance, this is impossible, because in the process of building code Prüfer number of leaves can both increase and **decrease**. But it is easy to note that the reduction occurs only in one case: if you remove the current code page of his ancestor has a smaller number (this will be the ancestor of the minimum sheet and removed from the tree the next step Prüfer code). Thus, the reduction of cases can be treated in time  $O(1)$ , and nothing prevents the construction of an algorithm with **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 of here - `prufer_code()` that returns Prüfer code for wood, set in global variables `n`(number of vertices) and `g`(adjacency lists that define the graph). Initially, we find for each vertex of its ancestor `parent[i]`- ie of ancestors that this vertex will have the time of disposal of wood (all that we can find in advance, using the fact that the maximal vertex  $n - 1$  never removed from the tree). Also, we find for each vertex her degree `degree[i]`. Variable `ptr`- is moving the pointer ("candidate" for the minimum list) that varies only ever upward. Variable `leaf`- is the current list with a minimum number. Thus, each iteration of code Priifer is to add `leaf`in response, as well as verification, if there were no `parent[leaf]`less than the current candidate `ptr`if there were fewer, we simply assign `leaf = parent[leaf]`, and otherwise - move the pointer `ptr` to the next sheet.

How easily can see the code, the asymptotic behavior of the algorithm is actually  $O(n)$ a pointer `ptr`to undergo a  $O(n)$ change, and all other parts of the algorithm is obviously working in linear time.

## Some properties codes Priifer

- Upon completion of construction of Prüfer code in the tree will remain undeleted two vertices. One of them will surely be a vertex with the maximum number -  $n - 1$ and that's about the other vertex nothing definite can be said.
- Each vertex is found in the code Priifer a 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 it is equal to one degree - ie at this point all the adjacent edges except one have been removed. (For the two remaining peaks after building code this statement is also true.)

## Recovering tree by its code Priifer

To restore the tree is sufficient to note from the previous paragraph that the degrees of all vertices in the target tree, we already know (and can calculate and store in an array `degree[]`). Therefore, we can find all the leaves, and, accordingly, the smallest number plate - which was removed in the first step. This sheet was connected to the top, the number of which is written in the first cell Prüfer code.

Thus, we find the first edge, remote Priifer code. Add this edge back, then reduce the level of `degree[]`both ends of the ribs.

We will repeat this operation until you have reviewed all of the code Priifer: look with minimal vertex `degree = 1`, combine it with another vertex Prüfer code, decrease `degree[]`at both ends.

In the end we are left with only two vertices with `degree = 1`- those peaks that algorithm Priifer left undeleted. Connect them an edge.

The algorithm is complete, the desired tree is built.

**Implement** this algorithm easily during  $O(n \log n)$ : maintaining a data structure to retrieve the minimum (e.g., set <>or priority-queue <>in C++) numbers of all vertices having `degree = 1`and extracting from it at least one time.

Let us give the proper implementation (where the function `prufer_decode()`returns a list of the edges of the desired 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;
}

```

## Restoration of the code tree in linear time Priifer

To obtain an algorithm with linear asymptotic behavior can apply the same technique that was used to produce a linear algorithm for computing code Priifer.

In fact, in order to find the lowest-numbered sheet optionally start a data structure for retrieving a minimum. Instead, you will notice that, after we find and treat the current sheet, he adds into consideration only one new vertex. Therefore, we can do a moving pointer with a variable that contains 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 codes Priifer

On the one hand, for each tree there is exactly one Prüfer code corresponding to it (this follows from the definition of Prüfer code).

On the other hand, the correct reconstruction algorithm Prüfer code tree, it follows that any Prüfer code (i.e., a

sequence of  $n - 2$  numbers, where each number is in the interval  $[1; n]$ ) corresponds to a tree.

Thus, all the trees and all the codes Prüfer form **one to one correspondence**.

## Cayley formula

Cayley formula states that **the number of spanning trees in full labeled graph** of  $n$  vertices is equal to:

$$n^{n-2}.$$

There are many **proofs** of this formula, but the proof using codes Prüfer clearly and constructively.

In fact, any set of  $n - 2$  numbers from the interval  $[1; n]$  corresponds uniquely to a tree of  $n$  nodes. All different codes Prüfer  $n^{n-2}$ . As in the case of a complete graph of  $n$  vertices as the core fits any tree, and the number of spanning trees of the same  $n^{n-2}$ , as required.

## The number of ways to make a graph connected

Power Prüfer codes is that they allow you to get a general formula than formula Cayley.

So, given the graph of  $n$  vertices and  $m$  edges; let  $k$ - the number of connected components in this graph. Required to find the number of ways to add  $k - 1$  an edge to a graph become connected (obviously  $k - 1$  edge - the minimum number of edges to make a graph connected).

We derive a ready formula for solving this problem.

We denote by  $s_1, \dots, s_k$  the size of the connected components of this graph. Since adding the edges of connected components within the forbidden, it turns out that the problem is very similar to searching for the number of spanning trees in a complete graph of  $k$  vertices: but the difference here is that each node has its own "weight"  $s_i$ : each edge adjacent to  $i$ th vertex multiplies the answer to  $s_i$ .

Thus, to count the number of ways is important what extent have all the  $k$  vertices in the core. To obtain equations for the problem it is necessary to summarize the answers to all possible degrees.

Let  $d_1, \dots, d_k$ - the degrees of vertices in the core. The sum of the vertex degrees is equal to twice the number of edges, so:

$$\sum_{i=1}^k d_i = 2k - 2.$$

If  $i$ th vertex has degree  $d_i$ , the Prüfer code it enters  $d_i - 1$  again. Prüfer code for the tree of  $k$  vertices has length  $k - 2$ . The number of ways to select a set  $k - 2$  of numbers, where the number  $i$  occurs exactly  $d_i - 1$  once as well **multinomial coefficient** (similar to [the binomial coefficients](#)):

$$\binom{k-2}{d_1-1, d_2-1, \dots, d_k-1} = \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}.$$

Given the fact that each edge adjacent to  $i$ th top, multiplies the response  $s_i$  we get that response, provided that the degrees of vertices are  $d_1, \dots, d_k$  equal:

$$s_1^{d_1} \cdot s_2^{d_2} \cdot \dots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}.$$

For the answer to the problem, we must add this formula over all admissible sets  $\{d_i\}_{i=1}^{i=k}$ :

$$\sum_{\substack{d_i \geq 1, \\ \sum_{i=1}^k d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdot \dots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}.$$

For coagulation of this formula we use the definition of a 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:

$$s_1 \cdot s_2 \cdot \dots \cdot s_k \cdot (s_1 + s_2 + \dots + s_k)^{k-2} = s_1 \cdot s_2 \cdot \dots \cdot s_k \cdot n^{k-2}.$$

(This formula is valid and  $k = 1$ , although formally in the proof of it should not have.)

## Tasks in the online judges

Tasks in the online judges, which use codes Priifer:

- [UVA # 10843 "Anne's Game"](#) [Difficulty: Low]
- [TIMUS # 1069 "Code Priifer"](#) [Difficulty: Low]
- [Codeforces 110D "Evidence"](#) [Difficulty: Medium]
- [TopCoder SRM 460 "TheCitiesAndRoadsDivTwo"](#) [Difficulty: Medium]

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 10 Jun 2008 22:13  
EDIT: 24 Aug 2011 1:45

## Finding a negative cycle in the graph

Dan directed weighted graph  $G$  with  $n$  vertices and  $m$  edges. You want to find in it any **negative weight cycle**, if there is one.

When another formulation of the problem - you want to find **all pairs of vertices** such that there is a path between any number of small length.

These two options to solve the problem it is convenient different algorithms, so the following will be considered both of them.

One common "life" productions of this problem - the following: known **exchange rates**, ie, Courses transferred from one currency to another. You want to know whether a certain sequence of exchanges benefit, ie, started with one unit of any currency, to receive as a result of more than one unit of the same currency.

### Contents [hide]

- Finding a negative cycle in the graph
  - The decision by the algorithm of Bellman-Ford
  - The decision by the algorithm Floyd-Uorschella
  - Tasks in the online judges

## The decision by the algorithm of Bellman-Ford

[Bellman-Ford algorithm](#) to validate the presence or absence of a negative weight cycle in the graph, and if present - to find one of these cycles.

We will not go into detail here (which are described in the [article on the algorithm of Bellman-Ford](#)), and give only the result - how the algorithm works.

Done  $n$  iterations Bellman-Ford, and if at the last iteration has been no change - that the negative cycle in the graph no. Otherwise, take the top, the distance to which has changed and will go on from her ancestors until it will enter the ring; 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) {
        x = -1;
        for (int j=0; j<m; ++j)
            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 found.";
    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] << ' ';
    }
}

```

## The decision by the algorithm Floyd-Uorshella

Floyd-Uorshella algorithm can solve the second statement of the problem - when you need to find all pairs of vertices  $(i, j)$  between which the shortest path does not exist (ie, it has an infinitesimal amount).

Again, a more detailed explanation is contained in [the description of the algorithm Floyd-Uorshella](#), and here we give only the result.

Once the algorithm Floyd-Uorshella work for the input graph, brute over all pairs of vertices  $(i, j)$ , and for each such pair check infinitesimal shortest path from  $i$  to  $j$  or not. For this brute over the top of the third  $t$ , and if it turned out  $d[t][t] < 0$  (ie it is in a cycle of negative weight), and she is reachable from  $i$  and out of it is achievable  $j$  - the path  $(i, j)$  may be an 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;

```

## Tasks in the online judges

A list of tasks that need to search for a cycle of negative weight:

- UVA # 499 "Wormholes" [Difficulty: Low]
- UVA # 104 "Arbitrage" [Difficulty: Medium]
- UVA # 10557 "XYZZY" [Difficulty: Medium]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:14  
EDIT: 10 Jun 2008 22:15

## Finding Euler path of O (M)

Euler path - a path in the graph that passes through all his ribs. Eulerian path - a Euler path is a cycle.

The challenge is to find an Euler path in **an undirected multigraph with loops**.

### Contents [hide]

- Finding Euler path of O (M)
  - Algorithm
  - The problem of dominoes
  - Implementation

### Algorithm

First check whether there is an Euler path. Then find all the 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, we find an Euler cycle, then remove the extra edge.

To test whether there is an Euler path, you must use the following theorem. Eulerian path exists if and only if the degrees of all vertices is even. Euler path exists if and only if the number of vertices with odd degree is equal to two (or zero in the case of the existence of an Euler tour).

In addition, of course, the count must be sufficiently connected (ie, if you remove from it all isolated vertices, then should get a connected graph).

Search all cycles and will combine them one recursive procedure:

```
procedure FindEulerPath (V)
    1. iterate over all edges emanating from the vertex V;
       each edge is removed from the graph, and
       FindEulerPath call from the second end of this edge;
    2. add the vertex V in response.
```

The complexity of this algorithm is obviously linear in the number of edges.

But the same algorithm, we can write in **a non-recursive** version:

```
stack St;
in St we place any vertex (the starting vertex);
St is not empty yet
let V - value at the top of St;
if the degree (V) = 0, then
    add V to the answer;
    V remove from the top of St;
otherwise
    find any edge that goes from V;
    remove it from the graph;
    the other end of the rib put in a St;
```

It is easy to verify the equivalence of these two forms of the algorithm. However, the second shape, obviously, is faster, with a code will not anymore.

## The problem of dominoes

We give here the classical problem for Euler cycle - the problem of dominoes.

There are N dominoes are known at two ends domino recorded one number (typically 1 to 6, but in this case is not important). Required to lay out all domino in a row so that any two adjacent dominoes number recorded on their common side coincide. Dominoes allowed to turn.

Reformulate the problem. Suppose that the numbers recorded on donimoshkah - vertices and domino - edges of the graph (each domino with numbers (a, b) - this edge (a, b), and (b, a)). Then our task is **reduced to** the problem of finding **an Euler path** in this graph.

## Implementation

The following program searches for and displays the Eulerian path or a path in the graph, or displays -1 if it does not exist.

First, the program checks the degrees of the vertices if the vertices with odd degree is not present, then the graph has an Euler cycle, if there are 2 vertices with odd degree, then the graph has an Euler path only (Euler cycle is not), and if such heights greater than 2, then Box no Euler cycle or Euler path. To find an Euler path (no cycle), we proceed as follows: if V1 and V2 - this two vertices of odd degree, then just add the edge (V1, V2), the resulting graph will find an Euler cycle (he obviously will exist), and then remove from the response of "fictitious" edge (V1, V2). Eulerian path will look exactly as described above (non-recursive version), and at the same time at the end of this algorithm will check was connected graph or not (if the graph was not connected, then at the end of the algorithm in the graph remain some ribs, and in this case we need to bring -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));
    ... Read 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);

}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:15  
EDIT: 10 Jun 2008 22:16

## Checking on the graph is acyclic and finding cycle

Let a directed or undirected graph without loops and multiple edges. Required to check whether it is acyclic, and if not, then search for any cycle.

We solve this problem by using [depth-first search](#) for the O (M).

### Algorithm

Proizvedëm series of depth-first search in a graph. Ie from each vertex, which we have never come, you will find in depth, which is at the entrance to the top will paint it gray, and at the outlet - in black. And if attempts dfs to a gray top, it means that we found a loop (if undirected graph, cases where dfs from some vertex in trying to ancestor is not considered).

The cycle can be restored array walking ancestors.

### Implementation

Here is an implementation for the case of a directed graph.

```
int n;
vector <vector <int>> G;
vector <char> C1;
vector <int> P;
int cycle_st, cycle_end;

bool dfs (int v) {
    C1 [v] = 1;
    for (size_t i = 0; i < G [v] .size (); i ++) {
        int to = G [v] [i];
        if (C1 [to] == 0) {
            P [to] = v;
            if (dfs (to )) Return True;
        }
        else if (C1 [to] == 1) {
            v = cycle_end;
            cycle_st = to;
            Return True;
        }
    }
}
```

### Contents [\[hide\]](#)

- Checking on the graph is acyclic and finding cycle
  - Algorithm
  - Implementation

```
        }
    }
    Cl [v] = 2;
    Return false;
}

int main () {
    ... Read Count ...

    p.assign (n, -1);
    cl.assign (n, 0);
    cycle_st = -1;
    for (int i = 0; i <n; i++)
        if (dfs (i) )
            Break;

    if (-1 == cycle_st)
        puts ("Acyclic");
    else {
        puts ("Cyclic");
        vector <int> cycle;
        cycle.push_back (cycle_st);
        for (int v = cycle_end; v! = cycle_st; P v = [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);
    }
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:30  
EDIT: 31 Dec 2011 1:58

## Lowest common ancestor. Finding of O ( $\sqrt{N}$ ) and O ( $\log N$ ) with preprocessing O (N)

Suppose we are given a tree G. The input received inquiries species (V1, V2), for each query you want to find their lowest common ancestor, ie vertex V, which lies in the path from the root to V1, the path from the root to V2, and all of these peaks should be chosen lowermost. In other words, the required vertex V - ancestor and V1, and V2, and among all of these 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. Specifically, for example, if V1 is the ancestor V2, V1 is then their lowest common ancestor.

In English, this problem is called the problem LCA - Least Common Ancestor.

### The idea of the algorithm

Before responding, perform the so-called **preprocessing**. Start the tour in depth of the root, which will build a list of visited vertices Order (current vertex is added to the list when entering the top, 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 the vertex, ie, Order [First [I]] = I for all I. Also, using depth-first search will find the height of each vertex (the distance from the root to it) - H [1..N].

As it is now responding? Let there be a current request - a pair of vertices V1 and V2. Consider the list of Order between the 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 vertices in that it will be the top of the lowest height.

So, to answer your inquiry, we just need to **find the top of the lowest height** in the array Order in the range between First [V1] and First [V2]. Thus, **the problem reduces to the LCA problem RMQ** ("minimum interval"). A final problem is solved with the help of data structures (see. [RMQ problem](#) ).

If you use the **sqrt-decomposition**, we can obtain a solution to meet the request of the **O ( $\sqrt{N}$ )** and performs preprocessing for the **O (N)**.

If you use **wood pieces**, you can get the solution corresponding to a request for **O ( $\log (N)$ )** and performs preprocessing for the **O (N)**.

### Implementation

There will be given implementation of LCA finished with 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)
{
```

### Contents [hide]

- Lowest common ancestor. Finding of O ( $\sqrt{N}$ ) and O ( $\log N$ ) with preprocessing O (N)
  - The idea of the algorithm
  - Implementation

```

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, s1, 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;
    ... Read Count ...

    lca_prepare (g, root);

    for (;;)
    {
        int v1, v2; // Received a request
        int v = lca (v1, v2); // Response to a request
    }
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:48  
Edited: 14 Mar 2012 2:09

## Lowest common ancestor. Finding of O (log N) (Binary lifting method)

### Contents [hide]

- Lowest common ancestor. Finding of O (log N) (Binary lifting method)
  - Algorithm
  - Implementation

Suppose we are given a tree G. The input received inquiries species (V1, V2), for each query you want to find their lowest common ancestor, ie vertex V, which lies in the path from the root to V1, the path from the root to V2, and all of these peaks should be chosen lowermost. In other words, the required vertex V - ancestor and V1, and V2, and among all of these 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. Specifically, for example, if V1 is the ancestor V2, V1 is then their lowest common ancestor.

In English, this problem is called the problem LCA - Least Common Ancestor.

Here will be considered an algorithm which is written much faster than the one described [here](#).

Asymptotic behavior of the resulting algorithm will be: preprocessing of **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, that is P [i] [j] - is  $2^j$ -th ancestor of vertex i,  $i = 1..N, j = 0..[\log N]$ . Also, for each vertex we find the time to call her and output depth-first search (see. "[Search in depth](#)") - this we will need to determine in **O (1)** whether one vertex ancestor of another (not necessarily direct). Such preprocessing can be done in **O (N log N)**.

Suppose now received another request - a pair of vertices (A, B). Immediately check, is not one vertex ancestor of the other - in this case, it is the result. If A is an ancestor of B, and B is not an ancestor of A, then we will climb the ancestors of A, until we find the highest (ie, closest to the root) summit, which is not an ancestor of (not necessarily direct) B (t. e. a vertex X, such that X is not an ancestor of B, and P [X] [0] - the ancestor of B). In this case, find the vertex of X is **O (log N)**, using an array of P.

We describe this process in more detail. Let  $L = \lceil \log N \rceil$ . First, let  $I = L$ . If  $P[A][I]$  is not an ancestor of  $B$ , then assign  $A = P[A][I]$ , and reduce  $I$ . If  $P[A][I]$  is the ancestor of  $B$ , then just reduce  $I$ . Obviously, when  $I$  would be less than zero, the vertex  $A$  just and will be required top - ie such that  $A$  is an ancestor of  $B$ , but  $P[A][0]$  - the 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  $B$ .

Asymptotic behavior. The whole algorithm query response comprises change of  $I$   $L = \lceil \log N \rceil$  to 0, and checks at every step in  $O(1)$  whether the ancestor of the other one vertex. Consequently, for each query will get the answer for  $O(\log N)$ .

## 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 () {

```

```
... Read n and g ...

tin.resize (n), tout.resize (n), up.resize (n);
l = 1;
while ((1 << l) <= n) ++ l;
for (int i = 0; i <n; ++ i) up [i] .resize (l + 1);
dfs (0);

for (;;) {
    int a, b; // The current query
    int res = lca (a, b); // Response to a request
}

}
```

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 24 Aug 2008 9:56  
EDIT: 30 Mar 2012 20:51

## Lowest common ancestor. Finding the O (1) preprocessing with O (N) (algorithm Farah-Colton and Bender)

Suppose we are given a tree G. The input received inquiries species (V1, V2), for each query you want to find their lowest common ancestor, ie vertex V, which lies in the path from the root to V1, the path from the root to V2, and all of these peaks should be chosen lowermost. In other words, the required vertex V - ancestor and V1, and V2, and among all of these 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. Specifically, for example, if V1 is the ancestor V2, V1 is then their lowest common ancestor.

In English, this problem is called the problem 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, e.g., gate-Vishkina).

### Algorithm

We use the classical LCA reducing the problem to the problem RMQ (at least on the segment) (for more details see. [Lowest common ancestor. Finding of O \(sqrt \(N\)\) and O \(log N\) with preprocessing O \(N\)](#)). Now learn how to solve the problem RMQ 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 (since the elements of the array - is nothing more than the height of the 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 a solution to this problem RMQ.

Let A be the array over which requests are made RMQ, and N - the size of the array.

We first construct an algorithm that solves this problem with preprocessing O (N log N) and O (1) on request . This is easy to do: create a so-called Sparse Table T [l, i], where each element T [l, i] is equal to the minimum in the interval A [l; l + 2<sup>i</sup> ]. Obviously, 0 <= i <= ⌈ log N ⌉, and therefore the size of Sparse Table is O (N log N). It is also easy to build for the O (N log N), if we observe that T [l, i] = min (T [l, i-1], T [l + 2<sup>i-1</sup>, i-1]). How now to respond to every request RMQ O (1)? Let received a request (l, r), then the answer is min (T [l, sz], T [r-2<sup>sz</sup> + 1, sz]), where sz - the largest power of two that do not exceed r-l + 1. Indeed, we seem to take the segment (l, r) and cover it with two runs of length 2<sup>sz</sup> - 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 asymptotic behavior of the O (1) on request, we must predposchitat sz value for all possible lengths from 1 to N.

We will now describe how to improve this algorithm to the asymptotic behavior of O (N).

We divide the array A into blocks of size K = 0.5 log<sub>2</sub> N. For each block, calculate the minimum element in it and its 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. We construct the array B Sparse Table, as described above, the size Sparse Table and time of its construction will be equal to:

$$\begin{aligned} N / K \log N / K &= (2N / \log N) \log (2N / \log N) = \\ &= (2N / \log N) (1 + \log (N / \log N)) \leq 2N / \log N + 2N = O (N) \end{aligned}$$

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 in different blocks, the answer will be a minimum of the following values: the minimum block l, starting from l to the end of the block, then the minimum units l and after a to r (non-inclusive), and finally the minimum block r, from the beginning of the block to r. At the request "at least in the" we can already meet in O (1) with Sparse Table, there were only requests RMQ in blocks.

Here we use "+ -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 equal to:

$$2^{K-1} = 2^{0.5 \log N - 1} = 0.5 \sqrt{N} = O (\sqrt{N})$$

Thus, the number of different blocks is O (sqrt (N)), and therefore we can predposchitat results RMQ within all the various blocks of O (sqrt (N) K<sup>2</sup>) = O (sqrt (N) log<sup>2</sup> N) = O (N). In terms of implementation, we can characterize each block bit mask length of the K-1 (which obviously fits in standard type int), and store in a predposchitannye RMQ array R [mask, l, r] size O (sqrt (N) log<sup>2</sup> N).

So we learned predposchityvat RMQ results within each block, as well as RMQ over by blocks, all for a total of O (N), and to respond to each request RMQ O (1) - using only the precomputed values, in the worst case four: in Block l, at block r, and blocks between l and r are not inclusive.

### Implementation

At the beginning of the program indicated 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;
```

### Contents [hide]

- Lowest common ancestor. Finding the O (1) preprocessing with O (N) (algorithm Farah-Colton and Bender)
  - Algorithm
  - Implementation

```

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 [SQRRT_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] = 1;
            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];
}

```

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 24 Aug 2008 11:26  
EDIT: 9 Sep 2008 15:52

## The task RMQ (Range Minimum Query - at least in the interval). Solution in O (1) preprocessing O (N)

Given an array A [1..N]. Receives requests like (L, R), for each query is required to find the minimum of the array A, starting from a position of L and ending at R. array A change in the process can not, that is, here described solution of the static problem RMQ.

Here's asymptoticheski 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 problem RMQ LCA, and then uses [an algorithm Farah-Colton and Bender](#), which reduces the problem back to the LCA RMQ (but a particular form) and solves her.

### Algorithm

We 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 Treap priorities ordered from smaller to larger in the root). Such a tree can be constructed in O (N). A request RMQ (l, r) is equivalent to the request LCA (l', r'), where l' - vertex corresponding to the element A [l], r' - corresponding to A [r]. Indeed, LCA find a vertex which is keyed between l' and r', i.e. by position in the array A is between l and r, and wherein the vertex closest to the root, i.e. with the lowest priority, i.e. the lowest value.

LCA problem we can solve in O (1) preprocessing O (N) using the [algorithm Farah-Colton and Bender](#), who, interestingly, LCA reduces the problem back to task RMQ, but a particular form.

### Contents [hide]

- The task RMQ (Range Minimum Query - at least in the interval).
  - Solution in O (1) preprocessing O (N)
    - Algorithm

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 2 Mar 2009 17:45  
EDIT: 2 Mar 2009 17:45

## Lowest common ancestor. Finding for $O(1)$ the offline (Tarjan algorithm)

### Contents [hide]

- Lowest common ancestor. Finding for  $O(1)$ the offline (Tarjan algorithm)
  - Tarjan algorithm
  - Implementation

Given a tree  $G$ with  $n$ vertices and given  $m$  the type of queries  $(a_i, b_i)$ . For each query  $(a_i, b_i)$ you want to find the lowest common ancestor of vertices  $a_i$ and  $b_i$ , ie a vertex  $c_i$ , which is furthest from the root, and thus is the ancestor of both peaks  $a_i$ and  $b_i$ .

We consider the problem of off-line, ie Considering that all requests are known in advance. Algorithm described below allows you to answer all  $m$ requests for the total time  $O(n + m)$ , ie sufficiently large  $m$ for  $O(1)$ the request.

## Tarjan algorithm

The basis for the algorithm is the data structure "system of disjoint sets" , which was invented Tarjanne (Tarjan).

The algorithm is actually a detour into the depths of the root of the tree, in the which are gradually responding to requests. Namely, the response to the request  $(v, u)$ is, when the depth is in the bypass at the vertex  $v$ and the vertex  $u$ has already been visited, or vice versa.

So let bypass in depth is at the top  $v$ (and have already been performed transitions in her sons), and it turned out that for some query  $(v, u)$ vertex  $u$ already has had a bypass in depth. Then learn how to find LCAthese two vertices.

Note that  $\text{LCA}(v, u)$ is either very top  $v$ , or one of its ancestors. So, we need to find the top of the lowest among the ancestors  $v$ (including her herself), for which the vertex  $u$ is a descendant. Note that for fixed  $v$ on such grounds (ie, what the lowest ancestor  $v$ is the ancestor of some top) top of the tree the tree split into a set of disjoint classes. For each ancestor  $p \neq v$ tops  $v$ its class contains the very top of this, as well as all subtrees with roots in those of her sons who are "left" on the way up  $v$ (ie, that have been treated earlier than has been achieved  $v$ ).

We need to learn how to effectively support all of these classes, for which we apply data structure "system of disjoint sets." Each class will meet in this structure are many, with the representative for this set we define the quantity ANCESTOR- the top of  $p$ which form this class.

A detailed discussion of the implementation of bypass in depth. Suppose we are in a vertex  $v$ . Put it in a separate class in the structure of disjoint sets  $\text{ANCESTOR}[v] = v$ . As usual crawled deep fingering all outgoing edges  $(v, to)$ . For each of these  $to$ , we first

need to call the tour into the depths of this summit, and then add it to the top with all its subtree in the top of the class  $v$ . This operation is implemented Union data structure "a system of disjoint sets", followed by the installation  $\text{ANCESTOR} = v$  for a representative set (as representative of the class after the merger could change). Finally, after processing all the edges we iterate through all kinds of questions  $(v, u)$ , and if  $u$  has been marked as visited bypass in depth, the answer to this query is the vertex  $\text{LCA}(v, u) = \text{ANCESTOR}[\text{FindSet}(u)]$ . It is easy to see that for each query, this condition (which one vertex is the current request, and the other has had before) is executed exactly once.

We estimate the **asymptotic behavior**. It consists of several parts. Firstly, it is the asymptotic behavior of bypass in depth, which in this case is  $O(n)$ . Secondly, this business combination sets, which together all reasonable  $n$  spend  $O(n)$  operations. Third, it is for each request verification of (twice on request) and an outcome (once on request) each, again, for all sentient  $n$  done for  $O(1)$ . The resulting asymptotic behavior is obtained  $O(n + m)$ , which means that for sufficiently large  $m$  ( $n = O(m)$ ) response for  $O(1)$  a single request.

## Implementation

We give full implementation of the algorithm, including a slightly modified (supporting **ANCESTOR**) the implementation of a system of intersecting sets (randomized version).

```

const int MAXN = максимальное число вершин в графе;
vector<int> g[MAXN], q[MAXN]; // граф и все запросы
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() {
    ... чтение графа ...

    // чтение запросов
    for (;;) {
        int a, b = ...; // очередной запрос
        --a, --b;
        q[a].push_back (b);
        q[b].push_back (a);
    }

    // обход в глубину и ответ на запросы
    dfs (0);
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:50  
EDIT: 2 May 2012 0:47

## Maximum flow by Edmonds-Karp for O (NM<sup>2</sup>)

**Warning:** This article is outdated and contains errors and is not recommended for reading. After some time, the article will be completely rewritten.

Suppose we are given a graph G, which highlights two peaks: the source S and drain T, and each rib 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, if we consider the graph as a network of tubes with some bandwidth. Ie stream - the function  $F_{u,v}$ , defined on the set of edges of the graph.

The challenge is to find the maximum flow. Here we consider the method of Edmonds-Karp, who works for the O(NM<sup>2</sup>), or (another estimate) O(FM), where F - value of the desired flow. The algorithm was proposed in 1972.

### Algorithm

**Residual bandwidth** is called bandwidth rib net current flow along this edge. It must be remembered that if a stream flows through the oriented edge, there is a so-called reverse edge (directed in the opposite direction), which will have zero capacity, and which will take place the same value on stream, but with a minus sign. If the edge has been oriented, it breaks up into two equally oriented edge bandwidth, and each of these edges is reversed for the other (if one proceeds flow F, then proceeds differently -F).

The general scheme of the algorithm **Edmonds-Karp** is. First, assume the flow to zero. Then look for complementary way, ie simple path from S to T at the edges whose residual capacity is strictly positive. If supplementing the path was found, then produced an increase in current flow along this path. If the same path is not found, the current flow is maximized. To find complementary ways can be used as a [bypass to the width and depth of a bypass](#).

Consider a more accurate procedure to increase the flow. Suppose we found some complementary way, then let C - the smallest of the residual capacity of edges of this path. Procedure to increase the flow is as follows: for each edge (u, v) fulfill complementary ways:  $F_{u,v} = +C$ , and  $F_{v,u} = -F_{u,v}$  (or, equivalently,  $F_{v,u} = C$ ).

The quantity of flow is the sum of all non-negative values of  $F_{S,v}$ , where v - any vertex joined to the source.

### Implementation

#### Contents [hide]

- Maximum flow by Edmonds-Karp for O (NM<sup>2</sup>)
  - Algorithm
  - 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, the stock - the 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;

}
```

# MAXimal

home  
something  
bookz  
forum  
about

Added: 10 Jun 2008 22:52  
EDIT: 10 Jun 2008 22:53

## Maximum flow by pushing predpotoka for O (N<sup>4</sup>)

### Contents [hide]

- Maximum flow by pushing predpotoka for O (N<sup>4</sup>)
  - Algorithm
  - Implementation

Suppose we are given a graph G, which highlights two peaks: the source S and drain T, and each rib 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, if we consider the graph as a network of tubes with some bandwidth. Ie stream - the function  $F_{u,v}$ , defined on the set of edges of the graph.

The challenge is to find the maximum flow. Here we consider a method of pushing predpotoka working for O (N<sup>4</sup>), or, more precisely, for the O (N<sup>2</sup> M). The algorithm was proposed by Goldberg in 1985.

### Algorithm

The general scheme is the algorithm. At each step, we will consider some predpotok - ie feature that allows flow properties similar, but not necessarily satisfy the law of conservation of flow. At each step, we try to apply any of the two operations: pushing or lifting the top of the stream. If at some stage it will become impossible to apply any of the two operations, we have found that the desired flow.

For each vertex defined its height  $H_u$ ,  $H_A = N$ ,  $H_T = 0$ , and for any residual edges  $(u, v)$ , we have  $H_u \leq H_v + 1$ .

For each node (except S) it is possible to determine the excess:  $E_u = F_{v,u}$ . Top with positive excess is called overflow.

Operation push Push  $(u, v)$  is available when the vertex u is full, the residual capacity of the  $C_{f_{u,v}} > 0$  and  $H_u \leq H_v + 1$ . Operation push is to maximize the flow from u to v, limited excess  $E_u$  and residual capacity  $C_{f_{u,v}}$ .

Operation lift Lift  $(u)$  raises crowded vertex u to the maximum height. Ie  $H_u = 1 + \min\{H_v\}$ , where  $(u, v)$  - residual rib.

It remains only to consider the initialization flow. Only necessary to initialize the following values:  $F_{S,v} = C_{S,v}$ ,  $F_{u,S} = -C_{u,S}$ , the remaining values are set 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 [in], c [u] [v] - f [u] [v]);
    f [u] [v] += d;
    f [v] [u] = - f [u] [v];
    e [in] -= 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, twenty (n));
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            cin >> c[i][j];
    // Source - vertex 0, the stock - the top n-1

    vvint f (n, twenty (n));
    for (int i=1; i<n; i++)
    {
        f[0][i] = c[0][i];
        f[i][0] = -c[0][i];
    }
}

```

```

}

Twenty h (n);
h[0] = n;

and twenty (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);

}

```

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 10 Jun 2008 22:53  
EDIT: 14 Oct 2011 0:31

## Modification of the method of pushing predpotoka for finding the maximum flow of $O(N^3)$

It is assumed that you have already read [predpotoka push method for finding the maximum flow of  \$O\(N^4\)\$](#) .

### Contents [hide]

- Modification of the method of pushing predpotoka for finding the maximum flow of  $O(N^3)$ 
  - Description
  - Implementation

### Description

Modification is very simple: at each iteration among all crowded vertices we select only those vertices that have **the Greatest height**, and use push / lift only to those heights. Furthermore, to select a maximum height of vertices we do not need any data structure, simply store a list of nodes with the greatest height and recalculate it only when all the vertices in the list have been processed (then be added to list is already at the top with height), and when a new vertex 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 procedure. To be exact, asymptotic received algorithm is  $O(NM + N^2 \sqrt{M})$ , in the worst case is  $O(N^3)$ .

This modification was proposed Cherianom (Cherian) and Maheshwari (Maheshvari) in 1989

### Implementation

Here is a ready implementation of this algorithm.

Unlike conventional algorithm push - only in the presence of an array maxh, which will be kept rooms crowded peaks with a maximum height. The size of the array 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 vertices); if it is then the array is still empty, the crowded peaks not, and the algorithm stops. Otherwise, we go over the tops of the list, applying to them pushing or lifting. After the operation of pushing the current vertex may cease to be crowded, in this case, remove it from the list maxh. If, after some operations of raising the height of the current node becomes greater than the height of vertices in the list maxh, then we clear the list ( $sz = 0$ ), and immediately proceed to the next iteration of the algorithm push (which will be built a 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 stream f ...
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:54  
EDIT: 10 Jun 2008 22:55

## Finding the flow in the graph in which each rib specified minimum and maximum flow

### Contents [hide]

- Finding the flow in the graph in which each rib specified minimum and maximum flow
  - Solution of Problem 1
  - Solution of Problem 2

Suppose we are given a graph  $G$ , where for each edge in addition to the capacity (maximum flow along this edge) is specified and the minimum value of the flow, which should take place on the edge.

Here we consider two problems: 1) the need to find an arbitrary stream that satisfies all the constraints, and 2) you want to find the minimum flow that satisfies all the constraints.

### Solution of Problem 1

Denote by  $L_i$  minimum value stream which may pass through the  $i$ -th rib, and by  $R_i$  - its maximum value.

Proizvedëm in the following graph **changes**. Add a new source  $S$  'and runoff  $T$ '. Consider all edges that have  $L_i$  is different from zero. Let  $i$  - number of such edges. Let the ends of the rib (oriented) - this is the top of the  $A_i$  and  $B_i$ . Add Edge  $(S', B_i)$ , in which  $L = 0$ ,  $L = R_i$ , add an edge  $(A_i, T')$ , in which  $L = 0$ ,  $L = R_i$ , and at the very edges of the  $i$ -th set  $R_i = R_i - L_i$ ,  $L_i = 0$ . Finally, add a graph edge from  $T$  to  $S$  (old drain and source) in which  $L = 0$ ,  $R = INF$ .

After performing these transformations all edges of the graph will have  $L_i = 0$ ; we have reduced this problem to the usual problem of finding the maximum flow (but in a modified box with the new source and drain) (in order to understand why the maximum - read the following explanation).

The correctness of these transformations more difficult to understand. Informal **explanation** is. Each edge which  $L_i$  is different from zero, we replace the two edges, one with a capacity of  $L_i$ , and the other - with  $R_i - L_i$ . We need to find a thread that would necessarily saturated the first edge of the pair (ie, flow along this edge should be equal to  $L_i$ ); second edge we care less - flow along it can be anything, as long as it does not exceed its capacity. So, we need to find a thread that would definitely satiated some set of edges. Let us consider each such edge, and perform such an operation: to sum to the end edge of a new source  $S'$ , to sum edge from its beginning to the drain of  $T'$ , itself an edge to delete, and runoff

from the old to the old source of the T S will hold an edge infinite bandwidth. By these actions, we proimitiruem that this edge is saturated - the rib will flow  $L_i$  flow units (we simulate it with a new source, which feeds on the right end of the rib flow amount), and will flow into it again  $L_i$  flow units (but instead of ribs this thread gets a new sink). Flow from the new source flows through one portion of the graph until the old dotekaet Photo T, it flows from the source to the old S, then flows through another portion of the graph, and finally arrives at the beginning of this edge and into a new stock T'. That is, if we find in this modified maximum flow graph (and the drain gets the right amount of flow, ie the sum of all values of  $L_i$  - otherwise the flow rate will be lower, and the answer simply does not exist), we simultaneously find flow in the original graph, which will satisfy all constraints minimum, and, of course, all the constraints of the maximum.

## Solution of Problem 2

Note that on the edge of an old run-old source with a capacity of INF flows all the old stream, ie, bandwidth of the edge influences the value of the old stream. For sufficiently large values of the capacity of this edge (ie INF) old flow is unrestricted. If we reduce the bandwidth, then, after a certain time, and will decrease the value of the old stream. But too small value of the flow rate will be insufficient to ensure that the constraints (to the minimum flow along the edges). Obviously, you can use **binary search on the value of INF**, and find it is the smallest value for which all constraints are satisfied yet, but the old thread will have a minimum value.

---

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 10 Jun 2008 22:55  
EDIT: 13 Aug 2009 23:03

## The flow of minimum cost (min-cost-flow). Algorithm increasing ways

Given network G, consisting of N vertices and M edges. Each edge (generally oriented, but in this connection see. Below) contains the capacity of the (non-negative integer), and the cost per unit of flow along this edge (an integer). Column Set the source S and drain T. gives some K value stream, find the flow of this magnitude, with among all flows of this magnitude to choose the least-cost flow ("task min-cost-flow").

Sometimes the task of putting a little bit different: it is required to find the maximum flow of the lowest value ("task min-cost-max-flow").

Both of these problems are solved effectively increase the algorithm described below ways.

### Description

The algorithm is very similar to the [Edmonds-Karp algorithm](#) for calculating the maximum flow .

#### The simplest case

Consider to begin with the simplest case when 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 ribs  $(i, j)$ , if there is an edge. Let  $C_{ij}$  - the cost per unit of flow along the edge  $(i, j)$ . Let  $F_{ij}$  - the value of the flow along the edges  $(i, j)$ , initially all the fluxes are zero.

**Modify the** network as follows: for each edge  $(i, j)$  added to the network so-called **reverse** edge  $(j, i)$  with a capacity of  $U_{ji} = 0$  and the value of  $C_{ji} = -C_{ij}$ . Because, by assumption, the edges  $(j, i)$  to this network was not modified in such a way that the network is still not multigraph. In addition, all along the algorithm will support true condition:  $F_{ji} = -F_{ij}$  .

Define **residual network** for a certain fixed flow F as follows (in fact, just as in the Ford-Fulkerson algorithm) network owned by only the residual unsaturated ribs (i.e. in which  $F_{ij} < U_{ij}$ ), and a residual bandwidth of each such as ribs  $UPI_{ij} = U_{ij} - F_{ij}$  .

Actually **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 value of  $C_{ij}$ ). If the path is not found, then the algorithm terminates, the flow F - sought. If the path has been found, we are increasing the flow along it as much as possible (ie, pass along this path, we find minimal residual capacity  $MIN\_UPI$  among the edges of the 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 point the flux has reached the K (given to us by the condition of the flux), we will also stop the algorithm (it should be noted that while in the last iteration of the algorithm by increasing the flow along the path you need to increase the flow to a value such that the final flow not exceeded K, but this is easily done).

Easy to see that if we put K equal to infinity, then the algorithm finds the maximum flow of minimum cost, ie the same algorithm without modification solves both problems min-cost-flow and min-cost-max-flow.

#### The case of undirected graphs, multigraphs

The case of undirected graphs and multigraphs conceptually no different from the above, therefore, the actual algorithm will work on these graphs. However, there are some difficulties in the implementation, which should be addressed.

**Undirected** edge  $(i, j)$  - is actually two oriented edges  $(i, j)$  and  $(j, i)$  with the same bandwidth and cost. Since

#### Contents [hide]

- The flow of minimum cost (min-cost-flow). Algorithm increasing ways
  - Description
    - The simplest case
    - The case of undirected graphs, multigraphs
    - Operating time analysis
  - Implementation

the above algorithm min-cost-flow is required for each undirected edges to create the opposite edge to him, the result is that the non-oriented edge is split into 4 oriented ribs, and we actually get the case **multigraph** .

What problems cause **multiple ribs** ? Firstly, the flow rate of each of multiple edges must be maintained separately. Second, when looking for the shortest path, be aware that it is important what kind of select multiple edges when you restore the path to the ancestors. Ie instead of the usual array of ancestors for each vertex, we should keep the top of the parent and the number of edges on which we have come out of it. Thirdly, when the flow along the edge of a need according to an algorithm to reduce the reverse flow along the rib. Since we can have multiple edges, you'll need to keep the number of each edge ribs, reverse it.

Another difficulty with undirected graphs and multigraphs not.

## Operating time analysis

By analogy with the analysis algorithm Edmonds-Karp, we get 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 [the simplest version of Dijkstra's algorithm](#) , the entire algorithm min-cost-flow will estimate  $O(N^3M)$  , however, Dijkstra's algorithm will have to modify it to work on graphs with negative weights (called Dijkstra's algorithm with potentials ).

Instead, you can use the [algorithm of Leviticus](#) , which, although asymptotically much worse, but in practice it works very quickly (in about the same time as the Dijkstra's algorithm).

## Implementation

Here is an implementation of the algorithm min-cost-flow, based on the [algorithm of Leviticus](#) .

The input to the algorithm is supplied network (undirected multigraph) with  $N$  vertices and  $M$  edges, and  $K$  - value stream you want to search. The algorithm finds the minimum flow value of  $K$  value, if one exists. Otherwise, he finds the maximum value of the flow of minimum cost.

The program has a special feature to add a directed edge. If you need to add a non-oriented edge, then this function should be called for each edge  $(i, j)$  twice by  $(i, j)$  and of  $(j, i)$ .

```
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;
    ... Read Count ...

    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 [rb] == 0) {
                q [qt ++] = rb;
                if (qt == n) qt = 0;
            }
            else if (id [rb] == 2) {
                if (--qh == -1) qh = n-1;
                q [qh] = rb;
            }
            id [rb] = 1;
            p [rb] = v;
            p_rib [rb] = 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 ...

}

```

# MAXimal

home  
something  
bookz  
forum  
about

Added: 10 Jun 2008 22:58  
EDIT: 8 Sep 2011 17:05

## Assignment problem. Solution with a min-cost-flow

The problem has two equivalent statement:

- Valued square matrix A [1..N, 1..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 are N and N orders of machine tools. About every order known cost of manufacture on 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 [of finding the minimum cost flow \(min-cost-Flow\)](#), solving the problem of the nominations for the **O (N<sup>5</sup>)**.

### Description

We construct a bipartite network: there is a source S, drain T, in the first part are N vertices (corresponding to the rows of the matrix or orders), the second - the same N vertices (corresponding to the columns of the matrix or the machine). 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<sub>ij</sub>. From the source S will hold the edges to all vertices i of the first part with a capacity of 1 and 0. The value from each vertex of the second part to the drain of T j will hold an edge with capacity 1 and the value 0.

We find in the resulting network maximum flow of minimum cost. Obviously, the flow rate will be equal to N. Further, it is clear that for each vertex i of the first part there exists exactly one vertex j of the second part, such that the flow F<sub>ij</sub> = 1. Finally, obviously this one-to-one correspondence between the vertices of the first part and the vertices of the second part is a solution of the problem (as found by the flow has a minimum value, the sum of the costs of selected edges will be the least possible, that is the criterion of optimality).

Asymptotics of the solution of the assignment problem depends on how the algorithm searches for the maximum flow of minimum cost. Asymptotics be **O (N<sup>3</sup>)** by using Dijkstra's algorithm or **O (N<sup>4</sup>)** using Bellman-Ford algorithm.

### Implementation

Listed here realization of long, perhaps it can be reduced significantly.

```

typedef vector<int> vint;
typedef vector <Vint> vvint;

const int INF = 1000*1000*1000;

int main()
{
    int n;
    vvint in (n, twenty (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);
        and <string> q;
        dist[s] = 0;
        p[s] = -1;
        type[s] = 1;
        q.push_back (s);
        for ( ; !q.empty(); )
        {
            int v = q.front(); q.pop_front();
            ...
            for (int i = 0; i < m; i++)
                if (f[i][v] < INF)
                    if (dist[i] >= f[i][v])
                        dist[i] = f[i][v];
                    else
                        cost += f[i][v];
                    if (p[i] == -1)
                        p[i] = v;
                    type[i] = 2;
        }
    }
}

```

### Contents [hide]

- Assignment problem. Solution with a min-cost-flow
  - Description
  - Implementation

```

type[v] = 0;
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 t = increase of enhancement! = - 1; p = increase [taking])
{
    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);
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 22:59  
EDIT: 23 Aug 2012 18:00

## Hungarian algorithm for solving the assignment problem

### Statement of the assignment problem

The assignment problem is placed quite naturally.

Here are a few **options for setting** (easy to see that they are all equivalent to each other):

- There are  $n$  workers and  $n$  jobs. For every worker knows how much money it will ask for the fulfillment of a task. Each worker can take him only one job. Required to distribute tasks to work so as to minimize the total cost.
- Given matrix  $a$  size  $n \times n$ . Requires at each of its row select one number, so that in any column was also selected exactly one number, while the amount of selected numbers would be minimal.
- Given matrix  $a$  size  $n \times n$ . Required to find a permutation of  $p$  length  $n$  that the value  $\sum a[i][p[i]]$  is minimal.
- Given the complete bipartite graph with  $n$  vertices; each edge is assigned some weight. You want to find a perfect matching of minimum weight.

Note that all the above cited setting " **SQUARE** ": they both dimensions are always the same (and equal  $n$ ). In practice, there are often similar to " **rectangular** " performances, when  $n \neq m$ , and we must choose the  $\min(n, m)$  elements. However, as it is easy to see from the "square" of the problem you can always go to the "square", adding rows / columns with zero / infinite values, respectively.

Also note that, by analogy with the search for **the minimum** solution is also possible to put the problem of finding **the maximal** solution. However, these two problems are equivalent to each other: just multiply all the weights  $-1$ .

### Contents [hide]

- Hungarian algorithm for solving the assignment problem
  - Statement of the assignment problem
  - Hungarian algorithm
    - Historical Background
    - Construction of an algorithm for  $O(n^4)$
    - Construction of an algorithm for  $O(n^3)O(n^2m)$
    - )
    - The implementation of the Hungarian algorithm for  $O(n^3)O(n^2m)$
  - Examples of tasks
  - Literature
  - Tasks in the online judges

## Hungarian algorithm

### Historical Background

The algorithm was developed and published by Harold **Kuhn** (Harold Kuhn) in 1955, Kuhn himself gave an algorithm called "Hungarian", because it was largely based on the earlier work of two Hungarian mathematicians: Dénes **König** (Dénes König) and Eigen **Egerváry** ( Jenő Egerváry).

In 1957, James **Munkres** (James Munkres) showed that this algorithm works for (strongly) polynomial time (ie the time of the order of the polynomial  $n$  that does not depend on the magnitude of the price).

Therefore, in the literature, this algorithm is known not only as the "Hungarian" but also as "Kuhn-Munkres algorithm" or "Munkres algorithm."

However, recently (in 2006) it became clear that the same algorithm was invented **a century before Kuhn** 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 is that the original algorithm Kuhn had the asymptotic behavior  $O(n^4)$ , and only later Jack **Edmonds** (Jack Edmonds) and Richard **Karp** (Richard Karp) (and independently of them **Tomidzava** (Tomizawa)) showed how to improve it to the asymptotic behavior  $O(n^3)$ .

## Construction of an algorithm for $O(n^4)$

Just note in order to avoid ambiguities, we mainly consider here the problem of appointments in the matrix formulation (ie, given a matrix  $a$ , and it is necessary to choose from  $n$  cells located in different rows and columns). Array indexing, we start with the unit, 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 **are non-negative** (if it is not, you can always go to a non-negative matrix, adding to all numbers a number).  $a[]$

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 you can see, the numbers  $u[i]$  correspond to strings, and numbers  $v[j]$  - the columns of the matrix.)

We call **the value of the potential** amount of its numbers:

$$f = \sum_{i=1}^n u[i] + \sum_{i=1}^n v[i].$$

On the one hand, it is easy to notice that the value of the desired solution **is not less than** the value of any potential:  $sol$

$$sol \geq f.$$

(Proof. The desired solution of the problem consists of a  $n$  matrix of cells, and for each condition  $u[i] + v[j] \leq a[i][j]$ . Since all elements are in different rows and columns, then summing these inequalities over all selected  $a[i][j]$  in the left-hand side we get  $f$ , 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**. Hungarian algorithm, described below, will be constructive proof of this fact. In the meantime, we direct our attention to the fact that if a solution has a value equal to the value of any potential, then this solution - **optimally**.

Fix some potential. We call an edge **hard**, if performed:  $(i, j)$

$$u[i] + v[j] = a[i][j].$$

Think of the alternative formulation of the assignment problem, using a bipartite graph. Denoted by  $H$  a bipartite graph composed only of hard edges. In fact, the Hungarian algorithm support for the current capacity of **the maximum on the number of matching edges  $M$**  of the graph  $H$ : and as soon as it becomes contain matching  $n$ edges, the edges of this matching, and will be the desired optimal solution (in fact it will be the solution, the cost of which coincides with the building).

Proceed directly to the **description of the algorithm**.

- At the beginning of the algorithm potential is set to zero  $u[i] = v[i] = 0$ , and matching  $M$ relies empty.
- Further, at each step of the algorithm we are trying, without changing the potential to increase the capacity of the current matching  $M$ unit (recall matching is sought in the hard edges of the graph  $H$ ).

This actually use an ordinary [search algorithm Kuhn maximum matchings in bipartite graphs](#). We recall here the algorithm.

All matching edges  $M$ are oriented in the direction of the second part to the first, all other edges of the graph  $H$ are oriented in the opposite direction.

Recall (from the terminology matchings search) that vertex is called saturated if it is adjacent an edge from the current matching. Vertices that are not adjacent to any one edge of the current matching is called unsaturated. The path of odd length, which does not belong to the first edge matchings, and for all subsequent edges there is an alternation (owned / does not belong) - called by magnifying.

Of all the vertices of the first proportion of unsaturated run round [in depth / width](#). If the bypass was achieved unsaturated vertex of the second part, it means that we found a way of increasing the proportion of the first to the second. If procheredovat edges along the path (ie the first rib to include matching, the second to eliminate the third turn, etc.), we thereby increase the power of the matching unit.

If increasing the path was not, then this means that the matching of the current  $M$ - the maximum in the graph  $H$ , so in this case proceed to the next step.

- If the current step has failed to increase the current capacity of the matchings, then made a recalculation capacity so that the next steps there are more opportunities to increase the matching.

We denote by  $Z_1$ the set of vertices of the first part, which were visited traversal algorithm Kuna when you try to search for increasing the chain; through  $Z_2$ - the set of visited vertices of the second part.

Calculate the amount of  $\Delta$ :

$$\Delta = \min_{i \in Z_1, j \notin Z_2} \{a[i][j] - u[i] - v[j]\}.$$

This value is strictly positive.

(Proof. Suppose that  $\Delta = 0$ . Then there is a rigid edge  $(i, j)$ , with  $i \in Z_1$ and  $j \notin Z_2$ . From this it follows that the edge  $(i, j)$ had to be focused on the second part to the first, that is rigid rib  $(i, j)$ should be included in the matching  $M$ . However, this is not possible, because . we could not get to the top of the rich  $i$ , but to passing along the edge of  $ja$   $i$ . a contradiction, then  $\Delta > 0$ .)

Now **recalculate the potential** this way: for all vertices  $i \in Z_1$ do  $u[i]+ = \Delta$ , and for all vertices  $j \in Z_2$  will do  $v[j]- = \Delta$ . The resulting capacity will remain valid potential.

(Proof. For this we need to show that it is still for all  $i$  and  $j$  done:  $u[i] + v[j] \leq a[i][j]$ . For when  $i \in Z_1 \& j \in Z_2$  or  $i \notin Z_1 \& j \notin Z_2$  so, since they amount  $u[i]$  and  $v[j]$  has not changed. When  $i \notin Z_1 \& j \in Z_2$ - inequality only increased. Finally, for the case  $i \in Z_1 \& j \notin Z_2$ - although the left-hand side and increases inequality still persists, since the value  $\Delta$ , as seen in its definition - this is just the maximum increase that does not lead to a breach of inequality.)

In addition, the old matching  $M$  of the hard edges can be left, ie, matching all edges remain rigid.

(Proof. For some hard-edge  $(i, j)$  ceased to be hard as a result of changes in capacity, it is necessary to equality  $u[i] + v[j] = a[i][j]$  turned into inequality  $u[i] + v[j] < a[i][j]$ . However, the left-hand side can be reduced only in one case: when  $i \notin Z_1 \& j \in Z_2$ . But again  $i \notin Z_1$ , it means that the edge  $(i, j)$  could not be matching edge, QED.)

Finally, to show that the potential change **can not occur indefinitely**, we note that for every such change in the potential number of vertices reachable bypass, ie  $|Z_1| + |Z_2|$ , strictly increasing. (This is not to argue that the increased number of hard edges.)

(Proof. First, any vertex that was achievable, attainable and will remain. In fact, if a vertex is reachable, then it has to be a path of vertices reachable starting at the top of the unsaturated first part; and as for the edges of the form  $(i, j)$ ,  $i \in Z_1 \& j \in Z_2$  sum  $u[i] + v[j]$  does not change, then all the way to continue and after the building was to be proved. Secondly, we show that as a result of conversion capacity has appeared at least one new vertex achievable. But it is almost obvious, if you go back to the definition  $\Delta$ : that edge  $(i, j)$ , which was reached at least now become rigid, and therefore, the top  $j$  will be achievable thanks to the edges and vertices  $i$ .)

Thus, the total can not occur more than  $n$  calculations of the original building, before being discovered increasing the chain and matching power  $M$  will be increased.

Thus, sooner or later be found potential, which corresponds to a perfect matching  $M$ , which is the answer to the problem.

If we talk about **the asymptotic behavior** of the algorithm, it is  $O(n^4)$  because of all needs to happen  $n$  increases matchings, to each of which there is no more  $n$  calculations of the original building, each of which is executed in a time  $O(n^2)$ .

Implementation for  $O(n^4)$  we shall not give here, as it still will not 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 **be added to the consideration of the rows of the matrix, one by one**, rather than treating them all at once. Thus, the above algorithm takes the form:

- Add to the consideration of the next row of the matrix  $a$ .
- Yet increasing the chain starting in this line, we recalculate potential.
- As soon as a magnifying circuit, alternating matching along it (including the most recent in a string matching), and go to the top (to the next line).

To achieve the required asymptotic need to implement steps 2-3 performed for each row of the matrix, for the time  $O(n^2)$  (for rectangular tasks - for  $O(nm)$ ).

To do this, we recall two facts, we proved above:

- If you change the capacity tops that were achievable bypass Kuhn, achievable and will remain.
- Total could only happen  $O(n)$  calculations of the original building, before increasing the chain is found.

This raises **the key ideas** that achieve the desired asymptotics:

- To check the availability of increasing the chain no need to run round Kuhn again after each conversion potential. Instead, you can arrange tour Kuhn in **an iterative** way: after each conversion potential we look hard edges which were added and, if left ends were achievable, mark their right ends as achievable and continue to crawl out of them.
- Taking this idea further, you can come back to the view of the algorithm: a cycle, each step of which is first converted potential, then there is a column that became attainable (and there is always such as after conversion capacities are always new reachable vertices), and if this column was unsaturated, it found increasing the chain, and if the column was filled with - then the corresponding matchings in line also becomes achievable.

Next, the algorithm takes the form: the cycle of adding columns, each of which is first converted potential, and then some new column is marked as attainable.

- To quickly recalculate potential (faster than the naive version of  $O(n^2)$ ), it is necessary to maintain support minima for each of the columns  $j$ :

$$\minv[j] = \min_{i \in Z_1} \{a[i][j] - u[i] - v[j]\}.$$

As is easily seen, the unknown quantity  $\Delta$  is expressed through them as follows:

$$\Delta = \min_{j \notin Z_2} \{\minv[j]\}.$$

Thus, the finding  $\Delta$  can be made now for  $O(n)$ .

Support this array  $\minv[]$  must be visited when new rows. This obviously can be done in  $O(n)$  one appending string (which result in a total  $O(n^2)$ ). Also update the array  $\minv[]$  must be recalculated at capacity, which is also done during  $O(n)$  one Central building (since  $\minv[]$  the only change to the unreached while columns: namely, reduced by  $\Delta$ ).

Thus, the algorithm takes the form in the outer loop, we add in consideration matrix row one after the other. Each row is processed during  $O(n^2)$ , as this could happen only  $O(n)$  calculations of the original building (each - for the time  $O(n)$ ), which at the time  $O(n^2)$  supported the array  $\minv[]$ ; Kuhn's algorithm will work for a total time  $O(n^2)$  (as it is in the form  $O(n)$  of iterations, each of which is visited by a new column).

The resulting asymptotic behavior is  $O(n^3)$ - or if the task rectangular  $O(n^2m)$ .

## The implementation of the Hungarian algorithm for $O(n^3)$ ( $O(n^2m)$ )

The reduced implementation was actually developed **by Andrei Lopatin** few years ago. It features amazing conciseness: the whole algorithm is placed in **30 lines of code**.

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-indeksatsii 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 additional checks.

Arrays  $u[0 \dots n]$  and  $v[0 \dots m]$  store potential. Initially, it is zero, that is true for a matrix consisting of zero rows. (Note that this implementation is not important whether or not there is a

matrix of  $a[]$  (the negative number).

The array  $p[0 \dots m]$  includes a matching: for each column  $i = 1 \dots m$  it stores the number of the selected row  $p[i]$  (or 0, if nothing is yet set). In this case,  $p[0]$  for the convenience of implementation is set equal to the number of the current line under consideration.

The array  $minv[1 \dots m]$  contains for each column  $j$  supporting the minimum necessary for the rapid conversion potential:

$$minv[j] = \min_{i \in Z_1} \{a[i][j] - u[i] - v[j]\}.$$

The array  $way[1 \dots m]$  contains information about where these minima are obtained, that we were able to restore subsequently magnifying chain. At first glance it seems that in the array  $way[]$  for each column should be stored line number and 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 along the edge of the matching columns, so the number of rows to restore the chain can always take out a matching (ie, from the array  $p[]$ ). Thus,  $way[j]$  each column  $j$  contains the number of the preceding column (or 0, if none).

The algorithm itself is a an external **loop through the rows of the matrix** within which are added into consideration  $i$ th row of the matrix. The inner part is a series of "do-while ( $p[j0]! = 0$ )", which works until it finds a free column  $j0$ . Each iteration of the loop marks visited new column number  $j0$  (counted last iteration and initially zero - ie we are starting a dummy column), as well as a new line  $i0$ - adjacent to him in a matching (ie  $p[j0]$ , as if originally  $j0 = 0$  taken  $i$ th row). Due to the emergence of a new line being visited  $i0$  need to recalculate the appropriate array  $minv[]$ , at the same time we find a minimum in it - value  $\delta$ , and in which column  $j1$  this minimum has been reached (note that for such an implementation  $\delta$  would be equal to zero, which means that the potential of the current step You can not change: a new column attainable there already). Thereafter, a Central capacity  $u[], v[]$  corresponding to the change of the array  $minv[]$ . At the end of cycle "do-while" we found a magnifying chain terminating in a column  $j0$ , "spin," which can, using an array of ancestors  $way[]$ .

Constant  $INF$ - it "infinity", ie a number, obviously more of all possible numbers in the 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;
    } while (j0 != 0);
}

```

```

        j0 = j1;
    } while (p[j0] != 0);
    do {
        int j1 = way[j0];
        p[j0] = p[j1];
        j0 = j1;
    } while (j0);
}

```

Recovery in response a more conventional form, i.e. finding for each row  $i = 1 \dots n$  number of selected column therein  $ans[i]$ , is as follows:

```

vector<int> ans (n+1);
for (int j=1; j<=m; ++j)
    ans[p[j]] = j;

```

Cost matchings found you can just take the potential of zero column (taken with the opposite sign). In fact, it is easy to trace through the code  $-v[0]$  contains the sum of all values  $\delta$ , ie, total change in potential. Although at each change of the potential change could several variables  $u[i]$  and  $v[j]$ , the total change in value of the potential is exactly  $\delta$  because there is not increasing the chain, the number of accessible lines exactly one more than the number of columns accessible (only the current row  $i$  has no "couple" in the form of a column visited):

```

int cost = -v[0];

```

## Examples of tasks

Let us give you some examples to solve the assignment problem: from the very trivial, to less obvious tasks:

- Given a bipartite graph, find in it matching **the minimum weight maximal matching** (ie primarily maximized size matching, the second - the cost is minimized).

To solve the problem of building a simple assignment, putting in place the number of missing edges "infinity". After that we solve the problem of the Hungarian algorithm, and removed from the edges of the response of infinite weight (they could get back, if the problem has no solution in the form of a perfect matching).

- Given a bipartite graph, find in it matching **a maximal matching of maximum weight**.

The decision again is obvious, but all the weight must be multiplied by minus one (or Hungarian algorithm to replace all the minima on the highs and infinity - at minus infinity).

- The problem of **detecting moving objects from photographs**: two shots were made, the results of which were obtained two sets of coordinates. Required to relate the objects on the first and second picture, ie, determine for each point of the second picture, the first picture which point match. In this case, you want to minimize the sum of the distances between the mapped points (ie, we are looking for a solution in which objects have been the lowest total path).

To solve, we just build and solve the assignment problem, where the weights are the edges of the Euclidean distance between the points.

- The problem of **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) received information in the form  $n$  of such areas. Is required

to determine the position of objects, ie, determine the estimated position of objects and their corresponding pairs of directions so as to minimize the sum of distances from facilities to ray-direction.

Decision - again, just build and solve the problem of the destination where the vertices of the first part are the  $n$  areas with the first locator vertices of the second part -  $n$  the directions of the second locator and weights of edges - the distance between the beams.

- Covering a **directed acyclic graph ways** : given a directed acyclic graph, find the smallest number of ways (with equal - with the smallest total weight) to each vertex of the graph would lie exactly in the same way.

The solution - to build on the graph corresponding bipartite graph and find in it a maximal matching of minimum weight. For details, see. [a separate article](#).

- **Coloring wood** . Given a tree in which each vertex, but leaves has exactly  $k - 1$  sons. You want to select for each vertex of some color  $k$  colors so that no two adjacent vertices have the same color. In addition, for each vertex of each color paint known 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 of  $d[v][c]$  which  $v$ - the number of vertices  $c$ - number of colors, and the value  $d[v][c]$ - the minimum value of the vertex coloring  $v$  with her descendants, with the very tip of  $v$  a color  $c$ . To calculate this value  $d[v][c]$ , it is necessary to allocate the remaining  $k - 1$  flowers on the top of his sons  $v$ , and for this we need to build and solve the assignment problem (in which the top of one share - the colors, the top of the other share - tops, sons, and the weights of edges - This value corresponds to the speaker  $d[]$ ).

Thus, each value  $d[v][c]$  is considered by solving the assignment problem, which ultimately gives the asymptotic behavior  $O(nk^4)$ .

- If the task assignment weights are not set at the edges, and at the tops, and then only **at the vertices of one share** , one can dispense with the Hungarian algorithm, but rather a sort tops by weight and run a normal **algorithm Kuhn** (for more details see. [separate article](#) ).
- Consider the following **special case** . Let each vertex of the first part attributed to a number  $\alpha[i]$ , and each vertex of the second part -  $\beta[j]$ . Let the weight of each edge  $(i, j)$  is equal to  $\alpha[i] \cdot \beta[j]$  (number  $\alpha[i]$  and  $\beta[j]$  we know). Solve the problem of appointments.

To solve without the Hungarian algorithm we first consider the case when 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  $\alpha[i]$  connect with the vertex more  $\beta[j]$ . This rule can be easily generalized to an arbitrary number of vertices: it is necessary to sort out the top of the first part in the increasing order  $\alpha[i]$ , the second part - in descending order  $\beta[j]$ , and to connect the top of the pairs in that order. Thus, we obtain the solution with the asymptotic behavior  $O(n \log n)$ .

- **The problem of the potentials** . Given matrix  $a[1 \dots n][1 \dots m]$ . Required to find two arrays  $u[1 \dots n]$  and  $v[1 \dots m]$  such that for any  $i$  and  $j$  executed  $u[i] + v[j] \leq a[i][j]$ , but the sum of the elements of arrays  $u$  and  $v$  a maximum.

Knowing the Hungarian algorithm solution to this problem will have no difficulty: Hungarian algorithm just finds just such a potential  $u[], v[]$  which satisfies the conditions of the problem. On the other hand, without the knowledge of the Hungarian algorithm to solve this task 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 Transportation and Assignment Problems** [1957]

## Tasks in the online judges

A list of tasks to solve the assignment problem:

- UVA # 10746 "**Crime Wave - The Sequel**" [Difficulty: Low]
- 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]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2009 21:03  
EDIT: 18 Oct 2011 19:15

## Finding the minimum cut. Algorithm Curtains Wagner

### Statement of the Problem

Given an undirected weighted graph  $G$  with  $n$  vertices and  $m$  edges. Cut  $C$  is called a subset of vertices (in fact, the incision - a partition of vertices into two sets: belonging to  $C$  everyone else). The weight of the cut is the sum of the weights of edges passing through the slit, ie such edges, exactly one end of which is owned by  $C$ :

$$w(C) = \sum_{\substack{(v,u) \in E, \\ u \in C, v \notin C}} c(v, u),$$

where through  $E$  the set of all edges of the graph  $G$ , and through  $c(v, u)$  - the weight of an edge  $(v, u)$ .

Need to find a **cut of minimum weight**.

Sometimes this problem is called the "global minimum cut" - in contrast to the task when given vertex-drain and source, and you want to find the minimal cut  $C$  containing runoff and does not contain the source. Global minimal incision is minimum among the cuts minimum cost over all possible pairs of source-drain.

Although this problem can be solved by means of an algorithm for finding the maximum flow (running it  $O(n^2)$  again for all possible pairs of source and drain), but described below is much more simple and fast algorithm proposed by Matilda Stohr (Mechthild Stoer) and Frank Wagner (Frank Wagner) in 1994 .

Generally accepted loops and multiple edges, though, of course, hinges absolutely no effect on the result, and all multiple edges can be replaced by one edge to their total weight. Therefore, for simplicity, we will assume that the input graph loops and multiple edges are absent.

### Description of the algorithm

The basic idea of the algorithm is very simple. Will iteratively repeat the following process: finding the minimum cut between any pair of vertices  $s$ , and then combine these two vertices into one (combining the adjacency lists). In the end, after  $n - 1$  iteration, the graph shrinks to a single vertex and the process stops. After that, the answer will be the minimum among all  $n - 1$  sections found. Indeed, at each  $i$  stage of the found minimal incision  $C_i$  between the vertices  $s_i$  and  $t_i$  is either not desired global minimum cut, or, on the contrary, the tops  $s_i$  and  $t_i$  unprofitable belong to different sets, so we did not deteriorate, combining these two vertices into one.

Thus, we have reduced the problem to the following: for a given graph find the **minimum cut between some, any, a pair of vertices  $s$  and  $t$** . To solve this problem, the following has been proposed also an iterative process. We introduce a set of vertices  $A$ , which initially contains only an arbitrary vertex. At each step, there is a vertex, **most strongly associated** with the set  $A$ , ie, vertex  $v \notin A$ , for which the maximum value of the following:

$$w(v, A) = \sum_{\substack{(v,u) \in E, \\ u \in A}} c(v, u)$$

(ie, the maximum sum of the weights of edges, one end  $v$  and the other belongs to  $A$ ).

Again, this process is completed through the  $n - 1$  iteration when all vertices will move to the set  $A$  (by the way, this process is very similar to [Prim's algorithm](#)). Then, according to **Theorem Curtains, Wagner**, if we denote by  $s$  and the last two added to the  $A$  top, then the minimal incision between the vertices  $s$  and  $t$  will consist of a single vertex -  $t$ . The proof of this theorem will be given in the next section (as it often is, by itself it does not contribute to the understanding of the algorithm).

Thus, the overall **chart Curtains, Wagner** is. The algorithm consists of  $n - 1$  phase. At each phase of the set  $A$  consisting of the first relies any vertex; counted starting vertex weights  $w(v, A)$ . Then the  $n - 1$  iteration, each vertex of which is selected  $u$  with the highest value  $w(v, A)$  is added to the set and  $A$  then converted values  $w$  for the remaining peaks (which is obviously necessary to go through all the edges of the selected vertex adjacency list  $u$ ). After completing all iterations we remember in  $s$  and  $t$  numbers of the last two vertices added, and as the minimum value found between the incision  $s$  and  $t$  can take the value  $w(t, A \setminus t)$ . Then it is necessary to compare the found minimal incision with the current response, 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  $w$ . If you make it over  $O(n)$ , given that all phases  $n - 1$ , and  $n - 1$  in each iteration, the final **asymptotic behavior of the algorithm** is obtained

### Contents [hide]

- Finding the minimum cut. Algorithm Curtains Wagner
  - Statement of the Problem
  - Description of the algorithm
  - The proof Curtains Wagner
  - Implementation
  - Literature

$O(n^3)$ .

If for finding the vertex with the highest value  $w$  use the **Fibonacci heap** (which can increase the value of the key  $O(1)$  in the middle and make the most of the  $O(\log n)$  average), all associated with a variety of  $A$  operations on one phase of the executed  $O(m + n \log n)$ . The resulting asymptotic behavior of the algorithm in such a case will be  $O(nm + n^2 \log n)$ .

## The proof Curtains Wagner

Recall the condition of this theorem. If we add to the set of  $A$  all vertices at a time, each time adding the top most strongly associated with this set, we denote the penultimate added through the top  $s$ , and the last - through  $t$ . Then the minimal  $s$ - $t$  incision consists of a single vertex -  $t$ .

To prove this, consider an arbitrary  $s$ - $t$  incision  $C$  and show that its weight can not be less than the weight of the section, which consists of a single vertex  $t$ :

$$w(\{t\}) \leq w(C).$$

To this end, we prove the following fact. Let  $A_v$  - the state of the set  $A$  immediately before the addition of the vertices  $v$ . Let  $C_v$  - the cut sets  $A_v \cup v$  induced by the cut  $C$  (in other words,  $C_v$  is the intersection of these two sets of vertices). Next, the vertex  $v$  is called an active (with respect to the section  $C$ ), if the vertex  $v$  and added to the previous  $A$  vertex belong to different parts of the section  $C$ . Then, it is argued, for any active vertices  $v$  the inequality:

$$w(v, A_v) \leq w(C_v).$$

In particular, if  $t$  is an active vertex (since before it adds vertices  $s$ ), and  $v = t$  this inequality becomes an assertion of the theorem:

$$w(t, A_t) = w(\{t\}) \leq w(C_t) = w(C).$$

So, let us prove the inequality, which use the method of mathematical induction.

For the first active tops  $v$  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  $v$  - the other.

Now suppose that this inequality holds for all active vertices up to a vertex  $v$ , let us prove it for the next active vertex  $u$ . To do this, we transform the left-hand side:

$$w(u, A_u) \equiv w(u, A_v) + w(u, A_u \setminus A_v).$$

First, we note that:

$$w(u, A_v) \leq w(v, A_v),$$

- This follows from the fact that when the set  $A$  was equal  $A_v$ , it was added to the top of it  $v$ , and do not  $u$ , therefore, it had the largest value  $w$ .

Further, since  $w(v, A_v) \leq w(C_v)$  the induction hypothesis, we get:

$$w(u, A_v) \leq w(C_v),$$

where we have:

$$w(u, A_u) \leq w(C_v) + w(u, A_u \setminus A_v).$$

Now, note that the top  $u$  and all the vertices  $A_u \setminus A_v$  are in different parts of the section  $C$ , so this value  $w(u, A_u \setminus A_v)$  represents the sum of weights of edges, which are included in  $w(C_u)$ , but have not yet been taken into account in  $w(C_v)$ , which yields:

$$w(u, A_u) \leq w(C_v) + w(u, A_u \setminus A_v) \leq w(C_u),$$

QED.

We have proved the relation  $w(v, A_v) \leq w(C_v)$ , and from it, as mentioned above, it should be the whole theorem.

## Implementation

For the most simple and clear implementation (with the asymptotic  $O(n^3)$ ) was chosen as a representation of the graph adjacency matrix. The answer is stored in a variable `best_cost` and `best_cut` (desired value of the minimum cut themselves vertices contained in it).

For each vertex in the array `exists` is stored, whether it exists or it has been combined with any other node. The list `v[i]` for each vertex of the compressed `i` source stored numbers of vertices that have been compressed in this summit `i`.

The algorithm consists of the  $n - 1$  phase (with respect to cycle ph). At each phase of the first all vertices are outside the set  $A$ ,

for which the array `in_a` is filled with zeros, and the connectedness `w` of all vertices are zero. At each  $n - p$  iteration of the vertex is located `sel` with the largest magnitude `w`. If this is the last iteration, the answer, if necessary, updated, and the penultimate `prev` and last `sel` selected vertices are merged into one. If not the last iteration, it `sel` is added to the set `A`, and then recalculated the weight of all the other vertices.

It should be noted that the algorithm in its work "spoils" the graph `g`, so if it is still needed later, it is necessary to keep a copy of it before calling the function.

```

const int MAXN = 500;
int n, g[MAXN][MAXN];
int best_cost = 10000000000;
vector<int> best_cut;

void mincut() {
    vector<int> v[MAXN];
    for (int i=0; i<n; ++i)
        v[i].assign (1, i);
    int w[MAXN];
    bool exist[MAXN], in_a[MAXN];
    memset (exist, true, sizeof exist);
    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) {
            int sel = -1;
            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(), v[sel].end());
                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 \[1995\]](#)

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun .  
EDIT: 12 Aug

## The flow of minimum cost, minimum-cost circulation. Algorithm for removing negative weight cycles

### Setting targets

Let  $G$ - the network (network), that is a directed graph in which the vertices are selected, the source  $s$  and drain  $t$ . The set of vertices is denoted by  $V$ , lots of ribs - through  $E$ . Each edge  $(i, j) \in E$  compared its capacity  $u_{ij} \geq 0$  and cost per unit flow  $c_{ij}$ . If some edge  $(i, j)$  in the graph is not, it is assumed that  $u_{ij} = c_{ij} = 0$ .

**Flow** (flow) in the network  $G$  is defined to be real-valued function  $f$  which assigns to each pair of vertices  $(i, j)$  flow  $f_{ij}$  between them, and satisfy conditions:

- Bandwidth limit (performed for all  $i, j \in V$ ):

$$f_{ij} \leq u_{ij}$$

- Antisymmetry (holds for all  $i, j \in V$ ):

$$f_{ij} = -f_{ji}$$

- Saving flow (holds for all  $i \in V$  except  $i = s, i = t$ ):

$$\sum_{j \in V} f_{ij} = 0$$

The quantity of flow 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 **the minimum cost flow** is that for a given value of the flow  $|f|$  is required to find a stream with a minimum cost  $z(f)$ . It is noting that the value  $c_{ij}$  attributed to the ribs, are responsible for the cost of a unit of flow along this edge; sometimes the problem is encountered edges of the flow stream correlated value of the rib (i.e., if the magnitude of any flow occurs, then this value will be charged, regardless of the quantity of flow) - This problem has nothing in common with the considered here and, moreover, is NP-complete.

The task of finding **the maximum value of the minimum flow** is to find a maximum value flow, and among all of these - the minimum value. In particular case when the weights of all edges of the same, this problem becomes equivalent to the conventional Maximum flow problem.

The problem of finding **the minimum cost circulation** is to find the flow of zero value at the lowest cost. If all non-negative value, then, of course answer is zero flow  $f_{ij} = 0$ ; If there is a negative weight edges (or rather, the cycles of negative weight), even at zero flow is possible to find a circulation of negative value. The problem of finding the minimum cost circulation can, of course, put on the network without the source and drain, because the semantic load they carry (however, in this graph, you can add the source and drain in the form of isolated vertices and get the usual formulation of the problem on). Sometimes the task of finding the maximum value of the circulation - it is clear enough to change the value to the opposite edges and solve the problem of finding the minimum cost circulation already.

All these tasks are, of course, can be extended to undirected graphs. However, the move from an undirected graph to easily oriented: each undirected edge  $(i, j)$  with capacity  $u_{ij}$  and cost  $c_{ij}$  should be replaced by two parallel edges  $(i, j)$  and  $(j, i)$  with the same bandwidth and cost.

### Residual Network

The concept of **residual network**  $G^f$  is based on a simple idea. Suppose there is some flow  $f$ ; along each edge  $(i, j) \in E$  takes some flow  $f_{ij}$ . Then along this edge can (theoretically) to put more  $u_{ij} - f_{ij}$  units of flow; this value and call **residual capacity**:

$$r_{ij}^f = u_{ij} - f_{ij}$$

The cost of these additional units of the flow will be the same:

$$c_{ij}^f = c_{ij}$$

However, in addition, **direct** ribs  $(i, j)$  in the residual network  $G^f$  appears **opposite edge**  $(j, i)$ . The intuitive meaning of this edge that we can in the future to cancel the portion of the stream that flows along the edge  $(i, j)$ . Accordingly, the flow passing along the edges of the reverse  $(j, i)$  actually formally, means a reduction of the flow along the edges  $(i, j)$ . Back Ribs has a capacity equal to zero (so, for example,  $f_{ij} = 0$  and the reverse  $f_{ji} = 0$ ) would be impossible to miss the stream, with a positive value  $f_{ij} > 0$  for the reverse fins on the antisymmetry property would be  $f_{ji} < 0$  that sm

### Contents [hide]

- The flow of minimum cost, minimum-cost circulation. / removing negative weight cycles
  - Setting targets
  - Residual Network
  - Optimality criterion for the presence of negative weight cycles
  - Algorithm for removing negative weight cycles
  - Implementation
  - Literature

$c_{ji}^f = 0$ , ie it will be possible to pass some flow along Back Ribs), the residual capacity - equal to the flow along a straight edge and cost - the o (after all, after the abolition of the stream, we should 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 to  $G$  correspond to two oriented edges in the residual network  $G^f$ , and each edge of the residual network, an addition feature - the residual bandwidth. However, it is easy to note that the expression for the residual capacity  $r_{ij}^f$  is essentially the same for both the f and reverse edges, ie, we can write for any edge  $(i, j)$  residual network:

$$r_{ij}^f = u_{ij}^f - f_{ij}^f$$

By the way, the implementation of this feature prevents storing residual capacity, and just calculate them as necessary for the edge.

It should be noted that the residual network removes all edges that have zero residual bandwidth. Residual network  $G^f$  should comprise only the with positive residual capacity  $r_{ij}^f$ .

Here it is necessary to pay attention to this important point: if the network  $G$  were simultaneously both edges  $(i, j)$  and  $(j, i)$ , in the residual net each of which appears on the reverse side, and in the end there will be multiple edges . For example, this situation often occurs when the net built on an undirected graph (and, it turns out, each undirected edge in the end lead to the appearance of the four edges in the residual network) feature should always remember, it leads to a slight complication of programming, although in general does not change anything. In addition, the designation of an edge  $(i, j)$  in this case becomes ambiguous, so here we are everywhere, we assume that such a situation, the network is not for the purpose of simplicity and correctness of description, the correctness of the ideas it has no effect).

## Optimality criterion for the presence of negative weight cycles

**Theorem.** Some flow  $f$  is optimal (ie has the lowest cost among all the threads of the same size) if and only if the residual network  $G^f$  contains no negative weight cycles.

**Proof: the need .** Suppose that the flow  $f$  is optimal. Assume that the residual network  $G^f$  cycle comprises negative weight. Take this cycle of n weight, and choose at least  $k$  among the residual capacity of edges of the cycle ( $k$  to be greater than zero). But then we can increase the flow along each edge of the loop on the value  $k$ , with no flow properties are not violated, the flow rate does not change, but the value of the stream is reduced (reduced by the value of the cycle multiplied by  $k$ ). 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 flow): any stream  $f$  can be represented as a set of paths from the source to the drain and cycles all - having a p flow. We prove this lemma is constructive: we show how to break into a plurality of flow paths and cycles. If the stream is non-zero value, then, obviously, from the source  $s$  leaves at least one edge with a positive flow; Let's go through this edge, we find ourselves at some top  $v_1$ . If this vertex  $v_1 = t$ , then stop - found a way out  $s \rightarrow t$ . Otherwise, according to property maintain the flow of  $v_1$  must leave at least one edge with a positive flow, go through it at some vertex  $v_2$ . Repeating this process, we will either arrive in stock  $t$ , or it will come to some vertex for the second time. In the case, we find a way out  $s \rightarrow t$  in the second - cycle. Found a path / cycle will have a positive flow  $k$  (minimum thread edges of this path / cycle), reduce the flux of the ribs along each path / cycle value  $k$ , thereby preparing again flow to which apply the process again. Sooner or later, the flow all edges will be zero, and we'll find it on the way to decompose and cycles.

**Lemma 2** (on the difference between the fluxes) for any two streams  $f$  and gone value ( $|f| = |g|$ ) flow  $g$  can be represented as a stream  $f$ , plus cycles in the residual network  $G^f$ . Indeed, consider the difference between these flows  $g - f$  (flows subtraction - a 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. Proizvedem decomposition of the circulation of the preceding lemma. Obviously, it can not contain decomposition pathways (as JavaScript  $\& t$  By positive flow means that the flow in the network is positive). Thus, the difference of flow  $g$  and  $f$  can be represented as a sum of the network cycles  $G$ . Moreover, it will be a residual and the 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 . Thus, we consider an arbitrary flow  $f$  in the residual network that does not contain negative cost cycles. Consider also the flow of the same size, but the minimum cost  $f^*$ ; prove that  $f$  and  $f^*$  have the same cost. According to Le the flow  $f^*$  can be represented as the sum of the flow  $f$  and several cycles. But once the value of all non-negative cycles, then the value of the stream  $f^*$  can not be less than the value stream  $f$ :  $z(f^*) \geq z(f)$ . On the other hand, because the flux  $f^*$  is optimal, then its value can not be higher flux . Therefore,  $z(f) = z(f^*)$ , qed

## Algorithm for removing negative weight cycles

Just above theorem gives us a simple algorithm for finding the minimum cost flow: if we have some kind of flow  $f$ , then build for him a residual check whether it has a negative weight cycle. If no such cycle, the flow  $f$  is optimal (least cost has among all the threads of the same size). If the was found negative value, then calculate the flow  $k$  that you can skip further through this cycle (this  $k$  is equal to the minimum of the residual capacity edges of the cycle). Increasing the flow on  $k$  each edge along the cycle, we obviously will not break the flow properties without changing the flow reduce the cost of this stream, getting a new thread  $f'$  for which it is necessary to repeat the whole process.

So to start the process of improving the value of the flow, we need to find a pre- arbitrary flow desired value (some standard algorithm for finding maximum flow. See, eg, [Edmonds-Karp algorithm](#) ). In particular, if you want to find the least-cost circulation, then you can start simply with a zero

We estimate the asymptotic behavior of the algorithm. Search negative cost cycle in a graph with  $n$  vertices and  $m$  edges is made for  $O(nm)$  corresponding article . If we denote  $C$  the largest of the values of edges through  $U$ - most of the bandwidth, the maximum value does not exceed the value of the flow  $mCU$ . If all of the cost and capacity - are integers, each iteration of the algorithm reduces the value of the flow of at least one; therefore, all make the algorithm  $O(mCU)$  iterations, and the asymptotic behavior of the total amount to:

$O(nm^2CU)$

This asymptotic behavior - not strictly polynomial (strong polynomial), because it depends on the amount of bandwidth and cost.

However, if you do not look for an arbitrary negative cycle, and use some kind of a clear approach, the asymptotic behavior can be significantly reduced. For example, if every time to look for a cycle with a minimum average cost (which also can be made for  $O(nm)$ ), while the work of the whole algorithm is reduced to  $O(nm^2 \log n)$ , and this asymptotic behavior is already strictly polynomial.

## Implementation

First, we introduce data structures and functions to store the graph. Each edge is stored in a separate structure `edge`, all edges are in the list `edges` and for each vertex `v` in the vector `g[i]` are stored number of edges emanating from it. Such an organization makes it easy to find the number of reverse edges to the number of direct ribs - they are listed `edges[adjacent]`, and number one is available on the number of other operations "`^ 1`" (it invert least significant bit). Adding oriented edges in the graph provides a function `add_edge` that adds immediately 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 goes infinite loop within which the algorithm is executed Bellman-Ford, and if it detects a cycle of negative cost, then this cycle increases along the stream. Since the residual network can be a disconnected graph, the algorithm of Bellman-Ford runs from not yet reached the top. In order to optimize the algorithm uses a queue (all current `Q` and new line `NQ`) so as not to touch at every stage of all edges. Found along the cycle each time the unit is pushed smoothly flow, although, of course, in order to optimize the flow rate can be determined as a minimum residual capacity.

```
const int INF = 1000000000;
for (;;) {
    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) {
                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) {
                        int id = g[v][k];
                        if (edges[id].f < edges[id].u)
                            if (d[v] + edges[id].c < d[edges[id].to])
                                d[edges[id].to] = d[v] + edges[id].c;
                                par[edges[id].to] = v;
                                nq.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(), find (path.begin(), path.end(), v));
                    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 feed, Charles Leiserson, Ronald Rivest, Clifford Stein. [Introduction to Algorithms](#) [2005]
- Ravindra Ahuja, Thomas Magnanti, James Orlin. [Network Flows](#) [1993]
- Andrew Goldberg, Robert Tarjan. [Finding Minimum-Cost Circulations by Cancelling Negative Cycles](#) [1989]

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 13 Aug 2010 19:48  
EDIT: 14 Jun 2012 5:01

## Algorithm Diniz

### Statement of the Problem

Given a network, ie, directed graph  $G$  in which each edge  $(u, v)$  is assigned bandwidth  $c_{uv}$ , and identified two peaks - the source  $s$  and drain  $t$ .

To find in the network flow  $f_{uv}$  from the source  $s$  to the drain  $t$  maximum value.

### A bit of history

This algorithm was published by the Soviet (Israeli) scientist Efim Diniz (Yefim Dinic, sometimes written as "Dinitz") in 1970, ie even two years before the publication of the Edmonds-Karp algorithm (though both algorithms have been independently discovered in 1968).

In addition, it should be noted that some simplification of the algorithm were produced Ewen Shimon (Shimon Even) and his student Alon Itai (Alon Itai) in 1979. It is thanks to them that the algorithm has received its modern appearance: they used to the idea of the concept of blocking flows Diniz Alexander Karzanov (Alexander Karzanov, 1974), as well as the reformulated 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 the flow algorithms extremely interesting to consider, given the "iron curtain" of those years between the USSR and the West. It can be seen as similar ideas sometimes appeared almost simultaneously (as in the case of the algorithm Diniz and Edmonds-Karp algorithm), however, while having varying degrees of success (the 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 for more than a decade (the algorithm Karzanov push in 1974, and the algorithm of Goldberg (Goldberg) push in 1985).

### Necessary definitions

We introduce three essential definitions (each of them is independent of the others), which will then be used in the algorithm Diniz.

**Residual network**  $G^R$  towards network  $G$  and some flow  $f$  therein is called a network in which each edge  $(u, v) \in G$  with a capacity  $c_{uv}$  and stream  $f_{uv}$  two edges correspond to:

- $(u, v)$  a bandwidth  $c_{uv}^R = c_{uv} - f_{uv}$
- $(v, u)$  a bandwidth  $c_{vu}^R = f_{vu}$

It is noteworthy that with this definition in the residual network multiple edges may appear: if the original network were as rib  $(u, v)$  and  $(v, u)$ .

Residual edge can be intuitively understood as a measure of how much more you can increase the flow along some edges. In fact, if the rib  $(u, v)$  with the capacity  $c_{uv}$  to flow  $f_{uv}$  it is potentially possible for more skip  $c_{uv} - f_{uv}$  flow units, and in the opposite direction can be skipped to  $f_{vu}$  flow units, which

### Contents [hide]

- Algorithm Diniz
  - Statement of the Problem
  - A bit of history
  - Necessary definitions
  - Algorithm
    - Chart
    - The correctness of the algorithm
    - Estimating the number of phases
    - Search for blocking the flow
    - Asymptotics
      - Single network
  - Implementation
    - Implementation of the graphs in the form of co-occurrence matrices
    - Implementation on graphs as adjacency lists

would mean flux cancellation in the initial direction.

**Blocking the flow** in the network is called a stream that any path from the source  $s$  to the drain  $t$  contains saturated this stream edge. In other words, the network not find such a path from the source to the drain, along which it is possible to increase the flow freely.

Blocking the flow is not necessarily maximal. Ford-Fulkerson theorem says that the flow will be maximized if and only if a residual network not find  $s - t$  the path; in blocking the thread says nothing about the existence of the path on the edges that appear in the residual network.

**Layered network** for this network is constructed as follows. First the length of the shortest path from the source  $s$  to all other vertices; we call the level of  $\text{level}[v]$  the top of its distance from the source. Then layered network include all those edges  $(u, v)$  of the source network, which lead from one level to any other, later, the level, i.e.  $\text{level}[u] + 1 = \text{level}[v]$  (in this case, why the difference between the distances can not exceed unity, it follows from the properties of the shortest distances). Thus, all edges are removed, located entirely within the levels as well as ribs, leading back to previous levels.

Obviously, a layered network is acyclic. In addition, any  $s - t$  path in the layered network is the shortest path to the source network.

Build a layered network of the network is very easy: you need to run round in the width of the ribs of the network, thereby arguing for the value of each node  $\text{level}[]$ , and then make a layered network is suitable ribs.

Note. The term "layered network" in Russian literature is not used; usually this design is simply called "auxiliary graph." However, English is commonly used, the term "layered network".

## Algorithm

### Chart

The algorithm consists of several **phases**. Each phase is constructed first, the residual network, and then in relation to it is constructed layered network (bypass wide), and it looks for an arbitrary blocking flow. Found blocking the flow is added to the current thread, and that the next iteration ends.

This algorithm is similar to the algorithm of Edmonds-Karp, but the main difference can be understood as follows: at each iteration of 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).

### The correctness of the algorithm

We show that if the algorithm terminates, the output he gets the flow is maximum.

In fact, suppose that at some point in a layered network built for the residual network, unable to find a blocking flow. This means that the flow  $t$  does not achievable in a layered network of the source  $s$ . But as a layered network contains all 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. Hence, applying Theorem Ford-Fulkerson, we see that the current thread is actually maximized.

### Estimating the number of phases

We show that the algorithm always performs Diniz **less nphases**. To this end, we prove two lemmas:

**Lemma 1**. The shortest distance from the source to each vertex is not reduced with the implementation of each iteration, ie

$$\text{level}_{i+1}[v] \geq \text{level}_i[v]$$

where the subscript indicates the number of phases, which are taken to the values of these variables.

**Proof.** We fix an arbitrary phase  $i$  and arbitrary vertex  $v$  and consider any shortest  $s - v$  path  $P$  in the network  $G_{i+1}^R$  (remember, so we denote the residual network, taken before executing  $i + 1$ th phase). Obviously, the length of the path  $P$  is  $\text{level}_{i+1}[v]$ .

Note that in the residual network  $G_{i+1}^R$  may include only the edges of  $G^R$ , as well as ribs, the ribs of the inverse  $G^R$  (this follows from the definition of the residual network). We consider two cases:

- Path  $P$  contains only the edges of  $G^R$ . Then, of course, the path length  $P$  greater than or equal  $\text{level}_i[v]$  (because  $\text{level}_i[v]$ , by definition - the shortest path length), which means that the following inequality.
- Path  $P$  contains at least one edge that is not contained in  $G^R$  (but the converse to some edge of  $G^R$ ). Consider the first such edge; let it be an edge  $(u, w)$ .

$$s \Rightarrow u \rightarrow w \Rightarrow v$$

We can apply the lemma to the top  $u$ , because it falls under the first case; So, we obtain the inequality  $\text{level}_{i+1}[u] \geq \text{level}_i[u]$ .

Now, note that since the rib  $(u, w)$  appeared in the residual network only after  $i$ th phase, it follows that along the edge  $(w, u)$  was further passed some thread; therefore, the edge  $(w, u)$  network before belonged layered  $i$ -th phase and, therefore  $\text{level}_i[u] = \text{level}_i[w] + 1$ . We take into account that the properties of shortest paths  $\text{level}_{i+1}[w] = \text{level}_{i+1}[u] + 1$ , and combining this with the 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 inverted rib adds to level a minimum of two), and as a result we obtain the required inequality.

**Lemma 2.** The distance between the source and drain strictly increases after each phase of the algorithm, ie. :

$$\text{level}'[t] > \text{level}[t],$$

where the prime labeled value obtained in the next phase of the algorithm.

**Proof:** by contradiction. Suppose that after the current phase was found that  $\text{level}'[t] = \text{level}[t]$ . Consider the shortest path from the source to the drain; by assumption, its length should remain unchanged. However, the residual network in the next phase contains only the edges of the residual network before performing the current phase or reverse them. Thus, a contradiction: found a  $s - t$  path that does not contain saturated edges, and has the same length as the shortest path. This path should be "locked" blocks the flow of what had happened, and what is a contradiction, and the proof.

This lemma can be intuitively understood as follows: for  $i$ th phase Diniz algorithm identifies and nourishes all  $s - t$  paths of length  $i$ .

Since the length of the shortest path from  $s$  to  $t$  can not exceed  $n - 1$ , then, consequently, the algorithm performs Dinitz **not more  $n - 1$  phases**.

## Search for blocking the flow

To complete the construction of the algorithm Diniz, it is necessary to describe an algorithm for finding blocking flow in a layered network - a key place algorithm.

We consider three possible options for the implementation of the search block the flow:

- Search  $s - t$  path one by one until such ways are. The path can be found for  $O(m)$  a bypass in depth and in all these ways will  $O(m)$  (because each path saturates at least one edge). Summary of the asymptotic behavior of the search will be blocking the flow  $O(m^2)$ .
- Similarly to the previous idea, but removed in the process of crawling into the depths of Count all the "extra" edges, ie, edges along which will not reach the drain.

It is very easy to implement: just remove a rib after we watched it crawled in depth (except for the

case when we passed along the edge and found a way to drain). In terms of implementation, it is necessary to simply maintain the adjacency list of each vertex pointer to the first undeleted edge and increase this point in the cycle within the bypass in depth.

We estimate the asymptotic behavior of the solutions. Each round is terminated in the depth or saturation of at least one edge (if it has reached bypass flow) moving forward or at least one direction (otherwise). One can understand that one starts crawling in depth from the main program runs for  $O(k + n)$  where  $k$  - the number of promotion pointers. Considering that all starts crawling deep in the search of blocking the flow is  $O(p)$  where  $p$  - the number of edges, it blocks the flow of saturated, the entire search algorithm for blocking the flow of exhaust  $O(pk + pn)$  that, given that all pointers in the amount passed away  $O(m)$ , gives the asymptotic behavior  $O(m + pn)$ . In the worst case, when blocking the flow saturates all edges, the asymptotic behavior is obtained  $O(nm)$ ; This asymptotic behavior will be used on.

We can say that this method of finding blocking flow is extremely effective in the sense that in the search for ways of enhancing it spends  $O(n)$  an average of operations. Therein lies the difference by an order of effectiveness of the algorithm Diniz and Edmonds-Karp (who is looking for a way of increasing  $O(m)$ ).

This method of solution is still simple to implement, but fairly effective, and therefore most often used in practice.

- You can apply a special data structure - dynamic trees Sletora (Sleator) and Tarjan (Tarjan)). Then each block the flow can be found in time  $O(m \log n)$ .

## Asymptotics

Thus, the entire algorithm for Diniz performed  $O(n^2m)$  if the locking thread to search for the above-described method  $O(nm)$ . Implementation using dynamic trees Sletora and Tarjan will work during  $O(nm \log n)$ .

## Single network

Network unit ("unit network") called such a network in which the capacity of all existing edges are equal to one, and every vertex except the source and drain any incoming or outgoing edge only.

This case is important enough, as the problem of finding **maximum matching** network is built exactly the unit.

**We prove** that the algorithm of single networks Diniz even a simple implementation (which is arbitrary graphs work out for  $O(n^2m)$ ) works during  $O(m\sqrt{n})$  reaching to the problem of finding maximum matching one of the best known algorithms - Hopcroft-Karp algorithm.

First, we note that the above cited search algorithm blocking flow that works on arbitrary networks for time  $O(nm)$ , in networks with unit capacities will work for  $O(m)$ : due to the fact that each edge will not be seen more than once.

Second, estimate the total number of phases that could occur in the case of single chains.

Suppose that we have produced  $\sqrt{n}$  phase algorithm Diniz; then all increasing path length at most  $\sqrt{n}$  have already been discovered. Let  $f$  - the current flow was found, and  $f^*$  - the desired maximum flow; consider their difference:  $f^* - f$ . It is a residual flux in the network  $G^R$ . This stream has a magnitude  $|f^*| - |f|$ , and along each edge is zero or one. It can be decomposed into a set of  $|f^*| - |f|$  paths from  $s$  to  $t$  and possibly cycles. Because the network a single, all these paths can not have common vertices, therefore, given the above, the total number of vertices in them  $cnt$  can be estimated as:

$$cnt \geq (|f^*| - |f|) \cdot \sqrt{n}.$$

On the other hand, given that  $cnt \leq n$  we get from here:

$$|f^*| - |f| \leq \sqrt{n},$$

which means that even through the  $\sqrt{n}$  phases of the algorithm Diniz guaranteed to find the maximum

flow.

Therefore, the total number of phases of the algorithm Diniz performed on individual networks, can be estimated as  $2\sqrt{n}$ , as required.

## Implementation

We present two implementations of the algorithm  $O(n^2m)$  running on the network, given the adjacency matrices and adjacency list, respectively.

### Implementation of the graphs in the form of co-occurrence matrices

```

const int MAXN = ...; // число вершин
const int INF = 1000000000; // константа-бесконечность

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 previously read: must be specified variables  $n, s, t$  and is also considered the matrix of capacities  $c[i][j]$ . The main function of the decision - `dinic()` which returns the value of the maximum flow found.

## Implementation on graphs as adjacency lists

```

const int MAXN = ...; // число вершин
const int INF = 1000000000; // константа-бесконечность

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 previously read: must be specified variables  $n, s, t$  and added all edges (oriented) via function calls `add_edge`. The main function of the decision - `dinic()` which returns the value of the maximum flow found.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 23:01  
EDIT: 15 Feb 2012 23:59

## Kuhn's algorithm for finding the greatest matching in a bipartite graph

Given bipartite graph  $G$  containing  $n$  nodes and  $m$  edges. Required to find the maximum matching, ie, choose as many edges to the selected edge, none had a common vertex with any other selected edge.

### Contents [hide]

- Kuhn's algorithm for finding the greatest matching in a bipartite graph
  - Description of the algorithm
    - Necessary definitions
    - Theorem Berge
    - Algorithm Kuhn
    - Operation time
  - Implementation
  - Improved implementation

## Description of the algorithm

### Necessary definitions

A matching  $M$  is a set of pairwise non-adjacent edges of the graph (in other words, every vertex of the graph must be incident to no more than one of a plurality of ribs  $M$ ). Power matching will call the number of edges in it. The greatest (or maximum) will be called a matching matching, whose power is maximum among all possible matchings in the graph. All those vertices that have adjacent edges of the matching (ie, that have exactly one degree in the subgraph formed  $M$ ), we call this a matching saturated.

Chain length  $k$  will call some simple way (ie not containing repeating vertices or edges) containing the  $k$  edges.

Alternating chain (in the bipartite graph, with respect to a matching) will be called a chain in which the edges are alternately belong / do not belong to matching.

Increasing the chain (in the bipartite graph, with respect to a matching) will be called an alternating chain, whose initial and final vertices do not belong to matching.

### Theorem Berge

Wording . Matching the maximum if and only if there is no enhancing circuits

relative thereto.

**The proof of necessity**. We show that if matching  $M$  is maximal, then there is increased relative to a chain. Proof of this is constructive: we show you how to increase with increasing the chain using this power matching  $M$ .

To do this, perform the so-called matching alternation along the chain  $P$ . We recall that, by definition, the first edge chain  $P$  does not belong to a matching, the second - belongs to the third - again does not belong to the fourth - owned, etc. Let's change the status of all the edges along the chain  $P$ : those edges that are not included in the matching (the first, third and so on until the last) will include a matching and edges that were previously included in the matching (the second, fourth, and so on up to penultimate) - Removed from him.

It is understood that the power matching with increased by one (because it was added at one edge longer than the deleted). It remains to verify that we have built a correct matching, ie, that no vertex has no right of two adjacent edges of this matching. For all vertices alternating chain  $P$ , except the first and last, it follows from the very interleaving algorithm: first, we have removed the top of each of these adjacent edges, then added. For the first and last vertex chain  $P$  and nothing could be broken as to interleave they were to be unsaturated. Finally, for all the other vertices, - not in the chain  $P$ - obviously nothing has changed. Thus, we really built a matching, and per unit of greater power than the old one, which completes the proof of the necessity.

**The proof of sufficiency**. We prove that if a relatively matchings  $M$  not increase the ways it - as much as possible.

The proof is by contradiction. Suppose there is a matching  $M'$  having more power than  $M$ . Consider the symmetric difference  $Q$  of these two matchings, ie leave all edges belonging to  $M$ , or in  $M'$ , but not both simultaneously.

Clearly, the set of edges  $Q$ - is certainly not matching. Consider what kind of a set of edges is; For convenience, we consider 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 matching and from another). It is easy to understand that while this graph consists only of cycles or paths, with neither one nor the other does not intersect with each other.

Now, note that the path in this graph  $Q$  may not be any, but only of even length. In fact, in any path in the graph  $Q$  edges alternate: after the ribs of  $M$  a rib comes 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 matching  $M$  or for  $M'$ . But this could not be, because in the case of matching  $M$  it contradicts with the condition, and in the case  $M'$ - with its maximum (as we have already proved the necessity of the theorem, which implies that the existence of increasing the matching circuit can not be maximized).

We now prove a similar assertion for cycles all cycles in the graph  $Q$  may have only chëtnyu length. It is very easy to prove: it is clear that in the cycle edges should also alternate (owned by turns  $M$ , then  $M'$ ), but this condition can not be executed in a cycle of odd length - in it there are certainly two adjacent edges of the matching one that contradicts the definition of matching.

Thus, all the way and the cycles of the graph  $Q = M \oplus M'$  are chëtnuyu length. Therefore, the graph  $Q$  contains an equal number of edges of  $M$  and from  $M'$ . But considering that  $Q$  contains all the edges  $M$  and  $M'$ , except for their common edges, it follows that the power  $M$  and  $M'$  the same. We have a contradiction: on the assumption matching  $M$  was not maximal, then the theorem is proved.

## Algorithm Kuhn

Kuhn's algorithm - a direct application of Theorem Berge. It can be briefly described as follows: first, take an empty matching, and then - until the graph fails to find a magnifying chain - will perform striping matching along the chain, and repeat the process of searching for increasing the chain. Once such a chain could not be found - the process stops - the current matching is maximum.

It remains to detail the method of finding increasing chains. **The algorithm Kuhn** - just looking for any of these circuits by means of **bypass in depth** or **width**. Kuhn's algorithm looks at all the vertices one by one, starting from each round, trying to find a magnifying circuit, starting at this vertex.

More convenient to describe this algorithm, assuming that the graph is already divided into two parts (in fact the algorithm can be implemented and so that he was not given to the input graph is clearly divided into two parts).

The algorithm looks at all the vertices of  $v$  the first part of the graph:  $v = 1 \dots n_1$ . If the current node  $v$  is already saturated with a matching current (ie already selected some edge adjacent to it), then skip this vertex. Otherwise - the algorithm is trying to saturate this summit, which starts search for increasing the chain, starting from this node.

Search magnifying circuit by means of a special bypass in depth or width (usually for ease of implementation, use is bypassing depth). Originally bypass in depth is in the unsaturated current top  $v$  of the first part. View all the edges of this vertex, let the current edge - this edge  $(v, to)$ . If the top  $to$  is not saturated with a matching, it means that we could find magnifying circuit: it consists of a single edge  $(v, to)$ ; in that case, just include it in the edge matching and stop increasing the search from the top of the chain  $v$ . Otherwise - if  $to$  already filled with some edge  $(p, to)$ , then try to pass along this edge: thus we will try to find a magnifying circuit passing through the ribs  $(v, to), (to, p)$ . To do this, simply move on to our crawl to the top  $p$ - now we are trying to find a magnifying chain of this vertex.

One can understand that as a result of this tour, started from the top  $v$ , or find magnifying circuit, and thus saturate the top  $v$ , or as a magnifying circuit will not find (and, therefore, the vertex  $v$  will not be able to become rich).

After all vertices  $v = 1 \dots n_1$  will be reviewed, current matching is maximal.

## Operation time

Thus, the algorithm can be represented as Kuhn series of  $n$  launches bypass in depth / width over the entire graph. Consequently, all this algorithm is executed during  $O(nm)$  that in the worst case there  $O(n^3)$ .

However, this estimate may be a bit **better**. It turns out that the algorithm Kuhn importantly, what proportion is selected for the first, and which - for the second. In fact, in the above implementation of starting a crawl depth / width of the peaks occur only the first part, so the whole algorithm is executed in a time  $O(n_1 m)$  where  $n_1$ - the number of vertices of the first part. In the worst case it is  $O(n_1^2 n_2)$  (where  $n_2$ - the number of vertices of the second part). This shows that it is cheaper, when the first share contains fewer vertices than the second. Very unbalanced graphs (when  $n_1$ and  $n_2$ very different), this translates into a significant difference since the work.

## Implementation

We give here the implementation of the above algorithm, based on the bypass in depth, 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,  $g[v]$ - a list of edges from the top of  $v$ the first part (ie, a list of numbers of vertices, in which lead from these edges  $v$ ). Tops in both lobes are numbered independently, ie, first share - with the numbers  $1 \dots n$ , the second - with the 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 only for the vertices of the second part:  $mt[i]$ - is the number of vertices of the first part, connected by an edge with the top of  $i$ the second part (or  $-1$ , if no matching edges of the  $i$ leaves are not). The second array - `used`- the usual array of "visited" vertices crawled deep (you need it, just to bypass the depth did not come at the same vertex twice).

Function `try_kuhn`- is bypassing the deep. It returns `true`if she managed to find a magnifying chain from the top  $v$ , it shall be deemed that this function has already made alternation matchings found along the chain.

Inside the function to view all edges emanating from the vertex  $v$ of the first part, and then checked if this edge leads to the top of unsaturated `to`, or if this node `to` is full, but manages to find a magnifying circuit recursive run out `mt[to]`, then we say that we have found a magnifying circuit, and before returning from the function with the result of `true`producing alternating current in the edge: redirect edge adjacent to `to`, at the top  $v$ .

In the main program first indicates that the current matching - empty (the list `mt`is filled with numbers  $-1$ ). Then moved the top of  $v$ the first part, and it starts from the bypass in depth `try_kuhn`, pre-zeroing array `used`.

It is worth noting that the size of the matching is easy to get the number of calls `try_kuhn`in the main program, who returned result `true`. Needless desired 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() {
    ... чтение графа ...

    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, that Kuhn's algorithm is easy to implement and so he worked on graphs, for which we know that they are bipartite, but clear their division into two parts found. In this case, will have to abandon the easy division into two parts, and all the information stored for all vertices. For this array lists *g* now specifies not only for the vertices of the first part, and for all vertices (of course, now the top of both lobes are numbered in total numbering - from 1 up *n*). Arrays *mt* and *used* now also determined for the vertices of both lobes and accordingly, they must be maintained in this state.

## Improved implementation

We modify the algorithm is as follows. Before the main loop of the algorithm will find some simple algorithm **arbitrary matching** (a simple **heuristic algorithm**), and only then will perform a series of function calls *kuhn ()*, which will improve this matching. As a result, the algorithm will run much faster on random graphs - because in most graphs can easily dial matching sufficiently large weight by heuristics, and then improve the matching found to have a maximum

conventional algorithm Kuhn. Thus, we save on starting a crawl depth of the vertices that we have already included using heuristics in the current matching.

**For example**, you can simply iterate over 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 can accelerate the Kuhn several times.

It should be noted that the main loop will have to be modified slightly. Since the function is called `try_kuhn` in the main loop is assumed that the current node is not included in the matching, then you need to add the appropriate test.

In implementing change only the code in the function `main ()`:

```
int main() {
    ... чтение графа ...

    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 heuristic** is as follows. At each step will be to look for the top of the least likely (but not isolated), select it from any edge, and add it to the matching, then remove both these vertices with all edges incident to them from the graph. Such greed works very well on random graphs, even in most cases builds maximal matching (albeit against her have a test in which it finds a matching significantly lower value than the maximum).

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 23:02  
EDIT: 10 Jun 2008 23:02

## Checking on the graph bipartition and splitting into two parts

Suppose we are given an undirected graph. Need to check whether it is bipartite, ie whether it is possible to divide the vertices into two parts so that there is no edges connecting two vertices of one share. If the graph is bipartite, then bring themselves share.

We solve this problem by using [breadth-first search](#) for the O (M).

### Sign bipartite

Theorem. Graph is bipartite if and only if all its simple cycles have length чётный.

However, from a practical point of view to look for all the simple cycles uncomfortable. Much easier to check the graph on the bipartition following algorithm:

### Algorithm

Произведём сериях of breadth-first search. Ие will be launching a breadth-first search from each unvisited vertex. The vertex from which we start to go, we put in the first part. In the search for wide, if we go into some new vertex, then we put it in a fraction different from the share of the current top. If we try to pass on to the top edge, which already had, then we check that this 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 of a graph into two parts.

### Implementation

```

int n;
vector <vector <int>> G;
... Read Count ...

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

```

### Contents [\[hide\]](#)

- Checking on the graph bipartition and splitting into two parts
  - Sign bipartite
  - Algorithm
  - Implementation

```
ok = & Part [to]! = Part [v];  
    }  
}  
  
puts (ok? "YES": "NO");
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jul 2008 15:37  
EDIT: 11 Jul 2008 17:13

## Finding the maximum weight vertex-weighted matchings

### Contents [hide]

- Finding the maximum weight vertex-weighted matchings
  - Algorithm
  - Proof

Given bipartite graph G. For each vertex of the first part of its specified weight. Required to find the maximum weight matching, ie, with the largest sum of the weights of saturated peaks.

Below we describe and prove the algorithm based on the [algorithm of Kuhn](#), who will find the optimal solution.

## Algorithm

The algorithm itself is extremely easy. Let's sort the top of the first part, in descending order (more precisely, non-increasing) scales, and apply the resulting graph [algorithm Kuhn](#).

It is alleged that obtained with the maximum (in terms of number of edges) and a matching is optimal in terms of the amount of saturated vertex weights (despite the fact that after sorting, we do not actually use these weight).

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 from the algorithm Kuhn.

## Proof

Recall the basic provisions of [the theory of matroids](#).

Matroid M - is an ordered pair  $(S, I)$ , where  $S$  - the a lot of,  $I$  - a non-empty family of subsets of  $S$ , which satisfy the following conditions:

1. The set  $S$  is finite.
2. I family is hereditary, ie if one set belongs to  $I$ , then all of its subsets also belong to  $I$ .
3. Structure M has the property replacement, ie if  $A \in I$ , and  $B \in I$ , and  $|A| < |B|$ , then there is an element  $x \in AB$ , that  $A \cup x \in I$ .

Elements of the family  $I$  called independent subsets.

Called the weighted matroid, if for each element defined  $x \in S$  some weight. Weight subset called 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 an empty subsets will be adding (unless, of course, the current item can be added without disturbing 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 we prove that our task** - not that other, as a weighted **matroid** .

Let  $S$  - set of all vertices of the first part. To reduce the problem in a bipartite graph matroid respect to the vertices of the first part, we assign each a matching is a subset  $S$ , which is the set of vertices of the first part saturated. You can also define the opposite line (from the set of saturated peaks - in matching), which, although not unequivocal, but we will be quite happy.

Then I define a family as a family of subsets of  $S$ , for which there is at least one corresponding matching.

Next, each element of  $S$ , that for each vertex of the first part, in terms of certain of some weight. And the weight of the subset, as we required in the framework of the theory of matroids, defined as the sum of the weights of elements in it.

Then the problem of finding the maximum weight matchings now reformulated as the problem of finding the maximum weight independent subset.

It remains to verify that the following three conditions described above, imposed on the matroid. Firstly, it is obvious that  $S$  is finite. Secondly, it is obvious that the removal of rib from removal matchings equivalent vertices of a plurality of vertices saturated, and therefore the property of inheritance is performed. Third, as follows from the correctness of the algorithm Kuhn, if the current matching is

not maximal, then there will always be a vertex, which can be satiate, without removing from the set of vertices of saturated other peaks.

Thus we have shown that our task is a weighted matroid on a variety of saturated peaks of the first part, but because it applies the greedy algorithm.

It remains to show that the **algorithm Kuhn is this greedy algorithm**.

However, it's pretty obvious. Kuhn's algorithm at each step of trying to saturate the current node - or just spending an edge in the top of the second part unsaturated or are lengthening chain and matching alternating along it. And in fact, in both cases no longer saturated vertex does not cease to be unsaturated and unsaturated vertices in the previous steps of the first part is not saturated, and in this step. Thus, the algorithm Kuhn is a greedy algorithm that builds the best independent subset of a matroid, which completes our proof.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 2 Mar 2009 17:45  
EDIT: 6 Dec 2012 11:58

## Edmonds algorithm for finding maximum matching in an arbitrary graph

Given an undirected unweighted graph  $G$  with  $n$  vertices. You want to find in it a maximum matching, ie, is the largest (in power), the set  $m$  of its edges that no two edges of the selected are not incident to each other (ie, have no common vertices).

Unlike the case of a bipartite graph (see. [The algorithm Kuhn](#) ), the graph  $G$  may be present cycles of odd length, which greatly complicates the search for ways of increasing.

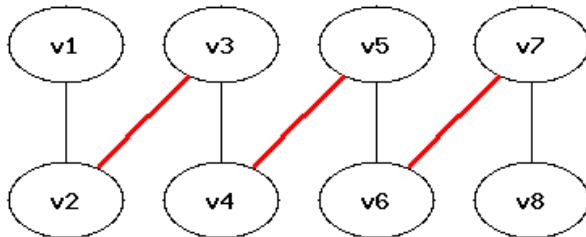
We give first Berge's theorem, which implies that, as in the case of bipartite graphs, maximum matching can be found by means of enhancing ways.

### Contents [hide]

- Edmonds algorithm for finding maximum matching in an arbitrary graph
  - Increasing path. Theorem Berge
  - Edmonds algorithm. Compression flowers
  - Effective implementation
  - Optimization: preliminary construction matchings
  - The case of a bipartite graph
  - Further optimization

### Increasing path. Theorem Berge

Let a fixed matching  $M$ . Then a simple chain  $P = (v_1, v_2, \dots, v_k)$  is called an alternating chain if it turns edges belong - do not belong matchings  $M$ . Alternating chain called increased if its first and last vertex does not belong to matching. In other words, a simple circuit  $P$  is then magnifying and only when the vertex  $v_1 \notin M$ , edge  $(v_2, v_3) \in M$ , rib  $(v_4, v_5) \in M$ , ..., an edge  $(v_{k-2}, v_{k-1}) \in M$ , and the vertex  $v_k \notin M$ .



**Theorem Berge** (Claude Berge, 1957). Matching  $M$  is greatest if and only if it does not exist for increasing the chain.

**The proof of necessity**. Suppose for matching  $M$  there is an increasing chain  $P$ . We show how to move to a matching higher power. Perform alternation matching  $\bar{M}$  along the chain  $P$ , ie, include in matching edges  $(v_1, v_2)$ ,  $(v_3, v_4)$ ...,  $(v_{k-1}, v_k)$  and remove from the matching edges  $(v_2, v_3)$ ,  $(v_4, v_5)$ ...,  $(v_{k-2}, v_{k-1})$ . The result is likely to be obtained by the correct matching, whose power will be one higher than the matching  $M$  (because we added  $k/2$  ribs and removed the  $k/2 - 1$  edge).

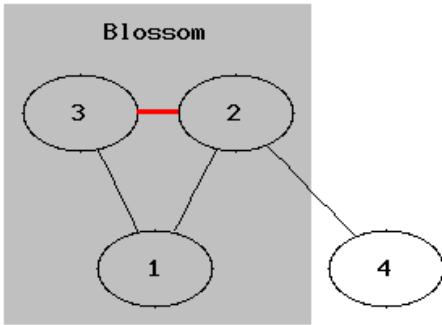
**The proof of sufficiency**. Suppose for matching  $M$  does not exist magnifying circuit, we prove that it is the largest. Let  $\bar{M}$  - maximum matching. Consider the symmetric difference  $G = M \oplus \bar{M}$  (ie, the set of edges belonging to either  $M$  or  $\bar{M}$ , but not both simultaneously). We show that  $G$  contains the same number of edges  $M$ , and  $\bar{M}$  (as we have excluded from  $G$  only their common edges, it will follow and  $|M| = |\bar{M}|$ ). Note that  $G$  only consists of simple chains and cycles (as otherwise one would be incident to the top two edges immediately any matching, which is impossible). Further cycles can not be odd length (for the same reason). Chain  $G$  also can not have an odd length (otherwise it is the increasing chain for  $M$  which contradicts the condition, or  $\bar{M}$  that is contrary to its maximum). Finally, in the even-numbered cycles and the even chain lengths  $G$  are included in the alternating ribs  $M$  and  $\bar{M}$  that means that  $G$  part of the same number of ribs  $M$  and  $\bar{M}$ . As mentioned above, it follows that  $|M| = |\bar{M}|$ , that  $M$  is a maximum matching.

Berge theorem provides a basis for the algorithm Edmonds - Search enhancing circuits and alternating along them until the chains are increasing.

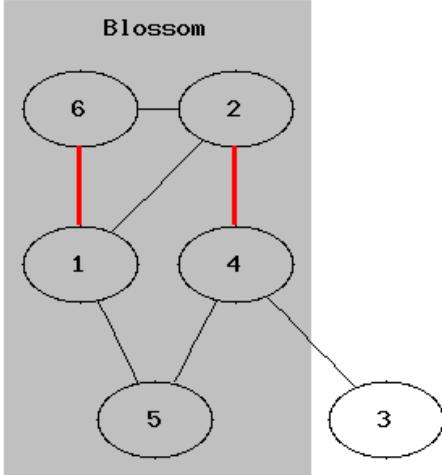
## Edmonds algorithm. Compression flowers

The main problem is how to find the path increases. If the graph has cycles of odd length, then just run round in depth / width impossible.

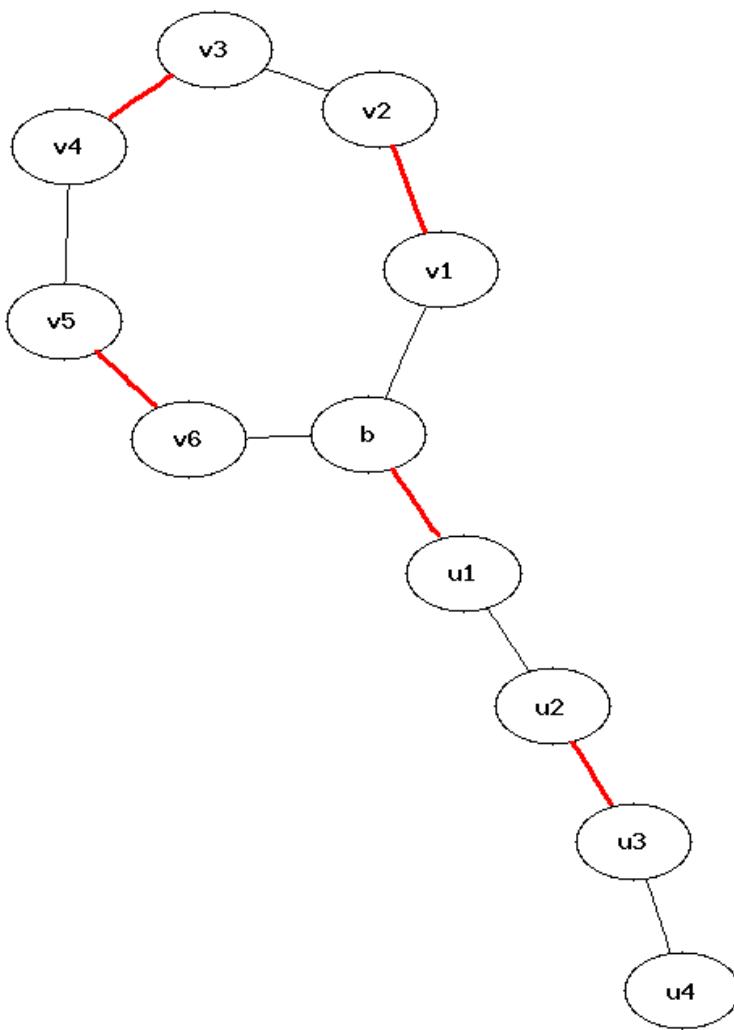
You can give a simple counterexample, when you run out of one of the vertices algorithm does not handle the extra cycles of odd length (in fact, [Kuhn's algorithm](#)) does not find a way increases, although it should. It is a cycle of length 3 with hanging on the edge of it, ie, Count 1-2, 2-3, 3-1, 2-4, 2-3 and the edge is taken in matching. Then when you start from the top 1 if the first round goes to the top 2, it "you are rested" in the top 3, instead of finding a magnifying circuit 1-3-2-4. However, in 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 on which at a certain order in the adjacency list algorithm Kuhn go down a blind alley. As an example, a graph with six vertices and edges 7: 1-2, 1-6, 2-6, 2-4, 4-3, 1-5, 4-5. If we apply the algorithm here Kuhn, he finds a matching 1-6, 2-4, after which he will have to find a chain of magnifying 5-1-6-2-4-3, but can never find it (if from the top 5 he will go first in 4, and then to 1 and when you run out of the top 3, it goes from the top 2 in the first 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 bypass can go on a cycle in the wrong direction. In fact, we are only interested in "saturated" cycles, i.e. which has  $k$  saturated the edges, where the length of the cycle is equal  $2k + 1$ . In this cycle, there is exactly one vertex not saturated edges of this cycle, let's call it **the base** (base). To the top of the base is suitable alternate way of even (possibly zero) of length, beginning in the free (ie not owned by a matching) the top, and this way is called a **stem** (stem). Finally, the subgraph formed "saturated" odd cycle, called the **flower** (blossom).



The idea of the algorithm Edmonds (Jack Edmonds, 1965) - in **compression flowers** (blossom shrinking). Compression flower - a compression of all odd cycles in a pseudo-vertex (respectively, all edges incident to the vertex of the cycle, are incident to the pseudo-top). Edmonds algorithm searches the graph all the flowers, compresses them, and then the graph does not remain "bad" cycles of odd length, and on such a graph (called a "surface" (surface) of the graph) can already be found magnifying circuit simply by walking in the depth / width. After finding increasing the chain in the surface box must "expand" flowers, restoring thereby increasing the circuit in the original graph.

However obvious that after compression of the flower is not violated graph structure, namely, that if the space  $\bar{G}$  existed increasing the chain, then it exists in the graph  $\bar{G}$  obtained after compression of the flower, and vice versa.

**Edmonds theorem**. In the field  $\bar{G}$  there is an increasing chain if and only if there is an increasing chain  $G$ .

**Proof**. So, suppose that the graph  $\bar{G}$  was obtained from the graph  $G$  compression of one flower (denoted by  $B$  a cycle of a flower, and after  $\bar{B}$  that the condensed vertex), we prove the theorem. First we note that it suffices to consider the case when the base of the flower is a free vertex (not owned by a matching). In fact, otherwise, the base ends of the flower alternate path even length commencing at the free top. Procheredovav matching along this path, the power matching will not change, and the base of the flower will be a free apex. So the proof we can assume that the base of the flower is a free vertex.

**The proof of necessity**. Let the path  $P$  is increasing in the graph  $G$ . If he does not go through  $B$ , then obviously it will increase in the graph  $\bar{G}$ . Let  $P$  passes through  $B$ . Then we can without loss of generality assume that the path  $P$  is a path  $P_1$  that does not pass on the tops  $B$ , plus a path  $P_2$  running along the tops  $B$  and possibly other vertices. But then the path  $P_1 + \bar{B}P_2$  will be increasing the path in the graph  $\bar{G}$ , as required.

**The proof of sufficiency**. Let way  $\bar{P}$  is by increasing the in box  $\bar{G}$ . Again, if the path  $\bar{P}$  does not go through  $\bar{B}$ , then the path  $\bar{P}$  is unchanged by increasing the in  $\bar{G}$ , so this case will not be considered.

We consider separately the case when  $\bar{P}$  the flower begins with a compressed  $\bar{B}$ , ie has the form  $(\bar{B}, c, \dots)$ . Then in the flower  $B$  exists a corresponding vertex  $v$ , which is connected (unsaturated) with an edge  $c$ . It remains only to note that the base of the flower always find alternate path of even length to the top  $v$ . Given all the above, we find that the path  $P = (b, \dots, v, c, \dots)$  is by increasing the in box  $G$ .

Suppose now that the path  $\overline{P}$  passes through the pseudo-vertex  $\overline{B}$ , but does not begin and end there. Then  $\overline{P}$  there are two edges passing through  $\overline{B}$ , let it  $(a, \overline{B})$  and  $(\overline{B}, c)$ . One of them must necessarily belong to a matching  $M$ , however, because base of the flower is not saturated, and all the other vertices of the cycle of a flower  $B$  full of ribs cycle, we arrive at a contradiction. Thus, this case is simply not possible.

So, we have considered all cases and in all of them showed the theorem Edmonds.

The overall chart Edmonds takes the following form:

```

void edmonds() {
    for (int i=0; i<n; ++i)
        if (вершина i не в паросочетании) {
            int last_v = find_augment_path (i);
            if (last_v != -1)
                выполнить чередование вдоль пути из i в last_v;
        }
}

int find_augment_path (int root) {
    обход в ширину:
    int v = текущая_вершина;
    перебрать все рёбра из v
        если обнаружили цикл нечётной длины, сжать его
        если пришли в свободную вершину, return
        если пришли в несвободную вершину, то добавить
            в очередь смежную ей в паросочетании
    return -1;
}

```

## Effective implementation

Immediately estimate the asymptotic behavior. A total of  $n$  iterations, each of which is bypassed in width for  $O(m)$  further compression operation may occur flowers - can be blocked  $O(n)$ . Thus, if we learn how to compress the flower behind  $O(n)$ , then the total amount to the asymptotic behavior of the algorithm  $O(n \cdot (m + n^2)) = O(n^3)$ .

The main difficulty represent operations compression flowers. If you execute them directly by combining the adjacency lists into one and removing it from the graph of extra vertices, the asymptotic behavior of the compression of one flower will  $O(m)$  also have difficulty with "unfolding" of flowers.

Instead, we shall, for each vertex of the graph  $G$  maintain a pointer to the base of the flower to which it belongs (or yourself, if the node does not belong to any flower). We need to solve two problems: compression of the flower  $O(n)$  when it is detected, as well as convenient storage of all information for further increasing the alternation along the way.

So, one iteration of the algorithm Edmonds is a wide detour that runs from the top of a given free  $root$ . Gradually will build a tree traversal in width, with the way in it to any vertex will be the alternate 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 these vertices the even - that is the root of the tree, and the second ends of edges in a matching). The tree itself will be stored in an array of ancestors  $p[]$ , in which for each odd vertex (ie, the distance to which the tree is odd ways, that is the first ends of the edges in the matching) will be stored ancestor - chëtnuyu top. Thus, to restore the path of the tree we need to alternately use arrays  $p[]$  and  $match[]$  where  $match[]$  - for each vertex contains adjacent to it in a matching, or  $-1$ , if not available.

Now it becomes clear how to detect cycles of odd length. If we are out of the current node  $v$  in the traversal wide come to a vertex  $u$  is the root of  $root$  or belonging to a matching and tree paths (ie,  $p[match[]]$  on which is not equal to 1), we found a flower. Indeed, under these conditions, and the top  $v$ , and the top  $u$  are the even vertices. Distance from them to their lowest common ancestor has one parity, so we have found a cycle of odd length.

Learn how to **compress cycle**. Thus, we have discovered an odd cycle when considering the rib  $(v, u)$  where  $u$  and  $v$  - the even peaks. Will find their lowest common ancestor  $b$ , and he will be the base of the flower. It is easy to notice that the base is also an even vertex (because odd vertices in the tree paths have only one son). However, it should be noted that  $b$  - it may pseudovertex, so we actually find the base of the flower, which is the lowest common ancestor of vertices  $v$  and  $u$ . We realize at once to find the lowest common ancestor (we are quite satisfied with the asymptotic behavior  $O(n)$ ):

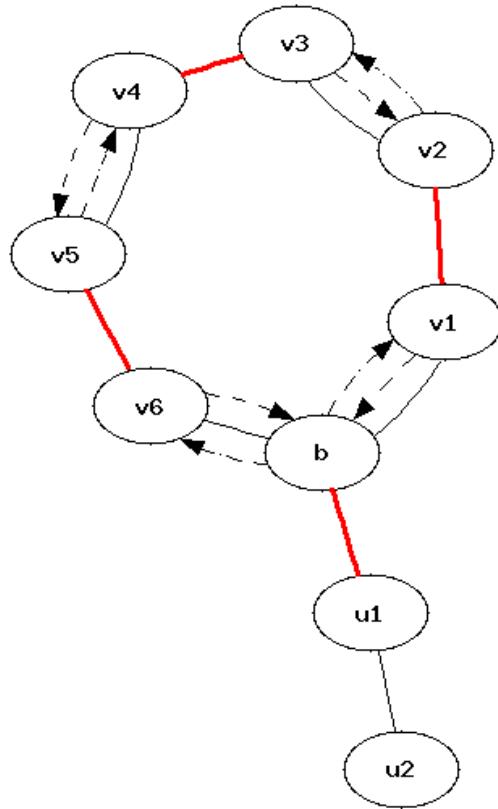
```

int lca (int a, int b) {
    bool used[MAXN] = { 0 };
    // поднимаемся от вершины a до корня, помечая все чётные вершины
    for (;;) {
        a = base[a];
        used[a] = true;
        if (match[a] == -1) break; // дошли до корня
        a = p[match[a]];
    }
    // поднимаемся от вершины b, пока не найдём помеченную вершину
    for (;;) {
        b = base[b];
        if (used[b]) return b;
        b = p[match[b]];
    }
}

```

Now we need to identify the cycle - to walk from the vertices  $v$  and  $u$  to the base  $b$  of the flower. Will be more convenient if we are simply marks in some special array (call it  $blossom[]$ ) vertices belonging to the current flower. After that we will have to reproduce the preorder traversal of the pseudo-tops - it is enough to put in place a preorder traversal of all vertices lying on the flower cycle. Thus, we avoid an explicit join adjacency lists.

However, there was still one problem: the correct path recovery at the end of bypass wide. For him, we maintain an array of ancestors  $p[]$ . But after compression flowers occurs only problem: a preorder traversal continued right of all vertices of the cycle, including odd and an array of ancestors we intended to restore an even ways to the top. Furthermore, when there exists a compressed graph magnifying chain through flower it generally will be for this cycle in such a direction that the paths in the tree will represent the motion is downward. However, all these problems are solved so gracefully maneuver: the compression cycle by putting all of its ancestors for the even vertices (except the base), that these "ancestors" pointed to the top of the next cycle. For vertices  $u$  and  $v$ , if they are not base pointers direct the ancestors of each other. As a result, if the restoration of enhancing the way we come to the cycle of a flower in the top of the odd, the way to the ancestors will be restored correctly and will result in the base of the flower (of which he has will continue to recover normal).



So, we are ready to implement the compression of the flower:

```

int v, u; // ребро (v,u), при рассмотрении которого был обнаружен цветок
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()` takes place on the way from the top to the base of the flower, affix a special array `blossom[]` for them `true` and affix ancestors for even vertices. Parameter `children`- the son to the very top  $v$  (with this option we zamknëm cycle 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, implement the basic function - `find_path (int root)` which will look enhances the way from the top of the free `root` and return the last vertex of this path, or `-1` if increasing path was not found.

Initially proizvedëm 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 preorder traversal. Considering the next edge  $(v, to)$ , we have several options:

- Edge nonexistent. By this we mean that  $v$ , and  $to$  belong to a single compressed pseudo-node ( $base[v] == base[to]$ ), so this graph the surface of the edge is not. Apart from this case, there is one case: when the edge  $(v, to)$  is already in the current matchings; because we believe that the top  $v$  is an even vertex, then pass along this edge is in the tree tract ascent to the top of the ancestor  $v$ , which is unacceptable.

```
if (base[v] == base[to] || match[v] == to) continue;
```

- Edge closes the cycle of odd length, ie, found a flower. As mentioned above, the odd cycle length detected by the condition:

```
if (to == root || match[to] != -1 && p[match[to]] != -1)
```

In this case, you must compress the flower. It has already been examined in detail this process, here we present 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 "customary" fin act like a normal search in width. The only subtlety - checking that this summit 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  $to$  has not yet come in, and she was unsaturated, we found a magnifying chain ending at the top  $to$ , it will be returned.

```
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 give the definition of global arrays, and the implementation of the main program of finding maximum matching:

```

const int MAXN = ...; // максимально возможное число вершин во входном графе

int n;
vector<int> g[MAXN];
int match[MAXN], p[MAXN], base[MAXN], q[MAXN];
bool used[MAXN], blossom[MAXN];

...

int main() {
    ... чтение графа ...
}

```

```

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;
        }
    }
}

```

## Optimization: preliminary construction matchings

As in the case of [Algorithm Kuhn](#), before performing Edmonds algorithm can be any simple algorithm to build a pre-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 the bipartite graphs are no cycles of odd length, and therefore the code that performs the compression of flowers, will never be fulfilled. Mentally deleting all of the code that handles the compression of flowers, we get [Algorithm Kuhn](#) almost pure form. Thus, for bipartite graphs Edmonds algorithm degenerates into [an algorithm Kuhn](#) and works for  $O(nm)$ .

## Further optimization

In all the above operations with flowers thinly veiled operation with disjoint sets that can be performed much more efficiently (see. [The system of disjoint sets](#)). If we rewrite the algorithm using this structure, the asymptotic behavior of the algorithm decreases to  $O(nm)$ . Thus, for arbitrary graphs we have the same asymptotic estimate that in the case of bipartite graphs (algorithm Kuhn), but much more complex algorithm.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 2 Mar 2009 17:45  
EDIT: 31 Dec 2011 1:57

## Floor ways directed acyclic graph

Dan directed acyclic graph  $G$ .

Required to cover its lowest number of ways, ie, find the smallest set of disjoint power over the tops of simple ways such that each vertex belongs to a path.

### Contents [hide]

- Floor ways directed acyclic graph
  - Reduction to the bipartite graph
  - Weighted case

## Reduction to the bipartite graph

Suppose we are given a graph  $G$  with  $n$  vertices. We construct the corresponding bipartite graph  $H$  in a standard way, ie .: in each part of the graph  $H$  will  $n$  vertices denote them by  $a_i$  and  $b_i$ , respectively. Then, for each edge  $(i, j)$  of the original graph  $G$  will hold the corresponding edge  $(a_i, b_j)$ .

Each edge  $G$  corresponds to one edge  $H$ , and vice versa. If we consider in  $G$  any way  $P = (v_1, v_2, \dots, v_k)$ , then it is associated with a set of edges  $(a_{v_1}, b_{v_2}), (a_{v_2}, b_{v_3}), \dots, (a_{v_{k-1}}, b_{v_k})$ .

More easy to understand is, if we add a "reverse" edges, ie, form the graph  $\bar{H}$  from the graph  $H$  by adding edges of the form  $(b_i, a_i)$ ,  $i = 1 \dots N$ . Then the path  $P = (v_1, v_2, \dots, v_k)$  in the graph  $\bar{H}$  corresponds to the path  $\bar{Q} = (a_{v_1}, b_{v_2}, a_{v_2}, b_{v_3}, \dots, a_{v_{k-1}}, b_{v_k})$ .

Conversely, consider any path  $\bar{Q}$  in the graph  $\bar{H}$ , beginning in the first part and ending in the second part. Obviously,  $\bar{Q}$  again, will look like

$\bar{Q} = (a_{v_1}, b_{v_2}, a_{v_2}, b_{v_3}, \dots, a_{v_{k-1}}, b_{v_k})$ , and it can be associated with a graph  $G$  path  $P = (v_1, v_2, \dots, v_k)$ . However, there is one subtlety:  $v_1$  could match  $v_k$ , so the path  $P$  would have made a cycle. However, under the graph  $G$  is acyclic, so it is impossible (this is the only place where it is used acyclic graph  $G$ ; however, on cyclic graphs procedure described here can not be generalized at all).

So, every simple path in the graph  $\bar{H}$  begins at the first beat and ends in the second, we can assign a simple path in the graph  $G$ , and vice versa. But note that such a path in the graph  $\bar{H}$  - is a **matching** in a graph  $H$ . Thus, any path from  $G$  one can associate a matching in a graph  $H$ , and vice versa. Moreover, disjoint paths in the  $G$  match disjoint matchings in  $H$ .

The last step. Note that there are more paths in this set, all the lower edges of

these paths contain. Namely, if there is a  $p$  disjoint paths that cover all the  $n$  vertices of the graph, then they both contain  $r = n - p$  edges. So as to minimize the number of paths, we must **maximize the number of edges** in them.

So, we have reduced the problem to finding a maximum matching in a bipartite graph  $H$ . After finding this matching (see. [The algorithm Kuhn](#)) we have to convert it to a set of paths  $G$  (this is a trivial algorithm ambiguity does not arise here). Some peaks can remain unsaturated a matching, in which case it is necessary to add in response to a zero-length path from each of these vertices.

## Weighted case

Weighted case is not very different from the unweighted, just in the graph  $H$  appear on the edges of weight, and you want to find a matching has the least weight. Restoring response similar to unweighted cases, we obtain the graph cover the least number of ways, and at equality - the lowest cost.

# MAXimal

[home](#)
[algo](#)
[bookz](#)
[forum](#)
[about](#)

 Added: 16 Sep 2010 17:09  
 EDIT: 2 Nov 2012 18:32

## Tutte matrix

Tutte matrix - is an elegant approach to solving the problem of a **matching** in an arbitrary (not necessarily bipartite) graph.

However, in the simplest form, the algorithm does not produce themselves edges included in the matching, and only the size of a maximum matching in a graph.

Below, we first consider the results obtained Tutt (Tutte) to verify the existence of a perfect matching (ie matching containing  $n/2$ edges, and therefore saturating all  $n$  vertices). After that, we will review the results later Lovas (Lovasz), which already allows the size of the maximum matching look, and not just limited to the case of perfect matchings. This is followed by the result of Rabin (Rabin) and Vazirani (Vazirani), who reported the recovery of the matching algorithm (as a set of its constituent edges).

### Contents [hide]

- Tutte matrix
  - Determination
  - Tutte's theorem
    - Practical application: a randomized algorithm
    - The proof of Tutte
      - Theorem Edmonds
      - Properties antisymmetric matrices
      - The proof of Tutte
  - Lovász theorem: generalization to find the size of the maximum matching
    - Formulation
    - Application
    - Proof
  - Rabin-Vazirani algorithm for finding the maximum matching
    - Statement of the theorem
    - Application
    - The proof of Theorem
  - Literature

## Determination

Suppose we are given a graph  $G$  with  $n$  vertices (  $n$ - is 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  $x_{ij}$  ( $1 \leq i < j \leq n$ ) - it is either independent variable corresponding to

the edges between the vertices  $i$  and  $j$  or identical zero if the edges between these vertices not.

Thus, in case of a complete graph with  $n$  vertices Tutte matrix contains  $n(n - 1)/2$  independent variables in the graph if some missing edges, the corresponding matrix elements Tatta converted to zeros. Generally, the number of variables in the matrix coincides with Tatta number of edges of the graph.

Tutte matrix is antisymmetric (skew-symmetric).

## Tutte's theorem

Consider the determinant of the  $\det(A)$  matrix of Thatta. It is, generally speaking, a polynomial in the variables  $x_{ij}$ .

**Tutte's theorem** states that in the graph  $G$ , there is a perfect matching if and only if the polynomial  $\det(A)$  is not identically zero (ie, has at least one term with non-zero coefficient). Recall that the matching is called perfect if it saturates all the vertices, ie, his power is  $n/2$ .

Canadian mathematician William Thomas Tutte (William Thomas Tutte) first pointed out the close relationship between matchings in graphs and determinant of the matrix (1947). A simpler form of this connection later discovered Edmonds (Edmonds) in the case of bipartite graphs (1967). Randomized algorithms for finding maximum matching values themselves edges of this matching have been proposed later, respectively, Lovas (Lovasz) (1979), and Rabin (Rabin) and Vazirani (Vazirani) (1984).

## Practical application: a randomized algorithm

Directly apply the theorem of Tutte even in the problem of testing the existence of a perfect matching inappropriate. The reason for this is that when the determinant of symbolic computation (ie, in the form of polynomials over variables  $x_{ij}$ ) are polynomials of intermediate results that contain  $O(n^2)$  variables. Therefore, the calculation of the determinant of Thatta in symbolic form requires an inordinate amount of time.

Hungarian mathematician László Lovász (Laszlo Lovasz) was the first to indicate the possibility of applying the **randomized** algorithm to simplify the calculations.

The idea is very simple: replace all variables  $x_{ij}$  with random numbers:

$$x_{ij} = \text{rand}().$$

Then, if the polynomial  $\det(A)$  is identically zero was, after such a change and he will remain zero; if it was different from zero, when a random number replacing the likelihood that it will turn to zero sufficiently small.

It is clear that such a test (random permutation of the values and the calculation of the determinant  $\det(A)$ ) and if wrong, it is only one way: can report no perfect

matching, when in fact it exists.

We can repeat the test several times, substituting the values of variables as random numbers, and with every restart we get more and more confident that the test gave the correct answer. In practice, in most cases, only one test to determine if there is a perfect matching in a graph or not; Several such tests have given a very high probability.

To estimate the **probability of error**, you can use the Schwarz lemma-Sippel (Schwartz-Zippel), which states that the probability of vanishing nonzero polynomial  $th$  degree by substituting the values of variables as random numbers, each of which can take the value of options - this probability satisfies inequality:  $P_{ks}$

$$\Pr[P(r_1, \dots, r_k) = 0] \leq \frac{k}{s}.$$

For example, when using standard random number function C ++ `rand()` obtain that this probability  $n = 300$  is about one percent.

**Asymptotic behavior of solutions** is found to be  $O(n^3)$  (using, for example, [Gauss](#)), multiplied by the number of iterations of the test. It is worth noting that the asymptotic behavior of this solution is significantly lower than the solution [algorithm of Edmonds compression flowers](#), but in some cases more preferred because of its ease of implementation.

**Restore** itself as a perfect matching set of edges is more difficult. The simplest, albeit slow, recovery of this option would be matching one edge: iterate through the first edge response, we choose it so that in the remaining graph there a perfect matching, etc.

## The proof of Tutte

To understand well the proof of this theorem, we first consider a simpler result, - Edmonds obtained for the case of bipartite graphs.

### Theorem Edmonds

Consider the bipartite graph, in which each share on  $n$  peaks. Form the matrix in which, by analogy with the matrix Tatta, a separate independent variable, if an edge is present in the column, and is identical to zero otherwise.  $B n \times nb_{ij}(i, j)$

This matrix similar to the matrix of Thatta, but Edmonds matrix has half the difference, and each edge here corresponds to only one cell of the matrix.

We prove the following **theorem**: the determinant  $\det(B)$  is nonzero if and only if there exists a bipartite graph a perfect matching.

**Proof**. We write the determinant according to its definition, as the sum over all permutations:

$$\det(B) = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) \cdot b_{1,\pi_1} \cdot b_{2,\pi_2} \cdot \dots \cdot b_{n,\pi_n}.$$

Note that since all non-zero elements of the matrix  $B$ - the various independent variables, in this sum all nonzero terms are different, and therefore no shortcuts in the process of summation occurs. It remains to note that any non-zero term in this sum is disjoint set top edges, 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 antisymmetric matrices

To prove the theorem of Tutte is necessary to use several well-known facts of linear algebra on the properties of the antisymmetric matrices.

**Firstly**, if the antisymmetric matrix has an odd size, its determinant is always zero (Theorem Jacobi (Jacobi)). It is sufficient to note that the antisymmetric matrix satisfies  $A^T = -A$ , and now we get a chain of equalities:

$$\det(A) = \det(A^T) = \det(-A) = (-1)^n \det(A),$$

from which it follows that for odd  $n$  determinant must be zero.

**Secondly**, it appears that in the case of antisymmetric matrices of even the size of their determinant can always be written as the square of a polynomial in the variables elements of this matrix (polynomial is called the Pfaffian (pfaffian), and the result belongs Muir (Muir)):

$$\det(A) = \operatorname{Pf}^2(A).$$

**Thirdly**, the Pfaffian is not an arbitrary polynomial, and the sum of the form:

$$\operatorname{Pf}(A) = \frac{1}{2^n n!} \sum_{\pi \in S_n} \operatorname{sgn}(\pi) \cdot a_{\pi_1,\pi_2} \cdot a_{\pi_3,\pi_4} \cdot \dots \cdot a_{\pi_{n-1},\pi_n}.$$

Thus, each term in the Pfaffian - is the product of  $n/2$  the elements of their indices together represent a partition of  $n$  at  $n/2$  par. Before each term has a factor, but it is kind of not concern us here.

### The proof of Tutte

Using the second and third property from the previous paragraph, we see that the determinant  $\det(A)$  of the matrix Thatta is a square of the sum of the 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  $n$ . Thus, again, as in the proof of Edmonds, each non-zero term of this sum corresponds to a perfect matching in a graph, and vice versa.

# Lovász theorem: generalization to find the size of the maximum matching

## Formulation

Rank Tutte matrix coincides with twice the size of the maximum matching in this graph.

## Application

For use in practice of this theorem we can use the same reception randomization that the above-described algorithm for the matrix Tatta, namely variables to substitute random values and find the resulting numerical rank matrix. The rank of the matrix, again searched for  $O(n^3)$  using the modified Gauss. See [here](#).

However, it should be noted that the above cited lemma Schwarz-Sippel inapplicable explicitly, and intuitively it seems that the probability of error is higher here. However, allegedly (see Ref. Lovász (Lovasz)), that here the probability of error (ie, the fact that the rank of the resulting matrix will be less than twice the size of the maximum matching) does not exceed  $\frac{n}{s}$  (where  $s$ , as above, denotes the size of the set, from which chooses a random number).

## Proof

The proof will follow from the theorem of linear algebra, known as the **Frobenius theorem** (Frobenius). Suppose we are given an antisymmetric matrix  $A$  size  $n \times n$ , and the empty set  $S$  and  $T$ - any two subsets  $\{1, \dots, n\}$ , with the size of these sets coincide and are equal to the rank of a matrix  $A$ . Denoted by  $A_{XY}$  the matrix obtained from the matrix  $A$  only rows with indices from the set  $X$  and columns with numbers from the set  $Y$  (where  $X$  and  $Y$ - some subsets  $\{1, \dots, n\}$ ). Then the following:

$$\det(A_{SS}) \cdot \det(A_{TT}) = \det(A_{ST}) \cdot \det(A_{TS}).$$

We show how this property allows you to set **correspondence** between the rank of the matrix  $A$  Thatta and value of the maximum matching.

On the one hand, consider the graph in  $G$  a maximal matching, and denote the set of vertices saturable them through  $U$ . Then, according to the theorem of Tutte, the determinant  $\det(A_{UU})$  is nonzero. The investigator, the rank of Thatta - at least  $2|U|$ , that is not less than twice the size of the maximum matching.

We now show the reverse inequality. We denote the rank of the matrix  $A$  through  $r$ . This means that they found such a submatrix  $A_{ST}$  where  $|S| = |T| = r$  the determinant is nonzero. Easy to see that  $A_{TS}$  also non-zero. But found above Frobenius theorem, this means that both the matrix  $A_{SS}$  and  $A_{TT}$  have a non-

zero determinant. It follows that  $r$  is even (because, as noted above, the antisymmetric matrix odd dimension always has zero determinant). Thus, we can apply to the submatrix  $A_{SS}$ (or  $A_{TT}$ ) Tutte theorem. Consequently, in the subgraph induced by the set of vertices  $S$ (or set of vertices  $T$ ), there is a perfect matching (and its magnitude is equal  $r/2$ ). Thus, the rank of Thatta not be more than twice the size of the maximum matching.

By combining two proven inequality, we obtain the theorem: the rank of Thatta coincides with twice the size of the maximum matching.

## 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 the maximum matching, but they themselves edges coming into it.

### Statement of the theorem

Let the graph, there is a perfect matching. Then it Tutte matrix is nonsingular, ie  $\det(A) \neq 0$ . Generate over it, as described above, the random number matrix  $B$ . Then, with a high probability  $(B^{-1})_{ji} \neq 0$  if and only if the edge  $(i, j)$  is included in any perfect matching.

(Here  $B^{-1}$ denotes the matrix inverse  $B$ . It is assumed that the determinant of the matrix  $B$  is non-zero, so the inverse matrix exists.)

### Application

This theorem can be used to restore themselves edges maximal matching. You will first need to allocate a subgraph, which contains the desired maximal matching (this can be done in parallel with the rank of the search algorithm).

After this problem is reduced to finding a perfect matching on this numerical matrix obtained from the matrix Tutte. Here we already apply Theorem Rabin-Vazirani - find the inverse matrix (which can be done using the modified algorithm for Gaussian  $O(n^3)$ ), we find in it any non-zero element is removed from the graph, and repeat the process. Asymptotic behavior of such a decision would not be the fastest -  $O(n^4)$ , but instead we get simple solutions (compared, for example, with a compression algorithm Edmonds flowers ).

### The proof of Theorem

Let us recall the well-known formula for the elements of the inverse matrix  $B^{-1}$ :

$$(B^{-1})_{ji} = \frac{\text{adj}(B)_{i,j}}{\det(B)},$$

through which  $\text{adj}(B)_{i,j}$  indicated the cofactor, ie this number is  $(-1)^{i+j}$  multiplied by the determinant of the matrix obtained from the  $B$  removal of the  $i$ th row and  $j$ th column.

Hence we immediately obtain that the element  $(B^{-1})_{ji}$  is nonzero if and only if the matrix  $B$  with a strike-  $i$ th row and the  $j$ -th column has a non-zero determinant that, applying the theorem of Tutte, is 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, MD Plummer. **Matching Theory** [1986]
  - Michael Oser Rabin, Vijay V. Vazirani. **Maximum matchings in graphs through Randomization General** [1989]
  - Allen B. Tucker. **Computer Science Handbook** [2004]
  - Rajeev Motwani, Prabhakar Raghavan. **Randomized Algorithms** [1995]
  - AC Aitken. **Determinants and matrices** [1944]
-

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 10 Jun 2008 23:07  
 EDIT: 31 Aug 2011 21:57

## Edged connectivity. Properties and finding

### Determination

Suppose we are given an undirected graph  $G$  with  $n$  vertices and  $m$  edges.

**Costal connection**  $\lambda$  graph  $G$  is the smallest number of edges that must be removed to Count ceased to be connected.

For example, for a disconnected graph edged connectivity is zero. For a connected graph with a single edged bridge the connection is equal to one.

A set  $S$  of edges **shared** vertices  $s$  and  $t$ , if you remove these edges from the graph vertices  $u$  and  $v$  are in different connected components.

It is clear that edged the connectivity graph is minimized by the least number of edges separating two vertices  $s$  and  $t$ , taken among all possible pairs  $(s, t)$ .

### Contents [hide]

- [Edged connectivity. Properties and finding](#)
  - [Determination](#)
  - [Properties](#)
    - [Whitney ratio](#)
    - [Ford-Fulkerson theorem](#)
  - [Finding edged connectivity](#)
    - [A simple algorithm based on the maximum flow search](#)
    - [A special algorithm](#)
  - [Literature](#)

### Properties

#### Whitney ratio

**Ratio Whitney (Whitney) (1932)** between the rib connection  $\lambda$ , [vertex connectivity](#)  $\kappa$  and the lower of the vertex degrees  $\delta$ :

$$\kappa \leq \lambda \leq \delta.$$

We prove this assertion.

We first prove the first inequality  $\kappa \leq \lambda$ . Consider the set of  $\lambda$  edges, making the graph disconnected. If we take from each of the ribs on one end (either of the two), and remove from the graph, thus using the  $\leq \lambda$  remote peaks (as one and the same vertex could meet twice) we will make the graph disconnected. Thus  $\kappa \leq \lambda$ .

We prove the second inequality  $\lambda \leq \delta$ . Consider a vertex of minimum degree, then we can remove all the  $\delta$  edges adjacent to it, and thus to separate it from the rest of the top of the graph. Hence  $\lambda \leq \delta$ .

Interestingly, the Whitney inequality **can not be improved** : ie for any three numbers satisfying this inequality, there is at least one corresponding graph. See. The task "[Construction of the graph with the specified value of the vertex and the rib connections and the lower of the vertex degrees](#)".

#### Ford-Fulkerson theorem

**Ford-Fulkerson theorem (1956):**

For any two vertices of the largest number of edge-disjoint chains connecting them, is the smallest number of edges separating these peaks.

## Finding edged connectivity

### A simple algorithm based on the maximum flow search

This method is based on the theorem of Ford-Falekrsone.

We have to iterate over all pairs of vertices ( $s, t$ ), and between each pair to find the largest number of disjoint paths in the ribs. This value can be found using the maximum flow algorithm: we do  $s$ source,  $t$  - the drain, and the capacity of each edge we place equal to 1.

Thus, pseudocode algorithm is as follows:

```
int ans = INF;
for (int s=0; s<n; ++s)
    for (int t=s+1; t<n; ++t) {
        int flow = ... величина максимального потока из s в t ...
        ans = min (ans, flow);
    }
```

Asymptotic behavior of the algorithm using \ edmonds\_karp {Edmonds-Karp algorithm for finding the maximum flow} is obtained  $O(n^2 \cdot nm^2) = O(n^3m^2)$ , however, it should be noted that hidden in the asymptotic constant is very small, since it is practically impossible to create such a graph algorithm to find the maximum flow slowly worked once for all source and drain.

Especially quickly this algorithm will work on random graphs.

### A special algorithm

Using streaming terminology, this problem - it is the task of finding **the global minimum cut**.

For its solution developed special algorithms. This site presents one of which - Curtains, Wagner algorithm working in time  $O(n^3)$  or  $O(nm)$ .

## Literature

- Hassler Whitney. **Congruent Graphs and the Connectivity of Graphs** [1932]
- Frank Harari. **Graph Theory** [2003]

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 10 Jun 2008 23:08  
EDIT: 10 May 2012 23:22

## Vertex connectivity. Properties and finding

### Contents [hide]

- Vertex connectivity. Properties and finding
  - Determination
  - Properties
    - Whitney ratio
  - Finding the vertex connectivity

### Determination

Suppose we are given an undirected graph  $G$  with  $n$  vertices and  $m$  edges.

**Vertex connectivity**  $\lambda$  of the graph  $G$  is the smallest number of vertices to be deleted to make the graph cease 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 is one. For a complete graph vertex connectivity is assumed to be  $n - 1$  (because some pair of vertices we may choose, even the removal of all the remaining vertices do not make them disconnected). For all graphs, but complete, vertex connectivity at most  $n - 2$  - because you can find a pair of vertices, between which there is no edge, and remove all other  $n - 2$  vertices.

A set  $S$  of vertices **shared** vertices  $s$  and  $t$ , if you delete these vertices from 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

**Ratio Whitney (Whitney) (1932)** between the **rib connection**  $\lambda$ , vertex connectivity  $\kappa$  and the lower of the vertex degrees  $\delta$ :

$$\kappa \leq \lambda \leq \delta.$$

We prove this assertion.

We first prove the first inequality  $\kappa \leq \lambda$ . Consider the set of  $\lambda$  edges, making the graph disconnected. If we take from each of the ribs on one end (either of the two), and remove from the graph, thus using the  $\leq \lambda$  remote peaks (as one and the same vertex could meet twice) we will make the graph disconnected. Thus  $\kappa \leq \lambda$ .

We prove the second inequality  $\lambda \leq \delta$ . Consider a vertex of minimum degree, then we can remove all the  $\delta$  edges adjacent to it, and thus to separate it from the rest of the top of the graph. Hence  $\lambda \leq \delta$ .

Interestingly, the Whitney inequality **can not be improved** : ie for any three numbers satisfying this inequality, there is at least one corresponding graph. See. The task "[Construction of the graph with the specified value of the vertex and the rib connections and the lower of the vertex degrees](#)" .

## Finding the vertex connectivity

Brute over a pair of vertices  $s$  and  $t$  and find the minimum number of vertices to be deleted to share  $s$  and  $t$ .

For this **bifurcated** each vertex: ie each vertex  $i$  will create two copies - one  $i_1$  for incoming edges, the other  $i_2$  - for the outgoing and the two copies are related to each other by an edge  $(i_1, i_2)$ .

Each edge  $(u, v)$  in the original graph of the modified network will turn into two edges:  $(u_2, v_1)$  and  $(v_2, u_1)$ .

All edges places a bandwidth equal to one. We now find the maximum flow in the graph between the source  $s$  and the drain  $t$ . By the construction of the graph, it will be the minimum number of vertices required for separation  $s$  and  $t$ .

Thus, if the search for the maximum flow algorithm we choose the [Edmonds-Karp](#), who works for the time  $O(nm^2)$ , the total amount to the asymptotic behavior of the algorithm  $O(n^3m^2)$ . However, the constant hidden in the asymptotic behavior is quite low as to make a graph on which algorithms have been working for a long time at any pair of source-drain is almost impossible.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 23:10  
EDIT: 31 Aug 2011 23:05

## Graphing with the specified vertex and rib connections and the lower of the degrees of the vertices

### Contents [hide]

- Graphing with the specified vertex and rib connections and the lower of the degrees of the vertices
  - Whitney ratio
  - Solution

Given magnitude  $\kappa, \lambda, \delta$ - are, respectively, the vertex connectivity , edged connectivity and the smallest of the degrees of vertices in the graph. Required to construct a graph, which would have the specified values, or to say that this graph does not exist.

### Whitney ratio

**Ratio Whitney (Whitney) (1932)** between the rib connection  $\lambda$  , vertex connectivity  $\kappa$  and the lower of the vertex degrees  $\delta$ :

$$\kappa \leq \lambda \leq \delta.$$

We prove this assertion.

We first prove the first inequality  $\kappa \leq \lambda$ . Consider the set of  $\lambda$ edges, making the graph disconnected. If we take from each of the ribs on one end (either of the two), and remove from the graph, thus using the  $\leq \lambda$ remote peaks (as one and the same vertex could meet twice) we will make the graph disconnected. Thus  $\kappa \leq \lambda$ .

We prove the second inequality  $\lambda \leq \delta$ . Consider a vertex of minimum degree, then we can remove all the  $\delta$ edges adjacent to it, and thus to separate it from the rest of the top of the graph. Hence  $\lambda \leq \delta$ .

Interestingly, the Whitney inequality **can not be improved** : ie for any three numbers satisfying this inequality, there is at least one corresponding graph. This we prove constructively by showing how to construct the corresponding graphs.

### Solution

Check whether the number of data  $\kappa$ ,  $\lambda$  and  $\delta$  the ratio of Whitney. If not, then there is no answer.

Otherwise, we construct the graph itself. It will consist of  $2(\delta + 1)$  vertices, with the first  $\delta + 1$  vertex subgraph fully meshed, and the second  $\delta + 1$  vertex subgraph fully meshed. In addition, we join these two parts  $\lambda$  ribs so that the first part of these edges are adjacent  $\lambda$  vertices, and on the other side -  $\kappa$  the top. It is easy to verify that the resulting graph will have the necessary characteristics.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 23:10  
EDIT: 10 Jun 2008 23:11

## The inverse problem of SSSP (inverse-SSSP - inverse problem of shortest paths from one vertex)

There is a weighted undirected multigraph G from N vertices and M edges. Given an array P [1..N] and select an initial vertex S. Requires change the weights of edges so that all IP [I] was equal to the length of the shortest path from S to I, moreover the sum of all changes (sum of absolute changes in the weights of edges) would be lower. If this is not possible, then the algorithm should output "No solution". Do negative edge weight is prohibited.

### Description of the solution

We will solve this problem in linear time, just to sort out all the edges (ie, in a single pass).

Let the current step, we consider an edge from vertex A to vertex B length R. We assume that the vertex A has all the conditions are met (ie, the distance from S to A is really equal to P [A]), and will check that the conditions for the vertex B. We have several options for the situation:

- 1.  $P[A] + R < P[B]$

This means that we have found a way shorter than it should be. Since P [A] and P [B] we can not change, then we have to extend the current edge (independently from the rest of the ribs), namely to perform:

$$R = P[B] - P[A] - R.$$

In addition, this means that we have already found the way to the top of B from S, the length of which is equal to the desired value P [B], so the next steps we will not have to shorten any ribs (see. Option 2).

- 2.  $P[A] + R >= P[B]$

This means that we have found a way longer than required. As such there may be several ways, we have to choose among all these paths (edges) then that will require the least changes. Again, if we extend some edge going into the top of B (option 1), then, we actually built the shortest path to the top of B, and therefore shorten the no edge will not have to.

Thus, for each vertex, we should keep an edge that is going to be shortened, ie edge with the smallest weight changes.

Thus, simply cycle through all the edges, and having considered the situation for each edge ( $O(1)$ ), we will solve the inverse problem SSSP in linear time.

If at any point we are trying to change has altered the edge, it is obvious that this can not be done, and should issue a "No solution". Furthermore, some peaks can not be achieved and the required estimate of the shortest path, then the answer is also "No solution". In all other cases (except, of course, is clearly incorrect values in the array P, that  $P[S]! = 0$  or negative) answer will be.

### Implementation

The program displays "No solution", if there is no solution, otherwise displays the first line of a minimum amount of changes in the weights of edges, and in the next M lines - new weights of edges.

```
const int INF = 1000 * 1000 * 1000;
int n, m;
```

### Contents [hide]

- The inverse problem of SSSP (inverse-SSSP - inverse problem of shortest paths from one vertex)
  - Description of the solution
  - Implementation

```

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) with a costs 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]);
}

```

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 10 Jun 2008 23:11  
EDIT: 13 Jun 2008 1:31

## The inverse problem of MST (inverse-MST - inverse problem minimum spanning tree) for O (NM<sup>2</sup>)

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 the graph (ie, selected N-1 edges that form a tree with N vertices). You want to change the weight of the edges so that the specified frame T is a minimum spanning tree of the graph (or more precisely, one of the minimum spanning tree), and to do so, so that the total change of all was the smallest scales.

### Solution

We reduce the problem of inverse-MST to the problem min-cost-flow, more precisely, to the problem of dual-min cost-Flow (in the sense of duality in linear programming); then solve the latter problem.

Thus, suppose given a graph G with N vertices, M edges. Weight of each edge is denoted by  $C_{ij}$ . Assume without loss of generality that the edges with numbers from 1 to N-1 are the edges T.

#### 1. A necessary and sufficient condition for MST

Let there be given a skeleton S (not necessarily minimal).

We introduce the first notation. Consider an edge j, does not belong to S. Obviously, the graph S has a single path connecting the ends of the edge, ie, the only path connecting the ends of the edge j and consisting only of edges belonging to S. Let  $P[j]$  the set of edges that form a path for the j-th rib.

To some skeleton S is minimal, necessary and sufficient to:

$$C_{ij} \leq C_{kj} \text{ for all } j \notin S \text{ and every } i \in P[j]$$

It can be noted that, since in our problem skeleton T belong ribs 1..N-1, we can write this condition as follows:

$$C_{ij} \leq C_{kj} \text{ for all } j = N..M \text{ and each } i \in P[j]$$

(And all i are in the range 1..N-1)

#### 2. Count the ways

Count the ways the concept is directly related to the preceding theorem.

Let there be given a skeleton S (not necessarily minimal).

Then the graph H ways for a graph G is the following graph:

- It contains M vertices, each vertex in H one-to-one correspondence to an edge in G.
- Bipartite graph H. The first of its share of the vertices are i, which correspond to edges in G, belonging to the skeleton S. Accordingly, in the second part are the vertices j, which correspond to edges not belonging to S.
- Rib held from vertex i to vertex j if and only if i belongs to  $P[j]$ .  
In other words, for each vertex j in the second part it includes the edges of all the vertices of the first part corresponding to a plurality of ribs  $P[j]$ .

In the case of our problem , we can simplify the description a little ways graph:

### Contents [hide]

- The inverse problem of MST (inverse-MST - inverse problem minimum spanning tree) for O (NM<sup>2</sup>)
  - Solution
    - 1. A necessary and sufficient condition for MST
    - 2. Count the ways
    - 3. Mathematical formulation of the problem
    - 4. Reduction of the inverse-MST to the problem, the dual assignment problem
    - 5. The solution of the dual problem of appointments
    - 6. A modified algorithm for min-cost-flow solutions for the assignment problem
    - 6. Summary
  - Implementation

an edge  $(i, j)$  exists in the  $H$ , if  $i \in P[j]$ ,  $j = N..M$ ,  $i = 1..N-1$

### 3. Mathematical formulation of the problem

Formally challenge inverse-MST is written as follows:

```
find an array A [1..M] such that
Ci + Ai <= Cj + Aj for all j = N..M and each i ∈ P[j] (i in 1..N-1)
and to minimize the sum | A1 | + | A2 | + ... + | Am |
```

here under the desired array A, we mean those values to be added to the weights of edges (ie, solving the problem of inverse-MST, we replace the weight of  $C_i$  of each rib on the value of  $i C_i + A_i$ ).

Obviously, it makes no sense to increase the weight of edges belonging to  $T$ , ie,

```
Ai <= 0, i = 1..N-1
```

and it makes no sense to shorten the edges not belonging to  $T$ :

```
Ai >= 0, i = N..M
```

(Because otherwise we will only worsen the answer)

Then we can have a **simplified** formulation of the problem by removing the sum of modules:

```
find an array A [1..M] such that
Ci + Ai <= Cj + Aj for all j = N..M and each i ∈ P[j] (i in 1..N-1)
Ai <= 0, i = 1..N-1
Ai >= 0, i = N..M,
and minimize the amount of An + ... + Am - (A1 + ... + An-1)
```

Finally, just change the "minimizing" 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 in 1..N-1)
Ai <= 0, i = 1..N-1
Ai >= 0, i = N..M,
and maximize the sum A1 + ... + An-1 - (An + ... + Am)
```

### 4. Reduction of the inverse-MST to the problem, the dual assignment problem

Formulation of the problem inverse-MST, which we have just given, is a formulation of the problem of **linear programming** with unknown  $A_1..A_m$ .

Applying the classical technique - consider the **dual** problem to her.

By definition, to get the dual problem, you need to compare each inequality dual variables  $X_{ij}$ , interchanged objective function (which had to be minimized) and the coefficients in the right-hand side, change the signs " $\leq$ " to " $>=$ ", and vice versa, change the maximization to minimize.

Thus, the **dual to the inverse-MST** problem:

```
Find all Xij for each (i, j) ∈ H, such that:
All Xij >= 0,
for each i = 1..N-1 Xij for all j: (i, j) ∈ H <= 1,
each j = ∑ N..M Xij for all i: (i, j) ∈ H <= 1,
X and minimize ∑ij (Cj - Ci) for all (i, j) ∈ H
```

The last problem is the **assignment problem**: we need a graph  $H$  ways to select multiple edges so that no

edge intersects with the other at the top, and the sum of the weights of edges (edge weight  $(i, j)$ ) is defined as  $C_j - C_i$  must be lower.

Thus, **the dual problem is equivalent to the inverse-MST assignment problem**. If we learn to solve the dual problem of appointments, we will automatically solve the problem of inverse-MST.

## 5. The solution of the dual problem of appointments

First, we'll pay some attention to the particular case of the assignment problem, which we got. Firstly, this unbalanced assignment problem, since one is N-lobe peaks 1 and the other - M vertices, i.e. in general, the number of vertices in the second part of the whole procedure more. To solve such a dual assignment problem has specialized algorithm that will solve it for  $O(N^3)$ , but here, this algorithm will not be considered. Second, such a task assignment can be called assignment problem with weighted vertices: the weight of edges set equal to 0, the weight of each vertex of the first part is set equal to  $-C_i$ , from the second part - equal to  $C_j$ , and the resulting solution of the problem will be the same most.

We will solve the problem of dual assignment problem using **a modified algorithm min-cost-Flow**, which will be the flow of minimum cost and at the same time the solution of the dual problem.

**Reduce** the problem of appointments to the problem min-cost-flow very easily, but for the sake of completeness, we describe the process.

Add to 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. From each vertex of the second part will hold an edge to t with bandwidth = 1 and the value of 0. capacity of all edges between the first and second installments as set equal to 1.

Finally, the modified algorithm for min-cost-flow (described below) to work, you need to **add an edge from s to t** with capacity =  $N + 1$  and the value = 0.

## 6. A modified algorithm for min-cost-flow solutions for the assignment problem

Here we consider the **algorithm of successive shortest paths with potentials** that resembles the usual algorithm min-cost-flow, but also uses the concept of **potential**, which by the end of the algorithm will contain **a solution of the dual problem**.

We introduce the notation. For each edge  $(i, j)$  is denoted by  $U_{ij}$  its capacity through  $C_{ij}$  - its value through  $F_{ij}$  - flow along this edge.

Also, we introduce the concept of potential. Each vertex has its own potential  $\Pi_i$ . The residual value of CPI edges  $_{ij}$  is defined as:

$$\text{CPI}_{ij} = C_{ij} - \Pi_i + \Pi_j$$

At any time of the algorithm **are the potentials** that the following conditions:

```
if  $F_{ij} = 0$ , then the  $\text{CPI}_{ij} >= 0$ 
if  $F_{ij} = U_{ij}$ , then the  $\text{CPI}_{ij} <= 0$ 
otherwise  $\text{CPI}_{ij} = 0$ 
```

The algorithm starts with zero flow, and we need to find some initial values of potentials that satisfy the specified conditions. It is easy to verify that this method is one of the possible solutions:

```
 $\Pi_j = 0$  for  $j = N..M$ 
 $\Pi_i = \min C_{ij}$ , where  $(i, j) \in H$ 
 $\Pi_s \Pi_i = \min_i$ , where  $i = 1..N-1$ 
 $\Pi_t = 0$ 
```

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, with the weights of edges using the residual value of CPI. Then we increase the flow along the path found by one, and update capabilities as follows:

$$\Pi_i - = D_i$$

where  $D_i$  - found the shortest distance from s to i (again, in the residual network with weights of edges 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** work of 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 the solution of the assignment problem (in the form of a flow  $F_{ij}$ ) and the solution of the dual assignment problem (in the array  $PI_i$ ).

(With  $PI_i$  will have to spend a small modification: all values of  $PI_i$  take  $PI_S$ , because its values are valid only for  $PI_S = 0$ )

## 6. Summary

So, we decided to dual assignment problem, and therefore the task of inverse-MST.

We estimate the **asymptotic behavior** of the resulting algorithm.

First, we will have to construct a graph of ways. To do this simply for each edge  $j \notin T$  preorder traversal to find the path skeleton  $T P[j]$ . Then we construct a graph of the ways of  $O(M) * O(N) = O(NM)$ .

Then we find the initial values of the potentials of  $O(N) * O(M) = O(NM)$ .

Then we iterate min-cost-flow, all iterations will be no more than  $N$  (because the source of the yield of  $N$  edges, each with a capacity of = 1) at each iteration we are looking at ways to graph the shortest path from the source to all other nodes. Since the vertices in the graph paths is  $M + 2$ , and the number of edges -  $O(NM)$ , then if the search of the shortest paths to realize the simplest version of Dijkstra's algorithm, each iteration of the min-cost-flow will perform for the  $O(M^2)$ , 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 of the above algorithm. The only change - instead of [Dijkstra's algorithm](#) is used [algorithm of Leviticus](#), which in many tests should run a bit 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 adjacency lists
    vector <rib2> ribs (m); // All edges in one list
    ... Read Count ...

    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 ...

}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2008 23:14  
EDIT: 10 Jun 2008 23:16

## Paint the edges of the tree

This is a fairly common tasks. Given a tree G. request comes in two forms: the first type - paint an edge, the second view - a request the number of colored edges between two vertices.

There will be described fairly simple solution (using [wood pieces](#) ), which will respond to requests for  $O(\log N)$ , with preprocessing (pre-treatment of wood) for  $O(M)$ .

### Contents [hide]

- Paint the edges of the tree
  - Solution
  - Implementation

### Solution

First, we have to implement [LCA](#) , to every request of the second type  $(i, j)$  is reduced to two requests  $(a, b)$ , where  $a$  - the ancestor of  $b$ .

We now describe the [preprocessing](#) itself for our problem. Run the dfs root of the tree, the depth-first search will be a list of visits vertices (each vertex is added to the list when it comes to the search for, and every time after dfs son returns from the current vertex) - incidentally, The same list is used by the algorithm LCA. In this list, each edge is present (in the sense that if  $i$  and  $j$  - ends of the ribs in the list is required to find a place where  $i$  and  $j$  are consecutive to each other), moreover 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  $i$ ).

We construct two tree lengths (for the amount of unit modification) on the 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 received another [request](#) the form  $(i, j)$ , where  $i$  - the ancestor of  $j$ , and you want to determine how many edges painted on the path between  $i$  and  $j$ . Let's find  $i$  and  $j$  in the list traversal depth (we definitely need a position where they meet for the first time), let it some position  $p$  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  $T1[p..q-1]$  - the sum of  $T2[p..q-1]$** .

How Come? Consider the segment  $[p; q]$  in the list traversal depth. It contains edges necessary to us the way from  $i$  to  $j$ , but also contains the set of edges that lie on other ways of  $i$ . However, between the ribs necessary to us and the rest of the ribs there is one big difference: the right edges will be contained in the list only once, and in the forward direction, and all other edges will meet twice, both literally and in the opposite direction. Consequently, the difference  $T1[p..q-1] - T2[p..q-1]$  will give us the answer (minus one need, because otherwise we will take more extra edge from the vertex  $j$  somewhere up or down). Request amounts tree segments is performed in  $O(\log N)$ .

The answer to the [query](#) of the form 1 (a painting of 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 (edge to find a list traversal, again, it is possible in  $O(1)$  if you do this search in the preprocessing). A single modification in the tree segments is performed in  $O(\log N)$ .

### Implementation

There will be given full implementation of 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 () {
                // Query about painting the edge labeled x;
                // If start = true, then the edge is painted, otherwise the paint is 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 {
                // Query 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 - a response to a request
            }
        }
    }
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: 6 Jul 2008 9:33  
EDIT: 7 Sep 2009 9:59

## Task 2-SAT

Task 2-SAT (2-satisfiability) - is the task of distribution of values of Boolean variables in such a way that they satisfy all constraints.

### Contents [hide]

- Task 2-SAT
  - Apps
  - Algorithm
  - Implementation

2-SAT problem can be represented as 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:

$(A \mid\mid c) \ \&\& \ (a \mid\mid !D) \ \&\& \ (b \mid\mid !D) \ \&\& \ (b \mid\mid !E) \ \&\& \ (c \mid\mid d)$

## Apps

The algorithm for the solution of 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 are links between these values:

- **Location text labels on the map or chart .**  
This refers to the presence of such an arrangement marks in which no two disjoint. It is worth noting that in the general case, where each label can occupy many different positions, we obtain a 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.
- **The fins when drawing the graph .**  
Similar to the previous paragraph, if we restrict ourselves to only two possible ways to hold an edge, we come to the 2-SAT.
- **Scheduling of games .**  
This refers to such a system, in which each team has to play with each other once, and you want to distribute the game by type of home-visiting, with some constraints.
- etc.

## Algorithm

First, we present the problem to another form - the so-called implicative form. Note that the expression of the form  $a \mid\mid b$  is equivalent to  $!A \Rightarrow b$  or  $!B \Rightarrow a$ . This can be interpreted as follows: if there is expression  $a \mid\mid b$ , and we need to make it appeal to true, if  $a = \text{false}$ , you must  $b = \text{true}$ , and vice versa, if  $b = \text{false}$ , then it is necessary  $a = \text{true}$ .

We now construct the so-called **Count implications** : for each variable in the column will be two peaks, which we denote by  $x_+$  and  $!x_+$ . Edges in the graph correspond to implicative relations.

For example, 2-CNF form:

(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 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 is achievable from  $x \neq X$ , 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 should 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 vertex is reachable another, and from one vertex is reachable first, these two vertices are in the same strongly connected component. Then we can formulate **a criterion for the existence of solutions** as follows:

To this problem 2-SAT **has a solution** if and only if for any vertex  $x$  of  $x$  and  $\neq X$  are in **different components of strong connectivity** graph implications.

This criterion can be verified in time  $O(N + M)$  with the help of [the search algorithm is strongly connected components](#).

Now build your own **algorithm for** finding a solution to the problem 2-SAT on the assumption that a solution exists.

Note that, despite the fact that a solution exists, some variables may be performed from that achievable  $x \neq X$ , or (but not both) of  $X$  achievable  $x$ . In this case, the choice of one of the values of  $x$  will lead to a contradiction, while another choice - will not. Learn to choose between two values that which does not lead to contradictions. Immediately, we note that by selecting a value, we have to run from it crawl in depth / width and mark all the values that follow from it, ie, achievable in the graph implications. Accordingly, already labeled vertices no choice between  $x$  and  $\neq X$  do not need to have a value already selected and recorded. The procedure described below applies only to untagged more vertices.

**It is alleged** the following. Let  $\text{comp}[v]$  denotes the number of strongly connected components, which belongs to the vertex  $v$ , moreover rooms are arranged in order of topological sorting component strongly connected components in the graph (ie, earlier in the order of topological sort correspond to large numbers: if there is a path from  $v$  to  $w$ , then  $\text{comp}[v] \leq \text{comp}[w]$ ). Then, if the  $\text{comp}[x] < \text{comp}[\neq X]$ , then select the value  $\neq X$ , otherwise, ie, if the  $\text{comp}[x] > \text{comp}[\neq X]$ , then choose  $x$ .

**We prove** that with this choice of values we come to a contradiction. Suppose, for definiteness, the selected vertices  $x$  (if the selected vertex  $\neq X$ , proved symmetrically).

First, we prove that  $x$  is not achievable  $\neq X$ . Indeed, since the number of components of strong connectivity  $\text{comp}[x]$  more rooms component  $\text{comp}[\neq X]$ , it means that the connected component containing  $x$ , is located to the left of the connected components containing  $\neq X$ , 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  $y$  is 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 property Count implications of!  $Y$  is achievable!  $X$ . But, by hypothesis, of  $y$  is achievable!  $Y$ . Then we get that from  $x$  achievable!  $X$ , 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 vertex  $x$  of  $x$  and!  $X$  are in different components of strong connectivity. Above showed the correctness of the algorithm. Therefore, we at the same time proved the above criterion of existence of solutions.

Now we can assemble **the whole algorithm** together:

- We construct a graph of implications.
- We find in this graph strongly connected components in a time  $O(N + M)$ , let the  $\text{comp}[v]$  - is the number of components of strong connectivity, which belongs to the vertex  $v$ .
- We verify that for each vertex  $x$  of  $x$  and!  $X$  lie in different components, ie  $\text{comp}[x] \neq \text{comp}[\neg x]$ . If this condition is not satisfied, then return "solution does not exist."
- If the  $\text{comp}[x] > \text{comp}[\neg X]$ , then choose the variable  $x$  is true, otherwise - false.

## Implementation

Below is the implementation of the solution of the problem 2-SAT for an already constructed graph implications  $g$  and its inverse graph  $gt$  (ie, in which the direction of each edge is reversed).

The program displays the number 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 () {
    ... Read n, the graph g, the construction of the 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);
    }
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: Sep 6, 2011 1:03  
EDIT: 16 Nov 2011 23:48

## Heavy-light decomposition

**Heavy-light decomposition** - it is a common technique that allows you to effectively solve many problems which reduce to the **needs of the tree**.

The simplest **example of the** problems of this kind - is the next task. Given a tree, each vertex is assigned a certain number.

Receives requests like  $(a, b)$  where  $a$  and  $b$  - number of vertices of the tree, and you want to know the maximum number of nodes on the path between  $a$  and  $b$ .

### Contents [hide]

- Heavy-light decomposition
  - Description of the algorithm
    - Construction of heavy-light decomposition
    - The proof of the correctness of the algorithm
  - Application in solving problems
    - The maximum number of the path between the two vertices
    - Sum of the numbers on the path between two vertices
    - Repainting ribs path between two vertices
  - Tasks in the online judges

## 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 **break the tree on multiple paths** so that for each vertex  $v$  we see that if we go up from  $v$  the root, then we change the path of no more  $\log n$  paths. In addition, all paths should not cross each other in the ribs.

It is clear that if we learn to look for such a decomposition for any tree, this will make any kind of inquiry "to learn something in the way of  $a$  to  $b$ " several requests like "learn something on a segment -th path. " $[l; r]k$

## Construction of heavy-light decomposition

Count for each vertex of  $v$  the size of its subtree  $s(v)$  (ie, the number of vertices in the subtree peaks  $v$ , including the very top).

Next, look at all the edges, leading to the sons of a vertex  $v$ . We call an edge **heavy**, if it leads to the top, such that:  $(v, c) \in$

$$s(c) \geq \frac{s(v)}{2} \Leftrightarrow \text{edge } (v, c) \text{ is heavy.}$$

All other edges will be called **the lungs**. It is obvious that one vertex  $v$  can come

down heavy at most one edge (because otherwise at the top of the two sons would be the size of  $s(v)/2$  that in view of the summit  $v$  gives the size  $2 \cdot s(v)/2 + 1 > s(v)$ , that is a contradiction).

Now we construct itself **decomposed** into disjoint path tree. Consider all the vertices of which does not go down any heavy ribs, and we will go from each of them up until we reach the root of the tree or not will walk a slight edge. As a result, we get a number of ways - we will show that this is the desired path of heavy-light decomposition.

## The proof of the correctness of the algorithm

First, we note that the resulting algorithm will be the way **disjoint**. In fact, if two any way have a common edge, it would mean that some of the peaks coming down the two heavy ribs, which can not be.

Secondly, we show that down from the root of the tree to an arbitrary vertex, we **will change the way for no more  $\log n$  paths**. Indeed, extending down to easy rib reduces the size of the current subtree more than doubled

$$s(c) < \frac{s(v)}{2} \Leftrightarrow \text{edge } (v, c) \text{ is light.}$$

Thus, we could not pass more  $\log n$  light edges. However, the transition from one track to another, we can only through a slight edge (as each way, apart from ending at the root, contains a slight edge in the end, and get right in the middle of the path, we can not).

Therefore, the path from the root to any node, we can not change more  $\log n$  ways, as required.

## Application in solving problems

In solving problems is sometimes convenient to consider the heavy-light as a set of **vertex-disjoint** paths (not disjoint edges). It's enough of each path, delete the last rib, if it is a slight edge - then no properties are not violated, but now each vertex belongs to exactly one way.

Below we look at some typical problems that can be solved with the help of heavy-light decomposition.

We should also pay attention to the problem of **the sum of the numbers on the way**, as an example of a problem that can be solved and simpler techniques.

## The maximum number of the path between the two vertices

Given a tree, each vertex is assigned a certain number. Receives requests like  $(a, b)$  where  $a$  and  $b$  - number of vertices of the tree, and you want to know the maximum number of nodes on the path between  $a$  and  $b$ .

We construct a pre-heavy-light decomposition. Get over each construct by [segment tree for the maximum](#), which will look for the top with the maximum number attributed in this segment for the specified path  $O(\log n)$ . Although the number of paths in the heavy-light decomposition can reach  $n - 1$ , the total size of all the paths is a value  $O(n)$ , and therefore the total size of the trees segments will also be linear.

Now, in order to respond to a query,  $(a, b)$  find the lowest common ancestor  $l$  of these vertices (for example, [by lifting the binary](#)). Now the problem is reduced to two requests:  $(a, l)$  and  $(b, l)$ , for each of which we can answer this way: find, in what way is the lower vertex, make a request to that path, let's move to the top-end of the path, again to determine in which way we were and make a request to him, and so on, until we reach the path containing  $l$ .

Gently should be the case when, for example,  $a$  and  $b$  were in the same way - then the maximum request to this path should be done not on the suffix, and in the domestic subsegments.

Thus, in the process of answering one subquery, we will walk on the  $O(\log n)$  paths, each of them making a request to the maximum suffix or prefix / subsegments (request for prefix / subsegments could be only once).

So we got the solution for  $O(\log^2 n)$  a single request.

If still further predposchitat on each path peaks on all suffixes, you get a solution for  $O(n \log n)$  - as request not high on the suffix occurs only once, when we get to the top  $l$ .

## Sum of the numbers on the path between two vertices

Given a tree, each vertex is assigned a certain number. Receives requests like  $(a, b)$  where  $a$  and  $b$  - number of vertices of the tree, and want to know the sum of the numbers on the path between the vertices  $a$  and  $b$ . The variant of this problem when there are further requests changes in the number attributed to a given vertex.

Although this problem can be solved with the help of heavy-light decomposition, by building on each segment tree for the sum (or simply predposchitav partial sums, if there is no problem of change requests), this problem can be solved by [a simple technique](#).

If the modification request are not available, then find out the amount of the path between two vertices can be parallel with the search for LCA two vertices in a [binary algorithm lifting](#) - just during pre-LCA to count not only  $2^k$ -s ancestors of each vertex, but the sum of the numbers on the way to this ancestor.

There is also a fundamentally different approach to this problem - consider Euler tree traversal, and build a segment tree above him. This algorithm is discussed in [the article to solve the same problems](#). (And if there are no requests modifications - it is enough to do predposchëtom partial sums, without a tree segments.)

Both of these methods provide a relatively simple solutions with the asymptotic

behavior  $O(\log n)$  for a single request.

## Repainting ribs path between two vertices

Given a tree, each edge originally painted white. Receives requests like  $(a, b, c)$  where  $a$  and  $b$  - number of vertices  $c$  - the color, which means that all the edges on the path from  $a$  to  $b$  need to re-paint color  $c$ . Required after all repaintings say how much in the end it turned edges of each color.

The solution - just make [wood pieces with painting on a segment](#) of the set of paths heavy-light decomposition.

Each request repainting on the road  $(a, b)$  will turn into two subqueries  $(a, l)$  and  $(b, l)$  where  $l$  - the lowest common ancestor of vertex  $a$  and  $b$  (found, for example, [the binary algorithm lift](#)), and each of these sub-queries - in  $O(\log n)$  requests to the trees above the track segments.

Total obtain a solution with the asymptotic behavior  $O(\log^2 n)$  for a single request.

## Tasks in the online judges

A list of tasks that can be solved using the 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 "tree or not a tree"](#) [Difficulty: High]
-

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 11 Jun 2008 10:09  
 EDIT: 11 Jun 2008 10:10

## The length of the union of the intervals on the line for O (N log N)

N segments are on the line, ie each segment is given by a pair of coordinates (X<sub>1</sub>, X<sub>2</sub>). Consider the union of these segments and find its length.

The algorithm was proposed Klee (Klee) in 1977. The algorithm works for O (N log N). It has been proven that this algorithm is faster (asymptotically).

### Description

Put all the coordinates of the ends of segments in array X and we can sort it by value coordinates. An additional condition for sorting - with equal coordinates of the first should 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, having a counter C overlapping segments. If C is non-zero, then the result is added to the difference X<sub>i</sub> - X<sub>i-1</sub>. If the current element is to the left end, we increase 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;
}
```

### Contents [hide]

- The length of the union of the intervals on the line for O (N log N)
  - Description
  - Implementation

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 11 Jun 2008 10:11  
 EDIT: 10 Nov 2011 20:42

## Sign area of a triangle and the predicate "Clockwise"

### Determination

Let three points  $p_1, p_2, p_3$ . Let us find the value of the sign area of  $S$  the triangle  $p_1p_2p_3$ , ie the area of this triangle is taken with the plus or minus sign depending on the type of pivot points formed by  $p_1, p_2, p_3$ : clockwise or her accordingly.

It is clear that, if we learn to compute a symbolic ("oriented") area, and we can find the usual area of any triangle, as well as be able to check, clockwise or counterclockwise directed any triple of points.

### Contents [hide]

- Sign area of a triangle and the predicate "Clockwise"
  - Determination
  - Calculation
  - Implementation

### Calculation

We use the concept of **skew** (pseudo-) 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  $\angle(a, b)$  is taken oriented, ie, is the angle between these vectors rotation counterclockwise.

(Module skew product of two vectors is equal to the modulus of the vector product of them.)

Skew product is calculated as the value of the determinant of the coordinates of the 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, getting rid of one multiplication:

$$2S = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

The last formula is convenient to write and memorize 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

The function that computes the area of a 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);
}
```

The function returns to the normal 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;
}
```

The function checks whether the specified forms a triple of 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;
}
```

The function checks whether the specified forms a triple of 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;
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:12  
EDIT: 20 Jun 2011 0:50

## 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. [the corresponding article](#).

### Contents [hide]

- Checking on the intersection of two segments
  - The first way: focused area of a triangle
  - The second method is the intersection of two lines

## The first way: focused area of a triangle

We use the [oriented area of the triangle and the predicate 'Clockwise'](#). Indeed, the segments  $AB$  and  $CD$  overlap, it is necessary and sufficient that point  $A$  and  $B$  were on opposite sides of the line  $CD$ , and, similarly, point  $C$  and  $D$ - on both sides of the line  $AB$ . You can check this by calculating the area of the corresponding triangles oriented and comparing their signs.

The only thing you should pay attention - borderline cases, when some points fall on the very direct. Thus there is only a special case where the above test will give nothing - where both segments lie [on a straight line](#). This case should be considered separately. It is sufficient to verify that the projection of these two segments on the axis  $X$  and  $Y$  cross (often this test is called "test of bounding box").

In general, this method - simpler than the one that is shown below (the intersection of two generating lines), and has fewer special cases, however, its main drawback - in that it does not find a point of intersection itself.

### 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 the area - as it is more "easy" test.

Of course, this code is applicable to the case of real coordinates, just all comparisons with zero should be made by Epsilon (and avoid the multiplication of two real-value `area()` by multiplying instead of their signs).

## The second method is the intersection of two lines

Instead of crossing segments fulfill [the intersection of two straight lines](#), as a result, if the lines are not parallel, we obtain some point you need to check for membership in both segments; it is enough to check that this point belongs to both segments in the projection on the axis  $X$  and on the axis  $Y$ .

If the lines were parallel, then, if they do not match, the segments do not overlap exactly. If direct match, the segments are collinear, and to check their intersection is sufficient to verify that intersect their projection on the axis  $X$  and  $Y$ .

There is still a special case, when one or both of the segment **degenerate** at the point: in this case to talk about direct incorrectly, and this method is not applicable (this case it will be necessary to disassemble separately).

**Implementation** (excluding the case of degenerate segments):

```

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 = a.y-b.y, B1 = b.x-a.x, C1 = -A1*a.x - B1*a.y;
    int A2 = c.y-d.y, B2 = d.x-c.x, C2 = -A2*c.x - B2*c.y;
    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, we first calculate the coefficient  $zn$ - the denominator in the formula Cramer. If  $zn = 0$ , then the coefficients  $A$  and  $B$  are directly proportional, and the lines are parallel or coincide. In this case, it is necessary to check whether or not they are the same, for which it is necessary to verify that the coefficients  $C$  are directly proportional with the same coefficient, which is enough to compute the determinant of the following two, if they are both zero, then the lines are the same:

$$\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}$$

If  $zn \neq 0$ , then the lines intersect, and the formula Cramer 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- then be normalized direct (i.e. bring them to a state that the sum of the squares of the coefficients  $a$ , and  $b$  equal to unity), otherwise errors in parallelism of the direct comparison and coincidence may be too large .

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:14  
EDIT: 24 Aug 2011 12:06

## 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 an infinite number of different lines).

### Contents [hide]

- Finding the equation for a straight line segment
  - Two-dimensional case
    - Integer case
    - The real-valued case
  - Three-dimensional and multi-dimensional case

### Two-dimensional case

Let a segment  $PQ$ , ie known coordinates of its ends  $P_x, P_y, Q_x, Q_y$ .

Required to construct **the equation of a straight line on a plane** passing through this segment, ie find the coefficients  $A, B, C$  straight-line equation:

$$Ax + By + C = 0.$$

Note that the required triples  $(A, B, C)$ , passing through a given segment **are infinitely many** : you can multiply all three coefficients of an arbitrary non-zero number and get the same straight 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) that fits the following set of factors:

$$\begin{aligned} A &= P_y - Q_y, \\ B &= Q_x - P_x, \\ C &= -AP_x - BP_y. \end{aligned}$$

### Integer case

An important advantage of this method of constructing a straight line is that if all coordinates are integers, the coefficients will be obtained **by integer**. In some cases, this allows geometric operations do not resort to real numbers.

However, there is a small drawback: for one and the same line can be obtained

different triples coefficients. To avoid this, but did not move away from the integer coefficients, you can use this technique, often called **ratiōning**. We find the greatest common divisor of the numbers  $|A|, |B|, |C|$ , divide it all three coefficients, and then proizvedēm normalization sign if  $A < 0$  or  $A = 0, B < 0$  then multiply all three coefficients on  $-1$ . As a result, we come to the fact that for the same lines will receive the same triple factors that make it easy to check directly for equality.

## The real-valued case

When working with real numbers should always be aware of errors.

Coefficients  $A$  and  $B$  we have obtained the order of the original coordinates, the coefficient  $C$  - is the order of the square from them. It may already be sufficiently large numbers, and, for example, the intersection of lines they would be even more, which can lead to large errors even when rounding the original coordinates order  $10^3$ .

Therefore, when working with real numbers, it is desirable to produce a so-called **normalization** straight line: namely, make such coefficients to  $A^2 + B^2 = 1$ . For this purpose it is necessary to calculate the number of  $Z$ :

$$Z = \sqrt{A^2 + B^2},$$

and divide all three coefficients  $A, B, C$  to it.

Thus, the order of the coefficients  $A$  and  $B$  will not depend on the order of input coordinates, and the ratio  $C$  will be of the same order as the input coordinates. In practice, this leads to a significant improvement in the accuracy of calculations.

Finally, we mention the **comparison** of direct - because after this normalization to the same straight line can be obtained only two triples factors: up to multiplication on  $-1$ . Accordingly, if we proizvedēm additional normalization taking into account the sign (if  $A < -\varepsilon$  or  $|A| < \varepsilon, B < -\varepsilon$  then multiplied by  $-1$ ), 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 the system of two equations, but it is inconvenient way).

Consequently, in three-dimensional and multidimensional cases, we must use a **parametric way to set straight**, ie in the form of a point  $p$  and a vector  $v$ :

$$p + vt, \quad t \in \mathbb{R}.$$

ie straight - it all points that can be obtained from the point of  $p$  addition of the vector  $v$  with an arbitrary coefficient.

**Construction of** the line in parametric form the coordinates of segment endpoints - is trivial, we just take one end of the length of the point  $p$  and the vector from the first to the second end - for the vector  $v$ .

# MAXimal

[home](#)

[something](#)

[bookz](#)

[forum](#)

[about](#)

Added: 11 Jun 2008 10:19  
EDIT: 7 Jul 2009 17:57

## The point of intersection of the lines

Suppose we are given two lines defined by their coefficients  $A_1, B_1, C_1$  and  $A_2, B_2, C_2$ . Required to find their point of intersection, or to find out what the lines are parallel.

### Contents [\[hide\]](#)

- The point of intersection of the lines
  - Solution
  - Implementation

## Solution

If two lines are not parallel, then they intersect. To find the point of intersection, it is enough to make two 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 **point of intersection** :

$$x = -\frac{\begin{vmatrix} C_1 & B_1 \\ C_2 & B_2 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{C_1B_2 - C_2B_1}{A_1B_2 - A_2B_1},$$

$$y = -\frac{\begin{vmatrix} A_2 & B_2 \\ A_1 & C_1 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{A_1C_2 - A_2C_1}{A_1B_2 - A_2B_1}.$$

If the denominator is zero, ie,

$$\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} = A_1B_2 - A_2B_1 = 0$$

the system has no solutions (direct **parallel** and do not coincide) or has infinitely many (straight **match**). If you need to distinguish between these two cases, it is necessary to verify that the coefficients  $C$  are directly proportional

with the same coefficient of proportionality, the coefficients  $A$  and  $B$ , which is enough to count two of the determinant 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;
    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;
}

```

# MAXimal

home  
something  
bookz  
forum  
about

Added: 11 Jun 2008 10:22  
EDIT: 20 Jun 2011 1:15

## The intersection of two segments

Given two segments  $AB$ , and  $CD$ (they may degenerate to a point). Required to find their intersection: it may be empty (if the segments do not overlap) may be a single point, and may be an integer interval (if the segments are superimposed on each other).

### Contents [hide]

- The intersection of two segments
  - Algorithm
  - Implementation

## Algorithm

Work with both segments will direct us build on two segments of the equation of direct, checking whether the parallel lines. If the lines are not parallel, then it's simple: find their intersection point and check that it belongs to both segments (it is enough to verify that the point belongs to each segment in the projection on the axis  $X$ and the axis  $Y$ separately). As a result, in this case the answer is either "empty", or only one point found.

A more complicated case - if the lines were parallel (This includes the case where one or both of the segment degenerated into a point). In this case, it is necessary to check that both segments are collinear (or, in the case where both of them are degenerate to the point - that this point is the same). If it is not, then the answer - "empty". If so, then the answer - it is the 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 end of the minimum.

At the beginning of the algorithm will write the so-called "check on bounding box" - firstly, it is necessary for the case when the two segments are collinear, and secondly, it is a lightweight verification allows the algorithm to work faster on average random tests.

## Implementation

We give here a complete implementation, including all auxiliary functions for working with points and lines.

The main feature here is `intersect`that intersects two segments referred to it, and if they intersect at least at one point, then returns `true`, and the arguments `left`and `right`returns the beginning and end of the interval response (in particular, when the answer - it is the only point that is returned beginning and end will be 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);
    }
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:23  
EDIT: 11 Jun 2008 10:24

## Finding the area of a simple polygon

Suppose we are given a simple polygon (ie without self-intersections, but not necessarily convex) sets the coordinates of its vertices in the traversal or counter-clockwise. Required to find its area.

### Contents [hide]

- Finding the area of a simple polygon
  - Method 1
  - Method 2

### Method 1

This is easy to do if iterate over all the edges and fold the area of a trapezoid, the limitations of each rib. The area must be taken with the sign with which she will (it is due to sign all the "extra" area will be reduced). le 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, brute over all edges, adding to the response-oriented area of the triangle formed by the edge and the point O (see. [centric area of the triangle](#) ). Again, thanks to the sign, the whole area will reduce the excess, and will only answer.

This method is good because it is easier to generalize to more complex cases (for example, when some parties - not straight and circular arcs).

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:25  
EDIT: 27 Sep 2010 22:57

## Pick theorem. Finding the square lattice polygon

Polygon without self-intersection is called lattice if all its vertices are the points with integer coordinates (Cartesian).

### Contents [hide]

- Pick theorem. Finding the square lattice polygon
  - Theorem Pick
    - Formula
    - Proof
  - Generalization to higher dimensions

## Theorem Pick

### Formula

Suppose we are given some Grate polygon with nonzero area.

We denote its area through  $S$ ; the number of points with integer coordinates lying strictly inside the polygon - through  $I$ ; the number of points with integer coordinates lying on the sides of the polygon - through  $B$ .

Then the following relation, called **Pick's theorem** :

$$S = I + \frac{B}{2} - 1.$$

In particular, if the values of  $I$  and  $B$  for some of the polygon, the surface area can be calculated for  $O(1)$  without even knowing the coordinates of its vertices.

This relationship is discovered and proved the Austrian mathematician Georg Alexander Pick (Georg Alexander Pick) in 1899

### Proof

The proof is carried out 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 non-degenerate rectangle with sides parallel to the coordinate axes. To prove denoted by  $a$  and  $b$  lengths of the sides of the rectangle.

Then we find that  $S = ab$ ,  $I = (a - 1)(b - 1)$ ,  $B = 2(a + b)$ . Direct substitution we see that Pick's theorem is true.

- Right triangle with legs parallel to the axes. For proof, note that any such triangle can be obtained by cutting off some of its diagonal of the rectangle. Denoting the  $c$  number of integer points lying on the diagonal, we can show that Pick's theorem is satisfied for such a triangle, regardless of the value  $c$ .
- Arbitrary triangle. Note that any such triangle can be transformed into a rectangle sticking to the sides of his right triangle with legs parallel to the axes (in this case you will need no more than three such triangles). From here you can get the correct formula for Pick any triangle.
- Arbitrary polygon. To prove his triangulate, ie We divide into triangles with vertices at integer points. For a triangle formula Peak we have already proved. Then, you can prove that, when added 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 this Reeve (Reeve), proposing in 1957 to consider a tetrahedron (now called **Riva tetrahedron**) with the following peaks:

$$\begin{aligned} A &= (0, 0, 0), \\ B &= (1, 0, 0), \\ C &= (0, 1, 0), \\ D &= (1, 1, k), \end{aligned}$$

where  $k$  - any natural number. Then the tetrahedron  $ABCD$  at all  $k$  does not contain within a single point with integer coordinates, and on its border - are only 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, Pick's theorem does not allow generalizations, even on the three-dimensional case.

However, a similar generalization to higher dimension of space is still there - 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 figure.

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 11 Jul 2008 17:19  
EDIT: 11 Jul 2008 19:32

## The problem of covering of points

Given N segments on the line. Required to cover their lowest number of points, ie find the smallest set of points such that each segment belongs to at least one point.

Also consider varying degrees of sophistication version of this problem - when Adding a "forbidden" set of segments, ie no point of the answer should not belong to any of the forbidden interval.

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 the fewest points.

The following will describe a greedy algorithm that solves both problems for  $O(N \log N)$ .

### The first task

Note first that can be considered only those solutions in which each of the points located on the right end of a segment. Indeed, it is easy to understand that any decision if it does not satisfy this property, you can bring to it, shifting it to the right point as much as possible.

Let us now try to build a solution that satisfies the specified property. Take point-right ends of the segments, sort them, and we will move them from left to right. If the current point is already covered by the right end of the interval, then we miss her. Now suppose that the current point is the right end of this segment, which has not been covered before. Then we need to add in response to the current location, and mark all the segments, which belongs to this point, as covered. Indeed, if we missed the current point and would not have to add her back, then, as it is the right end of this segment, we are no longer able to cover the current segment.

However, when a naive implementation of this method will work for the  $O(N^2)$ . We describe **an efficient implementation** of this method.

Take all the points-ends of the segments (both left and right), and sort them. In this case, for each point keep with it the number of the segment, as well as the way in which it is the end of his (left or right). In addition, sort the points so that if there are a few points from one coordinate, it will first go left ends, and only then - right. Zavedëm stack, which will store the numbers of segments under consideration at the moment; initially the stack is empty. We move through the

### Contents [hide]

- The problem of covering of points
  - The first task
  - The second task

points in sorted order. If the current point - left end, then simply add the number of its segments on the stack. If it is the right end, then check to see whether or not covered by this section (for that you can just make an array of Boolean variables). If it has already been covered, then do not do anything and go to the next point (looking ahead, we argue that in this case the stack segment is no longer current). If he has 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 just numbers stored uncovered more pieces, we will get out of the stack on one segment and notes 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 coordinates with equal first go 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 to  $O(N \log N)$ .

## The second task

There are already appearing prohibited segments, therefore, firstly, the decision may not exist at all, and secondly, we can not argue that the answer can be only from 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 forbidden), sort them by saving with every point of its type and the length, the end of which it is. Again, sort the segments so as to coordinate with equal left ends were right before, and if all types are equal, the left ends of banned should go to the left end of the target, and the right ends of the forbidden - after the target (that prohibited the segments considered as long as possible with equal coordinates). Zavedäm counter forbidden lines, which will be equal to the number of forbidden lines, covering the current point. Zavedäm queue (queue), which will store the number of the current target segments. Let's take the point in the sorted order. If the current point - the left end of the target segment, then just add the number of its segment in the queue. If the current point - the right end of the target segment, if the counter is prohibited segments is zero, then we proceed as in the previous problem - put an end to the current location, and push all the segments of the line, noting that they are covered. If the counter is prohibited segments greater than zero, at the current point, we can not shoot, but because we have to find the last point that is free from prohibited segments; for this we must maintain an appropriate pointer `last_free`, which will be updated when entering forbidden segments. Then we shoot `last_free-EPS` (because it can not be right in the shoot - this point belongs to the forbidden interval), and push the segments from the queue until the point `last_free-EPS` belongs to them. Namely, if the current point - the left end of the forbidden interval, then we increase the counter, and if before the counter is zero, then assign `last_free` current position. If the current point - the right end of the forbidden interval, simply reduce the counter.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 25 Apr 2011 22:25  
EDIT: 24 Nov 2011 0:14

## Centroids of polygons and polyhedra

The **center of gravity** (or **center of mass**) of a body is the point with the property that if a body hanging over this point, it will maintain its position.

The following are the two-dimensional and three-dimensional problems related to the search of different centers of mass - mainly in terms of computational geometry.

As discussed below, there are two solutions of the basic **facts**. The 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 union will lie on the segment joining the two centers, with he will share it in the same regard as the mass of the second figure refers to the weight first.

### Contents [hide]

- Centroids of polygons and polyhedra
  - The two-dimensional case: polygons
    - Center of mass of a system of points
    - The center of mass frame
    - Center of mass of solid figures
      - The case of the triangle
      - The case of the triangle: the proof
      - The case of the polygon
      - Case polygon: Alternative method
  - Three-dimensional case: polyhedra
    - Center of mass of a system of points
    - The center of mass frame of the polyhedron
    - Center of mass of the surface of the polyhedron
    - The center of mass of solid polyhedron
      - The case of the tetrahedron
      - An arbitrary polyhedron

## The two-dimensional case: polygons

In fact, speaking of the center of mass of the two-dimensional figures, it can be understood one of the three following **tasks**:

- Center of mass of a system of points - ie the whole mass is concentrated only at the vertices of the polygon.
- The center of mass frame - ie mass polygon focused on its perimeter.
- Center of mass of solid figures - ie the mass of the polygon is distributed over the entire area.

Each of these tasks is an independent decision, and will be discussed separately below.

### Center of mass of a system of points

This is the easiest of the three tasks, and its solution - known physical formula center of mass of a system of particles:

$$\vec{r}_c = \frac{\sum_i \vec{r}_i m_i}{\sum_i m_i},$$

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 **arithmetic mean** coordinate points. For a triangle , this point is called the **centroid** coincides with the intersection point of the medians:

$$\vec{r}_c = \frac{\vec{r}_1 + \vec{r}_2 + \vec{r}_3}{3}.$$

For the proof of these formulas is enough to recall that the equilibrium is reached at a point  $r_c$  in which the sum of the moments of all forces is zero. In this case, it becomes the condition that the sum of the radius vectors of all points around  $r_c$ , multiplying the masses of the corresponding points, 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.

## The 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 middle of the segment), with a mass equal to the length of this segment.

Now we have the problem of the system of material points, and applying it to the solution of the preceding paragraph, we find:

$$\vec{r}_c = \frac{\sum_i \vec{r}_i' l_i}{P},$$

wherein  $\vec{r}_i'$ - the mid-point  $i$  of the polygon -s,  $l_i$ - the length of  $i$ th hand  $P$ - the perimeter, i.e. the sum of the lengths of the sides.

For a triangle we can show the following result: this point is **the point of intersection of the bisectors** of the triangle formed by the midpoints of the sides of the original triangle. (To see this, we must use the formula found above, and then notice that the bisector of a triangle divide the parties in the same proportions as the centers of mass of the parties).

## Center of mass of solid figures

We believe that mass is distributed uniformly on the figure, ie, density at each point of the figure is the same number.

### The case of the triangle

It is argued that for a triangle answer is still the same **centroid**, ie point formed by the average of the coordinates of the vertices:

$$\vec{r}_c = \frac{\vec{r}_1 + \vec{r}_2 + \vec{r}_3}{3}.$$

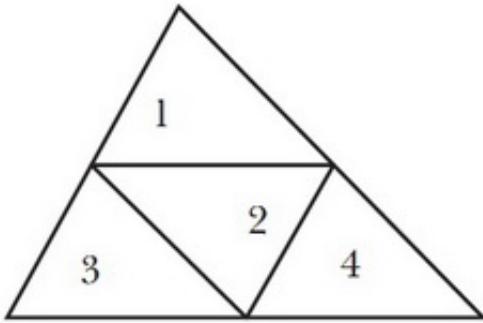
### The case of the triangle: the proof

We give here an elementary proof that does not use the theory of integrals.

Like the first, purely geometric proof led Archimedes, but it was very difficult, with a large number of geometric constructions. Proof given here is taken from an article by Apostol, Mnatsakanian "Finding Centroids the Easy Way".

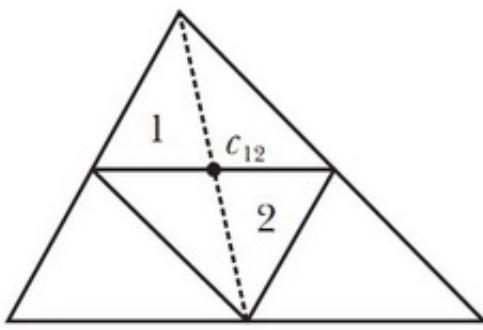
Proof boils down to in order to show that the center of mass of the triangle lies on one of the medians; repeating this process twice more, we thus show that the center of mass lies at the intersection of the medians, which is the centroid.

We divide this triangle  $T$  into four, joining midpoints of the sides, as shown below:

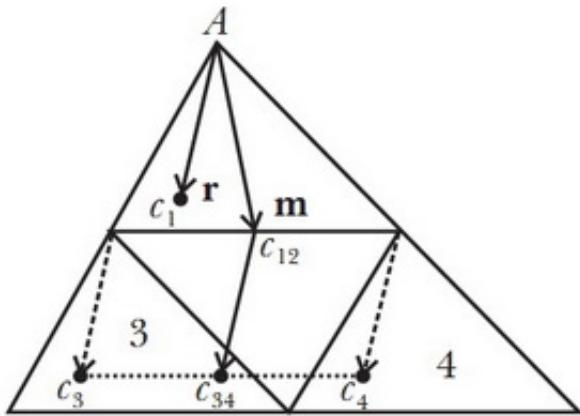


Four of the resulting triangles are similar triangle  $T$  with a coefficient  $1/2$ .

Triangles №1 and №2 together form a parallelogram whose center of mass  $c_{12}$  lies at the intersection of its diagonals (since this figure is symmetric with respect to both diagonals, and thus, its center of mass shall be on each of the two diagonals). Point  $c_{12}$  is in the middle of the common side triangles №1 and №2, and lies on the median of the triangle  $T$ :



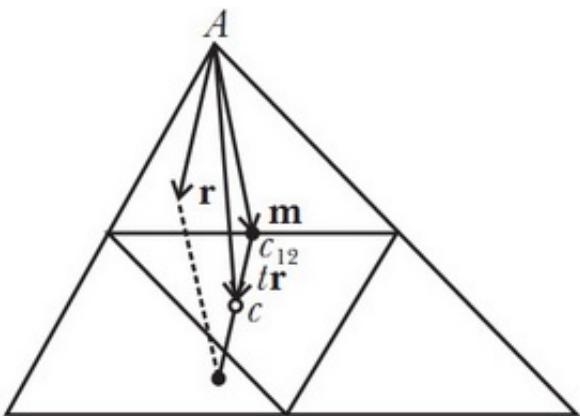
Now suppose that the vector  $\vec{r}$  - vector drawn from the top of  $A$  the center of mass  $c_1$  of the triangle №1, and let the vector  $\vec{m}$  - vector drawn from the  $A$  point to  $c_{12}$  (which, recall, is the midpoint of the side on which it is):



Our goal - to show that the vectors  $\vec{r}$  and  $\vec{m}$  are collinear.

Denote by  $c_3$  and  $c_4$  the points are the centers of mass of triangles №3 and №4. Then, obviously, the center of mass of the two triangles together will be a point  $c_{34}$ , which is the midpoint  $c_3c_4$ . Furthermore, the vector from point  $c_{12}$  to point  $c_{34}$  coincides with the vector  $\vec{r}$ .

Seeking the center of mass  $c$  of the triangle  $T$  lies in the middle of the segment joining the point  $c_{12}$  and  $c_{34}$  (since we split the triangle  $T$  into two equal areas: №1-№2 and №3-№4):



Thus, a vector from the vertex  $A$  to the centroid  $c$  is  $\vec{m} + \vec{r}/2$ . On the other hand, because the №1 triangle similar to the triangle  $T$  by a factor  $1/2$ , this is the same vector  $2\vec{r}$ . Hence we obtain the equation:

$$\vec{m} + \vec{r}/2 = 2\vec{r},$$

where we find:

$$\vec{r} = \frac{2}{3}\vec{m}.$$

Thus, we have proved that the vectors  $\vec{r}$  and  $\vec{m}$  are collinear, which means that the required centroid  $C$  lies on the median coming from the top  $A$ .

Moreover, in passing we have proved that the centroid divides each median in relation 2 : 1 counting from the top.

### The case of the polygon

Let us now turn to the general case - ie to the case **mnoougougnika**. For him, these arguments are no longer applicable, so we reduce the problem to a triangle: namely, we will divide the polygon into triangles (ie triangulate it), 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 triangulation given polygon  $S_i$ - the area  $i$ of the triangle th triangulation  $S$ - the area of the entire polygon.

Triangulation of a convex polygon - a trivial task: to do so, for example, can take the triangles  $(r_1, r_{i-1}, r_i)$  where  $i = 3 \dots n$ .

### Case polygon: Alternative method

On the other hand, the application of the reduced formula is not very convenient for **non-convex polygons**, since they produce a triangulation - in itself a challenge. But for these polygons can think of a simpler approach. Namely, an analogy with the way you can search for an arbitrary polygon area: Select an arbitrary point  $z$ , and then summed iconic square triangles formed by this point and polygon points:  $S = |\sum_{i=1}^n S_{z,p_i,p_{i+1}}|$ . A similar technique can be applied to find the center of mass: only now we will summarize the centers of mass of the triangles  $(z, p_i, p_{i+1})$ , combined with coefficients proportional to their area, ie,

The final formula for the center of mass is:

$$\vec{r}_c = \frac{\sum_i \vec{r}_{z,p_i,p_{i+1}}^o S_{z,p_i,p_{i+1}}}{S},$$

where  $z$ - arbitrary point  $p_i$ - the point of the polygon  $\vec{r}_{z,p_i,p_{i+1}}^o$ - Centroid  $(z, p_i, p_{i+1})$ ,  $S_{z,p_i,p_{i+1}}$ - a symbolic area of this triangle  $S$ - sign area of all the

polygon (ie  $S = \sum_{i=1}^n S_{z,p_i,p_{i+1}}$ ).

## Three-dimensional case: polyhedra

Similarly, two-dimensional case, in 3D, you can talk directly about four possible formulations of the problem:

- Center of mass of a system of points - vertices of the polyhedron.
- The center of mass frame - edges of the polyhedron.
- Center of mass of the surface - that is, mass is distributed over the surface area of a polyhedron.
- The center of mass of solid polyhedron - ie mass is distributed around the polyhedron.

### Center of mass of a system of points

As in the two-dimensional case, we can use physical formula and get the same result:

$$\vec{r}_c = \frac{\sum_i \vec{r}_i m_i}{\sum_i m_i},$$

that in case of equal mass turns into the arithmetic average of coordinates of all points.

### The center of mass frame of the polyhedron

Similarly, two-dimensional case, we simply replace each edge of the polyhedron material point situated in the middle of the edge, and with a mass equal to the length of this edge. After receiving the task of a material point, we can easily find its 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 - a two-dimensional figure, the center of mass that we know how to look for. Find the center of mass and replacing every facet of its center of mass, we obtain a problem with the material points which it is easy to solve.

### The center of mass of solid polyhedron

#### The case of the tetrahedron

As in the two-dimensional case, we solve first a simple task - for the tetrahedron.

It is argued that the center of mass of the tetrahedron coincides with the point of intersection of the medians (median tetrahedron is called a segment drawn from the top of its center of mass in the opposite face, so the median tetrahedron passes through the top and through the intersection of the medians of the triangular faces).

Why is this so? There are valid arguments similar two-dimensional case: if we rassechém tetrahedron two tetrahedron using a plane passing through the vertex of the tetrahedron and some median opposite face, both the resulting tetrahedron will have the same volume (as triangular face will break the median into two triangles of equal area, and the height of the two tetrahedra does not change). Repeating this several times, we find that the center of mass lies at the intersection of the medians of the tetrahedron.

This point - the point of intersection of the medians of a tetrahedron - called its **centroid**. It can be shown that it actually has coordinates equal to the arithmetic mean of the coordinates of the vertices of the tetrahedron:

$$\vec{r}_c = \frac{\vec{r}_1 + \vec{r}_2 + \vec{r}_3 + \vec{r}_4}{4}.$$

(This can be deduced from the fact that the median centroid divides against 1 : 3 )

Thus, between the cases of the tetrahedron and the triangle is no fundamental difference: the point is equal to the arithmetic mean of the vertex is the center of mass in two formulations of the problem: when the masses and is only at the vertices, and when the masses are distributed throughout the area / volume. In fact, this result can be generalized to arbitrary dimension: the center of mass of an arbitrary **simplex** (simplex) is the arithmetic mean of coordinates of its vertices.

## An arbitrary polyhedron

Let us now turn to the general case - the case of arbitrary polyhedron.

Again, as in the two-dimensional case, we produce the reduction of this problem to the already decided: divide the polyhedron into tetrahedra (ie make it tetraedrizatsiyu), we find the center of mass of each of them, and obtain the final answer to the problem in the form of a weighted sum of centers found wt.

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 11 Jun 2008 10:25  
 EDIT: 20 Aug 2008 17:44

## The intersection of a circle and a straight line

Given circle (the coordinates of its center and radius) and direct (his equation). Required to find the point of intersection (one, two or none).

### Contents [hide]

- The intersection of a circle and a straight line
  - Solution
  - Implementation

## Solution

Instead of a formal solution of the system of two equations are approaching the problem **from the geometric side** (and, due to this we get a more accurate solution in terms of numerical stability).

Assume without loss of generality that the center of the circle is at the origin (if it is not, then transfer the back of his Wikipedia suitable constant C in the equation line). Ie have a circle centered at (0,0) with radius r and direct to the equation  $Ax + By + C = 0$ .

First, find **the closest to the center point of the line** - the point with some coordinates  $(x_0, y_0)$ . Firstly, this point should be at such a distance from the origin:

$$\frac{|C|}{\sqrt{A^2 + B^2}}$$

Secondly, since the vector  $(A, B)$  perpendicular to the line, the coordinates of the point must 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:

$$x_0 = \frac{AC}{A^2 + B^2}$$

$$y_0 = \frac{BC}{A^2 + B^2}$$

(There are not obvious signs only 'negative', but these formulas can be easily verified by substituting in the equation of the line - is supposed to be zero)

Knowing the closest to the center of the circle point, we can now determine how many points will contain the answer, and even to give an answer, if these points are 0 or 1.

Indeed, if the distance from  $(x_0, y_0)$  to the origin (and we have already expressed his formula - See above.) larger than the radius, then **the answer - zero points**. If this distance is equal to the radius, then the **response will be one point** -  $(x_0, y_0)$ . But in the remaining case there will be two points and their coordinates to find us.

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 straight line must lie on the same distance  $d$  from the point  $(x_0, y_0)$ , where the 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 a straight line, but because the desired point  $(ax, ay)$  and  $(bx, by)$  may be prepared by adding to a point  $(x_0, y_0)$  of vector  $(-B, A)$ , the normalized the length  $d$  (we get a desired point), and subtracting the same vector (get a second desired point).

The final decision is:

$$\begin{aligned} d^2 \\ \text{mult} &= \sqrt{\frac{d^2}{A^2 + B^2}} \\ ax &= x_0 + B \text{ mult} \\ ay &= y_0 - A \text{ mult} \\ bx &= x_0 - B \text{ mult} \\ by &= y_0 + A \text{ mult} \end{aligned}$$

If we solved this problem in a purely algebraic, then most likely would get a decision in another form, which gives large errors. Therefore, the "geometric" method described here, in addition to visibility, more and more accurate.

## 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 equation of the line.

```
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';
}
```

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 11 Jun 2008 10:26  
 EDIT: 11 Jun 2008 10:27

## The intersection of two circles

Given two circles, each determined the coordinates of its center and radius. You want to find all of their points of intersection (either one or two, or no point to a circle are the same).]

### Contents [hide]

- The intersection of two circles
  - Solution

### Solution

We reduce our problem to the problem of the [intersection of the circle and a straight line](#).

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 added back to the coordinates of the center). Then we have a system of two equations:

$$\begin{aligned}x^2 + y^2 &= r_1^2 \\(x - x_2)^2 + (y - y_2)^2 &= r_2^2\end{aligned}$$

Subtract the second equation of the first to get rid of the squares of the variables:

$$\begin{aligned}x^2 + y^2 &= r_1^2 \\x(-2x_2) + y(-2y_2) + (x_2^2 + y_2^2 + r_1^2 - r_2^2) &= 0\end{aligned}$$

Thus, we have reduced the problem of the intersection of two circles to the problem of the intersection of the first circle and a straight line following:

$$\begin{aligned}Ax + By + C &= 0 \\A &= -2x_2, \\B &= -2y_2, \\C &= x_2^2 + y_2^2 + r_1^2 - r_2^2.\end{aligned}$$

A solution of this problem is described in [the relevant article](#).

Only **the degenerate case**, which must be considered separately - when the centers of the circles coincide. Indeed, in this case, instead of the line equation we obtain an equation of the form  $0 = C$ , where  $C$  - a number, and this case will be handled correctly. Therefore, this case must be considered separately: if the radii of the circles are the same, then the answer - infinity, otherwise - no points of intersection.

# MAXimal

home  
something  
bookz  
forum  
about

Added: 11 Jun 2008 10:30  
EDIT: 22 Mar 2012 12:17

## Construction of the convex hull of a bypass Graham

Given N points of the plane. Build their convex hull, ie the smallest convex polygon containing all these points.

We consider the method of **Graham** (Graham) (proposed in 1972) with improvements Andrew (Andrew) (1979). Use it to construct the convex hull of the time  $O(N \log N)$  using only comparison operations, addition and multiplication. The algorithm is asymptotically optimal (shown that there is no algorithm with a better asymptotic), although some tasks it is unacceptable (in the case of parallel processing or online-processing).

### Description

Algorithm. Let us find the leftmost and rightmost points A and B (if several such points, then take the lowest among the left and right among the very top). It is clear that A, B and certainly fall within the convex hull. Next, draw through these lines AB, dividing the set of all points on the upper and lower subsets S1 and S2 (points lying on a straight line can be attributed to any set - they still will not be included in the shell). Points A and B othetõm to both sets. Now we construct for S1 upper shell, and for S2 - lower shell, and combine them to get an answer. To get in, say, the top shell, you need to sort all points on the abscissa, and then go through all the points, considering at each step except for the point of the previous two points included in the envelope. If the current triple of points does not form a right turn (which is easily verified by [the oriented area](#) ), the nearest neighbor to remove from the shell. Eventually, there will be only a point included in the convex hull.

Thus, the algorithm is to sort all the points on the abscissa and two (in the worst case) all rounds points, i.e. 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 x * (by-cy) + bx * (cy-ay) + cx * (ay-by) < 0;
}

bool ccw (pt a, pt b, pt c) {
    return x * (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]);
}
```

### Contents [hide]

- Construction of the convex hull of a bypass Graham
  - Description
  - Implementation

## Finding the area of combining triangles. Vertical decomposition method

Dana N triangles. Need to find the area of their union.

### Solution

Here we consider the method of **vertical decomposition**, which challenges the geometry is often very important.

So, we have N triangles that can somehow interfere with each other. Get rid of these intersections using vertical decomposition: find all the 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 along this array. At the i-th step we consider the elements of B [i] and B [i + 1]. We have a vertical strip between the lines  $X = B[i]$  and  $X = B[i + 1]$ , and, according to the construction of the array B, inside this band segments can not intersect with each other. Therefore, within this triangle strips are cut to the trapezoidal rule, with the part of these trapezoids inside the band do not overlap at all. We move to the sides of the trapezoidal rule from the bottom up, and dump area of a trapezoid, making sure that each piece was uchitan exactly once. In fact, this process is very similar to the processing of nested parentheses. Adding the area of a trapezoid within each band, and adding the results for all bands, we find the answer - the area of combining triangles.

Consider again the process of adding the area of a trapezoid, already in terms of implementation. We iterate through all aspects of the triangles, and if any party (not vertical, vertical sides, we do not need, and on the contrary, will greatly disturb) falls into this vertical bar (fully or partially), then we put it aside 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 bar, 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 element in the vector will contain a lower side of the lower trapezoid. Now we just go to the resultant vector. Let i - the current item; this means that the i-th piece - is the bottom side of a trapezoid, a block (which may contain several trapezoids), an area we want to immediately add to the answer. Therefore, we set a counter triangle is equal to 1, and climb up on the segments, and increase the counter if we find the side of a triangle for the first time, and reduce the counter when we find a triangle for the second time. If at some point in j counter becomes equal to zero, we find the upper limit of the block - at the same time we stop, add the area of the trapezoid bounded by segments i and j, and i to  $j + 1$ , and repeat the whole process again.

So, thanks to the vertical decomposition method, we solved this 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 (p.x >= s1.x1-EPS && p.x <= s1.x2+EPS && p.y >= s2.x1-EPS && p.y <= 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 (a-b) < EPS;
}

```

### Contents [hide]

- Finding the area of combining triangles. Vertical decomposition method
  - Solution
  - Implementation

```

}

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 csz = 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];
                add += contribute;
                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;
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:32  
EDIT: 11 Jun 2008 10:33

## Checking points on the convex polygon affiliation

Given a convex polygon with N vertices, the coordinates of all vertices are integral (though it does not change the essence of the decision); vertices are given by way of counter-clockwise (otherwise, you just need to sort them). Receives requests - point and is required for each point to determine is it inside the polygon or not (polygon boundaries are included). Each request will respond to the on-line for O ( $\log N$ ). Pretreatment of the polygon will be performed for the O (N).

### Algorithm

Will solve the **binary search on the corner**.

One solution is as follows. Choose a point with the smallest coordinate X (if there are several, you choose the lowest, ie with the smallest Y). Concerning this point, denoted by Zero, all other vertices of the polygon lie in the right half-plane. Further, we note that all the vertices of the polygon already ordered the angle relative to the point Zero (this follows from the fact that the polygon is convex, and has already ordered counterclockwise), with all the angles are in the range  $(-\pi / 2; \pi / 2]$ .

Let comes another request - a point P. Consider its polar angle relative to the point Zero. Binary search will find two such neighboring vertices of the polygon L and R that the polar angle P lies between the polar angles L and R. Thus, we found that sector of the polygon, which contains the point P, and we only need to check if the point P lies in the triangle (Zero, L, R). This can be done, for example, by [the oriented area of the triangle and the predicate "Clockwise"](#), just look, clockwise or counterclockwise is triple of vertices (R, L, P).

Thus, we have for the O ( $\log N$ ) find the sector of the polygon, and then in O (1) check the points belonging to the triangle, and therefore, the required asymptotic behavior is reached. Pretreatment of the polygon is only to predposchitat polar angles at all points, though, these calculations can also be transferred to the binary search step.

### Comments on the implementation of

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 do not use fractional arithmetic.

Note that the polar angle of the point (X, Y) relative to the origin is uniquely determined by the fraction Y / X, on condition that the point is in the right half. Moreover, if the polar angle one point smaller than the other, then the fraction Y1 / X1 is less Y2 / X2, and vice versa.

Thus, for comparison of the polar angles of two points, it suffices to compare fractions Y1 / X1 and Y2 / X2, it is already possible to perform integer arithmetic.

### Implementation

This implementation assumes that there are no duplicate this polygon vertices of the polygon and the area is non-zero.

```

struct PT {
    int x, y;
};

struct ang {
    int A, b;
};

bool Operator <(const & P ang, ang const & q) {
    if (0 == Pb && qb == 0)
        Return pa < qa;
    Return pa * * 1LL qb <Pb * * 1LL qa;
}

Long Long sq (PT & A, b & PT, PT & C) {
    Return AX * * 1LL (by-CY) + BX * * 1LL (CY -ay) + CX * * 1LL (AY-by);
}

int main () {

    int n;
    CIN >> n;
}

```

### Contents [hide]

- Checking points on the convex polygon affiliation
  - Algorithm
  - Comments on the implementation of
  - Implementation

```

vector <PT> P (n);
int = 0 zero_id;
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;
}
P = PT zero [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 = false in;
    if (QX> = zero.x)
        if (QX == == zero.x && QY zero.y)
            in = True;
        else {
            = {ang my QY-zero.y, QX-zero.x};
            if (0 == my.a)
                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)
                    = True in;
            }
        }
    puts (in? "INSIDE" : "OUTSIDE");
}
}

```

# MAXimal

home  
something  
bookz  
forum  
about

Added: 11 Jun 2008 10:33  
EDIT: 11 Jun 2008 10:34

## 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 the radius of the largest inscribed circle.

Here we describe a simple method to solve this problem using two ternary search working for  $O(N \log^2 C)$ , where C - a coefficient determined by the value of the coordinates and the required accuracy (see. below).

### Algorithm

We define the function **Radius (X, Y)**, which returns the radius of the inscribed polygon at the circle centered at the point (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)** - simply pass on all sides of the polygon, we consider for each distance to the center (and you can take away from both direct to the point, is not necessarily viewed as a segment) and returns the minimum of the distances found - obviously, he will be the greatest radius.

So, we need to maximize this function. Note that, as a convex polygon, then this function will be suitable for **the ternary search** in both arguments: the fixed  $X_0$  (of course, such that the line  $X = X_0$  intersects the polygon) function **Radius ( $X_0, Y$ )** as a function of one argument  $Y$  will first increase, then decrease (again, we consider only the  $Y$ , the point  $(X_0, Y)$  belongs to the polygon). Moreover, the function **max (on  $Y$ ) {Radius ( $X, Y$ )}** as a function of one argument  $X$  will first increase and then decrease. These properties are clear from geometrical 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 ternary search because the calculation of the function **Radius** outside the polygon will be incorrect. To search for  $X$  no difficulty, simply choose the abscissa of the left and the right-most point. To search by  $Y$  are those segments of the polygon, which gets the current  $X$ , and find the coordinates of the points of these segments at the abscissa  $X$  (vertical segments is not considered).

It remains to estimate **the asymptotic behavior**. Let the maximum value that can take the coordinates - this is  $C_1$ , and the required accuracy - about  $10^{-C_2}$ , 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 resulting asymptotic behavior is obtained:  $O(N \log^2 C)$ .

### Implementation

Constant steps determines the number of steps both ternary search.

In the implementation should be noted that for each side immediately predposchityvayutsya coefficients in the equation of direct and immediately normalized (divided into  $\sqrt{A^2 + B^2}$ ) to avoid unnecessary operations in 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 * l.a + y * l.b + l.c);
}

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;
```

### Contents [hide]

- Finding the inscribed circle of a convex polygon using the ternary search
  - Algorithm
  - Implementation

```

    for (int i=0; i<n; ++i) {
        int x1 = a [i] .x, x2 = a [(i + 1)% n] .x, y1 = a [i] .and, y2 = a [(i + 1)% n] .and;
        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 his = 0; its <steps; ++ are) {
        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)
            l = 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] .x - a [(i + 1)% n] .and;
        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 y_radius years = (W, a, 1);
    printf ("% .7lf", ans);
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:34  
EDIT: 28 Feb 2012 11:27

## Finding the inscribed circle of a convex polygon method "compression sides" ("shrinking sides") for $O(n \log n)$

Given a convex polygon with  $n$  vertices. Required to find the inscribed circle of maximum radius, ie find its radius and center coordinates. (If for a given radius of several options centers, it is sufficient to find any of them.)

Unlike described [here](#) of the double ternary search asymptotic behavior of the algorithm -  $O(n \log n)$ - does not depend on restrictions on the location and the required accuracy, and therefore, this algorithm is suitable at much larger  $n$  and more restrictions on the value of the coordinates.

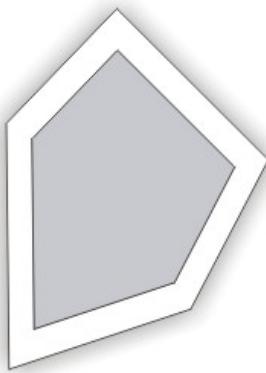
Thank you [Ivan Krasilnikov \(mf\)](#) for a description of this beautiful algorithm.

### Contents [hide]

- Finding the inscribed circle of a convex polygon method "compression sides" ("shrinking sides") for  $O(n \log n)$ 
  - Algorithm
  - Implementation

## Algorithm

So, given a convex polygon. Let's start at the same time and at the same speed **shift** all its sides parallel to each other inside the polygon:



Suppose, for convenience, this movement occurs at a rate of 1 unit per second coordinate (ie, time in a sense, equal to the distance: a unit of time after each point will overcome a distance equal to one).

In the course of the movement of the polygon will gradually fade (refer to point). Sooner or later, the whole polygon shrinks to a point or a segment, and this time,  $t$  will be the **answer to the problem** - the desired radius (and the center of the desired circle will lie in this interval). In fact, if we had to compress the thickness of the polygon  $t$  in all directions, and he appealed to the point / interval, it means that there is a point removed from all sides of the polygon at a distance  $t$ , and for longer distances - such a point does not already exist.

So, we need to learn how to effectively simulate the compression process. To this end, each side will learn **to tell time**, through which it shrinks to a point.

For this we consider carefully the process of moving parties. Note that the vertices of the polygon is always moving along the bisectors of angles (this follows from the equality of the corresponding triangles). But then the question of time through which the party shrinks, is reduced to the question of determining the height  $H$  of the triangle, in which the known length of the side  $L$  and two adjacent angles to it  $\alpha$  and  $\beta$ . Using, for example, the law of sines, we obtain:

$$H = L \cdot \frac{\sin \alpha \cdot \sin \beta}{\sin(\alpha + \beta)}.$$

Now we know how for  $O(1)$  determine the time after which the party shrinks to a point.

Zanesëm these times for each side in some **data structure to extract a minimum**, for example, a red-black tree (set in the language C ++).

Now if we izvlechëm side with **the least time**  $H$ , this side of the first shrinks to a point - at the time  $H$ . If a polygon has not shrunk to the point / segment, then this aspect is necessary to **remove** from the polygon, and continue the algorithm for the remaining parties. When you delete a part, we must **connect** with each other its left and right neighbor, **extending** them to the point of their intersection. This will need to find this point of intersection, count the length of two sides and their

times of extinction.

When implemented for each side will have to keep the number of its left and right neighbor (and thus how to build a doubly-linked list of sides of the polygon). This allows for the removal and binding of the two sides of her neighbors for  $O(1)$ .

If the remote side is that its side neighbors are parallel , then it means that the polygon then compression degenerates into a point / interval, so we can immediately stop the algorithm and return as a response time of disappearance of the current hand (so that problems with parallel sides does not occur).

If such a situation with parallel sides does not occur, then the algorithm will finish before the point at which a polygon will be only two sides - and then the answer to the problem will be the removal of the previous hand.

Obviously, the asymptotic behavior of the algorithm is  $O(n \log n)$ as algorithm consists of  $n$ steps, each of which is removed on one side (which made several transactions setover time  $O(n \log n)$ ).

## Implementation

We give implementing the algorithm described above. This implementation returns only the desired radius of the circle; however, the addition of O center of the circle not be easy.

This elegant algorithm that from computational geometry is only required to find 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** , i.e. 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;
    ... чтение n и 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], nxt1 = (nxt+1)%n,
            prv = prev[i], prvl = (prv+1)%n;
        if (parallel (line (p[nxt], p[nxt1]), 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[nxt1],
            p[prvl], p[prv],
            p[next[nxt]], 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 feature here - it's *get\_h()* that on her side and the left and right neighbors calculates the disappearance of this party. To do this, search for a point of intersection of the side with the neighbors, and then found above formula is made Calculate your desired time.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 16 Jan 2009 0:58  
EDIT: 26 Apr 2011 16:58

## Voronoi diagram in 2D

### Determination

Given  $n$  points  $P_i(x_i, y_i)$  in the plane. Consider a partition of the plane into  $n$  regions  $V_i$  (called Voronoi polygons or Voronoi cells, sometimes - polygons proximity cells Dirichlet partition Thyssen), where  $V_i$  - the set of all points in the plane which are closer to the point  $P_i$  than to any other points  $P_k$ :

$$V_i = \{(x, y) : \rho((x, y), P_i) = \min_{k=1 \dots N} \rho((x, y), P_k)\}$$

Needless to partition the plane is called a Voronoi diagram of a set of points  $P_k$ .

Here  $\rho(p, q)$  - given metric, usually the standard Euclidean metric:

$\rho(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$  but lower case will be considered and the so-called Manhattan metric. Hereinafter, unless otherwise specified, will be considered for the Euclidean metric

Cells are convex Voronoi polygons, some are endless. Points belonging by definition to multiple Voronoi cell, and usually attributed to several cells (in the case of the Euclidean metric set of points has measure zero; in the case of the Manhattan metric everything a little more complicated).

These polygons were first studied in depth by the Russian mathematician Voronoi (1868-1908 gg.).

### Contents [hide]

- Voronoi diagram in 2D
  - Determination
  - Properties
  - Application
  - A simple algorithm for constructing Voronoi diagrams for  $O(n^4)$
  - The case of special metrics

### Properties

- Voronoi diagram is a planar graph, so it has  $O(n)$  vertices and edges.
- We 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 the half-plane formed by the straight line, which is the point  $P_i$ . Then the intersection of all half-planes for each  $j$  Voronoi cell will  $P_i$ .
- 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 the point  $P_i$  lies on the border of the convex hull  $P_k$ .
- Consider the graph dual to the Voronoi diagram, ie in this graph vertices are the points  $P_i$ , and the edge is drawn between the points  $P_i$  and  $P_j$  if their Voronoi cell  $V_i$  and  $V_j$  have a common edge. Then, under the condition that no four points lie on a circle, dual to the Voronoi diagram is a graph of the Delaunay triangulation (which has 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 needed to build most of the Voronoi diagram, in the asymptotics is not considered.

- Finding the closest point for each.

Note the simple fact that if the point  $P_i$  is the nearest point  $P_j$ , this point  $P_j$  is "his" edge of the cell  $V_i$ . It follows that to find for each point closest to her, enough to see its ribs Voronoi cell. However, each edge belongs to exactly two cells, so it will be seen exactly twice, and due to the linearity of the number of edges we obtain the 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 find the Voronoi diagram in any infinite edge and begin to move in any fixed direction (e.g., counterclockwise) the cell containing this edge until we reach the next endless rib. Then move on through this edge to the next cell and continue to crawl. As a result, all viewed edges (except for infinite) will be sought by the parties of the convex hull.

Obviously, the running time -  $O(n)$ .

- Finding the Euclidean minimum spanning tree.

Required to find the minimum spanning tree with vertices at these points  $P$ , connecting all these points. If you use the standard methods of graph theory, then, as Count in this case has  $O(n^2)$  ribs, even optimal algorithm will have no less asymptotics.

Consider the graph dual to the Voronoi diagram, ie Delaunay triangulation. It can be shown that the presence of the Euclidean minimum spanning tree is equivalent to building a core Delaunay triangulation. Indeed, [Prim's algorithm](#) each time sought the shortest edge between two points mozhestvami; if we fix a point of the set, the closest point has an edge in the Voronoi cell, so the Delaunay triangulation will attend an edge to the nearest point, as required.

Triangulation is a planar graph, ie, has a linear number of edges, so we can apply [Kruskal's algorithm](#) and an algorithm with running time  $O(n \log n)$ .

- Finding the largest empty circle.

Required to find the circumference of the largest radius, not contained within any of the points  $P_i$  (the center of the circle must lie within the convex hull of the points  $P_i$ ). Note that because function circle of largest radius at a given point  $f(x, y)$  is a strictly monotonic within each Voronoi cell, it reaches its maximum value at one of the vertices of the Voronoi diagram or edges at the intersection point of the convex hull and the diagram (and the number of such points is more than twice the number of edges chart). Thus, we can only sort out these points, and for each to find the nearest, ie solution for  $O(n)$ .

## A simple algorithm for constructing Voronoi diagrams for $O(n^4)$

Voronoi diagrams - is well-studied object, and for them to get a lot of different algorithms for optimal asymptotic behavior of working  $O(n \log n)$ , and some of these algorithms even work an average of  $O(n)$ . However, these algorithms are very complex.

We consider here the simplest algorithm based on the above cited property that each Voronoi cell is the intersection of half-planes. We fix  $i$ . Spend between a point  $P_i$  and each point  $P_j$  line - the perpendicular, then cross the pairs all received direct - get  $O(n^2)$  points, and each check for membership to all  $n$  half-plane. As a result of the  $O(n^3)$  action we get all the vertices of the Voronoi cell  $V_i$  (they will be not more  $n$ , so we can without impairing the asymptotics sort them by polar angle), but only on the construction of Voronoi diagram required  $O(n^4)$  actions.

## The case of special metrics

Consider the following metrics:

$$\rho(p, q) = \max(|x_p - x_q|, |y_p - y_q|)$$

Consideration should start with simple cases - the case of two points  $A$  and  $B$ .

If  $A_x = B_x$  or  $A_y = B_y$ , the Voronoi diagram for them to be respectively vertical or horizontal line.

Otherwise Voronoi diagram will be in the form of "corners": cut angle 45 degrees in the rectangle formed by the points  $A$  and  $B$ , and horizontal / vertical beams of its ends, whichever is longer whether vertical or horizontal side of the rectangle.

A special case - when the rectangle is the same length and width, ie  $|A_x - B_x| = |A_y - B_y|$ . 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 at once cells. In this case, an additional condition is determined, how should we understand these areas (sometimes artificially introduced a rule that every corner treats 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 will have to be able to quickly cross.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 2 Mar 2009 17:  
EDIT: 24 Aug 2011 12

## Finding all the faces, the outer edge of a planar graph

Dan planar stacked on a plane graph  $G$  with  $n$  vertices. You want to find all of its facets. Fringe 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 edge. In 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.

### Contents [hide]

- Finding all the faces, the outer edge of a planar graph
  - Euler's theorem
  - Bypass all faces
  - Isolation of the outer edge
  - Construction of a planar graph

### Euler's theorem

We give here Euler's theorem and some consequences of it, from which it follows 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 planar graph  $G$  is connected. Denoted by  $n$  the number of vertices in the graph  $m$ - the number of edges,  $f$ - the number of faces. Then we have the **Euler's theorem** :

$$f + n - m = 2$$

To prove this formula follows easily. In the case of a tree ( $m = n - 1$ ) formula is easily verified. If the graph - not a tree, it will remove any edge that belongs to any cycle; The magnitude of the  $f + n - m$  change. We repeat this process until you come to the tree, for which the identity  $f + n - m = 2$  has been established. Thus, the theorem is proved.

**Corollary**. For an arbitrary planar graph let  $k$ - the number of connected components. Then the following:

$$f + n - m = 1 + k$$

**Corollary**. The number of edges  $m$  of a planar graph is a simple magnitude  $O(n)$ .

Proof. Suppose that the graph  $G$  is connected and  $n \geq 3$  (in the case of  $n < 3$  approval is obtained automatically). Then, on the one hand, each face is bounded by at least three edges. On the other hand, the maximum of each edge two limit faces. Consequently,  $3f \leq 2m$  where, substituting this in Euler's formula, we get:

$$f + n - m = 2 \Leftrightarrow 3f = 6 - 3n + 3m \Leftrightarrow 6 - 3n + 3m \leq 2m \Leftrightarrow m \leq 3n - 6$$

Ie  $m = O(n)$ .

If the graph is not connected, then summing the resulting estimates for its connected components, again get  $m = O(n)$  what we wanted to prove.

**Corollary**. The number of faces  $f$  of a planar graph is a simple magnitude  $O(n)$ .

This corollary follows from the previous investigation and communication  $f = 2 - n + m$ .

### Bypass all faces

Always assume that the graph if it is not connected, is laid on the plane so that no connected component does not lie within the other (for example, a square with lying strictly inside the segment - an incorrect algorithm for our test).

Naturally, it is considered that the graph correctly laid on the plane; no two vertices are not the same, and the edges do not intersect in "unauthorized" points. If the input graph admits such overlapping edges, the pre-need to get rid of them, entering into every point of intersection the additional node (it should be noted that as a result of this process, instead of  $n$  points we can get the order  $n^2$  points). For more details about this process. Below in the appropriate section.

Suppose that for each vertex all outgoing edges from it ordered the polar angle. If not, they should be streamlined, producing sorting each adjacency list (because  $m = O(n)$ , it takes  $O(n \log n)$  operations).

Now we choose an arbitrary edge  $(a, b)$  and let the next round. Coming to some vertex  $v$  on an edge, out of this summit we need the following in order of sorting edge.

For example, in the first step we are in the top  $b$ , and must find a vertex  $a$  in the vertex adjacency list  $b$ , then we denote  $c$  the next node in the adjacency list (if  $a$  was the latter, as  $c$  we take the first vertex), and will walk along the edge  $(b, c)$ .

By repeating this process many times, sooner or later we will come back to the starting edge  $(a, b)$ , and then to stop. It is easy to see that for such a traversal we dispense exactly one face. And the direction of the circuit is anti-clockwise to the outer edge, and clockwise - for internal faces. In other words, when traversing this inner edge is always at the right side of this edge.

So we learned to get one face, starting from any edge on its boundary. You're starting to learn how to choose the edges so that the resulting faces are not repeated. Note that each edge two different ways in which you can get: Each of them will receive their faces. On the other hand, it clear that such one oriented edge belongs to exactly one edge. Thus, if we will mark all edges of each detected faces in a array `used`, and do not run round the edges of the already marked, we dispense all sides (including foreign), though exactly once.

We give immediately **implementing** this workaround. We assume that in the graph  $G$  adjacency lists already ordered by the corner, and multiple edges and loops are absent.

A first embodiment of a simplified, the next node in the list of adjacency he is looking for a simple search. This implementation works for theoretically  $O(n^2)$ , although in practice many tests it works very quickly (with a hidden constant is much less than unity).

```

int n; // число вершин
vector < vector<int> > g; // граф

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;
            }
            ... вывод facet - текущей грани ...
        }
    }
}

```

Another, more optimized embodiment - uses 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 (for example, filling out her as a class, as in the example below), then the search terms in the adjacency list, you can use binary search. The result of the implementation  $O(n \log n)$ .

```

class cmp_ang {
    int center;
public:
    cmp_ang (int center) : center(center)
    {}
    bool operator() (int a, int b) const {
        ... должна возвращать true, если точка а имеет
        меньший чем в полярный угол относительно center ...
    }
};

int n; // число вершин
vector < vector<int> > g; // граф

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;
            }
            ... вывод facet - текущей грани ...
        }
    }
}

```

And possible options based on the 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 does not work correctly with **isolated** peaks - such vertices it just does not show up as individual faces, though, from a mathematical point of view, they should be separate connected components and facets.

In addition, a special face is **an exterior face**. How to distinguish it from "ordinary" 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 is found algorithm separately.

## Isolation of the outer edge

The above code displays all the faces, making no distinction between the outer edge and inner faces. In practice, on the contrary, is only require

to find or outer face or inner only. There are several techniques highlight the exterior face.

For example, it is possible to determine the area - the external face must have the largest area (it should only take into account that the inner face can have the same area as the exterior). This method will not work if the planar graph  $G$  is not connected.

Another, more reliable criterion - in the direction of traversal. As already mentioned above, all the faces except external cost in a clockwise direction. The outer edge, even if it consists of several circuits will cost algorithm counterclockwise. Determine the direction of the circuit can be simply considering a [landmark area of the polygon](#). The area can be considered directly in the course of the inner loop. However, this method has its subtlety - processing faces zero area. For example, if the graph consists of a single edge, then the algorithm will find a single face area which will be zero. Apparently, if a face area is zero, it is the outer face.

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 disjoint diagonals, its outer face will contain all the vertices. But again, you have to be careful with the case where both the external and internal surface have the same number of vertices.

Finally, there are the following method to find the outer edge: You can start from such a special rib that was found in the outer face will result. For example, you can take the top of the left-most (if there are several, it will approach any) and select it from the edge going first in the sort order. As a result of circumvention of the rib find outer face. This method can be extended to the case of a disconnected graph: it is necessary to find each connected component of the leftmost top and run circumvention of the first rib from it.

We present the implementation of a simple method based on the sign of the area (I myself bypassing for example took over  $O(n^2)$ , it does not matter here). If the graph is not connected, the code is "... the outer face is ..." is provided separately for each circuit constituting the outer face.

```
... обычный код по обнаружению граней ...
... сразу после цикла, обнаруживающего очередную грань: ...

// считаем площадь
double area = 0;
// добавляем фиктивную точку для простоты подсчёта площади
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)
    ... грань является внешней ...
}
```

## Construction of a planar graph

For the above algorithms is essential that the input graph is correctly packed planar graphs. However, in practice, often fed to the input of the program set pieces may intersect with each other in "unauthorized" points, and we need to build on these segments planar graph.

To implement the construction of a planar graph as follows. We fix any input segment. Now cross the length of this with all the other segments. Point of intersection point, as well as the ends of the interval set in the vector, and sort the standard way (ie first in one coordinate, with equal - c the other). Then loop through this vector and will add edges between adjacent points in this vector (of course, making sure we did not add the loop). After completing this process for all segments, ie for  $O(n^2 \log n)$  we construct the corresponding planar graph (which will be  $O(n^2)$  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) {
    ... стандартная процедура, пересекает два отрезка а и б и закидывает результат в res ...
    ... если отрезки перекрываются, то закидывает те концы, которые попали внутрь первого отрезка ...
}

int main() {
    // входные данные
    int m;
    vector < pair<point,point> > a (m);
    ... чтение ...
```

```
// построение графа
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();
// сортировка по углу и удаление кратных рёбер
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());
}
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Jun 2009 21:03  
EDIT: 6 Nov 2012 12:38

## Finding a pair of nearest points

### Statement of the Problem

Are given  $n$  points  $p_i$  in the plane defined by its coordinates  $(x_i, y_i)$ . Required to find among them are two points, the distance between them is minimal:

$$\min_{\substack{i,j=0\dots n-1, \\ i \neq j}} \rho(p_i, p_j).$$

Distance we take the usual Euclidean:

$$\rho(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

A trivial algorithm - through all pairs and calculating the distance for each - for works  $O(n^2)$ . The following describes an algorithm that runs in time  $O(n \log n)$ . This algorithm has been proposed by Preparata (Preparata) in 1975 and Drug Shamos also shown that the decision tree model in the algorithm is asymptotically optimal.

### Contents [hide]

- Finding a pair of nearest points
  - Statement of the Problem
  - Algorithm
  - Evaluation of the asymptotics
  - Implementation
  - Summary: Search the triangle with minimal perimeter
  - Tasks in the online judges

### Algorithm

We construct an algorithm for the general scheme of algorithms "**divide-and-rule**" : the algorithm presented in the form of a recursive function, which passed a set of points; This recursive function divides the set of two, calls itself recursively on each half, and then performs some operations to consolidate the responses. Merge operation is detected when one point optimal solutions fell into one half, and another point - to another (in this case, the recursive calls from each of the two halves separately detect the pair, of course, can not). The main difficulty, as always, lies in the effective implementation of this stage of the association. If a recursive function is passed the set of  $n$  points, then the stage of integration should work no more than  $O(n)$ , then the asymptotic behavior of the whole algorithm  $T(n)$  will be out of the equation:

$$T(n) = 2T(n/2) + O(n).$$

The solution of this equation is known to be  $T(n) = O(n \log n)$ .

So, let's move to the construction of the algorithm. In the future to come to the effective implementation of the process of consolidation, split into two set of points according to their will  $x$  are the coordinates: in fact we spend some vertical line which divides the set of points into two subsets of approximately equal size. Such a division is convenient to perform as follows: sort the standard terms as a pair of numbers, ie.:

$$p_i < p_j \iff (x_i < x_j) \vee ((x_i = x_j) \wedge (y_i < y_j)).$$

Then take the average after sorting point  $p_m$  ( $m = \lfloor n/2 \rfloor$ ), all point to it and very  $p_m$  to the first half, and all points after it - in the second half:

$$\begin{aligned} A_1 &= \{p_i \mid i = 0 \dots m\}, \\ A_2 &= \{p_i \mid i = m + 1 \dots n - 1\}. \end{aligned}$$

Now, to cause recursively from each of the sets  $A_1$  and  $A_2$  we find the answers  $h_1$  and  $h_2$  for each of the halves. Take the best of them:  $h = \min(h_1, h_2)$ .

Now we need to make **the step of combining**, ie try to find such a pair of points, the distance between them is less  $h$ , with one point lies in  $A_1$ , and the other - in  $A_2$ . It is obvious that it is sufficient to consider only those points which are spaced from the vertical line section is not a distance less than  $h$ , ie, a plurality of  $B$  considered

points at this stage is:

$$B = \{p_i \mid |x_i - x_m| < h\}.$$

For each point of the set  $B$  must try to find points that are closer to it than that  $h$ . For example, it is sufficient to consider only those points whose coordinates  $y$  which differ by no more than  $h$ . Moreover, it makes no sense to consider the points at which  $y$  the coordinate of more  $y$  coordinates of the current point. Thus, for each point  $p_i$  define the set of points considered,  $C(p_i)$  as follows:

$$C(p_i) = \{p_j \mid p_j \in B, \quad y_i - h < y_j \leq y_i\}.$$

If we sort the points in the set  $B$  to  $y$ -coordinate, the finding  $C(p_i)$  will be very easy: it is several points in a row to the point  $p_i$ .

So, in the new notation **stage association** is as follows: construct a set  $B$ , sort in it to the point  $y$ -coordinate, then for each point  $p_i \in B$  to consider all the points  $p_j \in C(p_i)$ , and each pair  $(p_i, p_j)$  to calculate the distance and compare with the current best distance.

At first glance, this is still not optimal algorithm: it seems that the size of the sets  $C(p_i)$  will be of the order  $n$  and the required asymptotic behavior does not work. However, surprisingly, it is possible to prove that the size of each of the sets  $C(p_i)$  have a value  $O(1)$ , ie, does not exceed a certain small constant regardless of the points themselves. The proof is given in the next section.

Finally, pay attention to the sort of which the above algorithm contains just two: first, sorted in pairs ( $x, y$ ) and then sorting the elements of the set  $B$  on  $y$ . In fact, both of these sorts inside the recursive function can be removed (otherwise we would not have reached the evaluation  $O(n)$  stage for the association, and the general asymptotic behavior of the algorithm would have been  $O(n \log^2 n)$ ). From the first sorting rid easy - just in advance of the launch of recursion to perform this sort: it inside the recursion elements themselves do not change, so there is no need to sort again. With the second sorting little harder to execute it without first obtaining. But, remembering **mergesort** (merge sort), which also operates on the principle of divide-and-conquer, you can simply embed this sort in our recursion. Let recursion, taking some set of points (as we recall, ordered pairs  $(x, y)$ ) returns the same set, but already sorted in the coordinate  $y$ . To do this, simply perform a mail merge (for  $O(n)$ ) two results returned by recursive calls. Thereby obtaining sorted by  $y$  set.

## Evaluation of the asymptotics

To show that the above algorithm is indeed satisfied for  $O(n \log n)$ , it remains to prove the following fact  $|C(p_i)| = O(1)$ .

So, let us consider some point  $p_i$ . Recall that the set  $C(p_i)$  - a set of points,  $y$  the coordinate of which lies in the interval  $[y_i - h; y_i]$ , and, in addition, the coordinate  $x$  and the point itself  $p_i$ , and all the points of the set  $C(p_i)$  lie in the strip width  $2h$ . In other words, we are considering the points  $p_i$  and  $C(p_i)$  lie in the rectangle size  $2h \times h$ .

Our task - to estimate the maximum number of points that can lie in this box  $2h \times h$ ; thus we estimate and the maximum size of the set  $C(p_i)$ . In this case, when assessing the need to keep in mind that may occur duplicate points.

Recall that  $h$  receive at least the results of the two recursive calls - from the sets  $A_1$  and  $A_2$ , moreover  $A_1$  contains the points to the left of the dividing line and partially on it  $A_2$  - the remaining points of the line section and point to the right of it. For any pair of points  $A_1$ , as well as from  $A_2$  the distance can not be less than  $h$  - otherwise it meant incorrectness recursive function.

To estimate the maximum number of points in the rectangle  $2h \times h$  we will divide it into two square  $h \times h$ , the square of the first otnesem all points  $C(p_i) \cap A_1$ , and the second - all the others, ie  $C(p_i) \cap A_2$ . from the above considerations that in each of the square of the distance between any two points is not less  $h$ .

We show that in each square **no more than four** points. For example, this can be done as follows: We divide the square into four sub-squares with sides  $h/2$ . Then each of these sub-squares can be more than one point (as is even diagonal  $h/\sqrt{2}$  is smaller  $h$ ). Consequently, throughout the square can not be more than 4 points.

So, we have proved that in the rectangle  $2h \times h$  can not be more  $4 \cdot 2 = 8$  points, and hence the size of the set  $C(p_i)$  can not exceed 7 what was required to prove.

## Implementation

We introduce a data structure for storing point (its coordinates and a number) and comparison operators are required 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()* that will calculate the 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 the recursion. It is assumed that prior to her calling the array *a* is already sorted by *x*-coordinate. Recursion passed just two pointers *l*, *r* which indicate that it should seek an answer for *a[l...r]*. If the distance between *r* and *l* is too small, the recursion must be stopped, and perform a trivial algorithm to search for the nearest pair and then sort by subarray *y*-coordinate.

To merge two sets of points obtained by recursive calls, one (ordered by *y*-coordinate), we use the standard function STL *merge()*, and create an auxiliary buffer *t* (one for all recursive calls). (Use *inplace\_merge()* impractical because 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 on to the fractional values, and store in *mindist* the square of the minimum distance.

In the main program cause recursion as follows:

```
sort (a, a+n, &cmp_x);
mindist = 1E20;
rec (0, n-1);
```

## Summary: Search the triangle with minimal perimeter

The algorithm described above interesting generalized to this problem: among a given set of points to choose three different points so that the sum of pairwise distances between them was the lowest.

In fact, to solve this problem, the algorithm remains the same: we divide the field into two halves of the vertical line, call the recursive solution of both halves, choose at least *minper* one found perimeters, build strip thickness *minper*/2, and it sort out all the triangles that can improve response. (Note that the triangle has a perimeter  $\leq \text{minper}$  length of the side  $\leq \text{minper}/2$ .)

## Tasks in the online judges

List of tasks that are reduced to the search for the next two points:

- [UVA 10245 "The Closest Pair Problem"](#) [Difficulty: Low]
- [SPOJ # 8725 CLOPPAIR "Closest Point Pair"](#) [Difficulty: Low]
- [CODEFORCES Team Olympiad Saratov - 2011 "Minimum Amount"](#) [Difficulty: Medium]
- [Google CodeJam 2009 Final "Min Perimeter"](#) [Difficulty: Medium]
- [SPOJ # 7029 CLOSEST "Closest Triple"](#) [Difficulty: Medium]

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: Sep 14 2010 2:07  
EDIT: 6 Jan 2011 1:11

## Transformation of geometric inversion

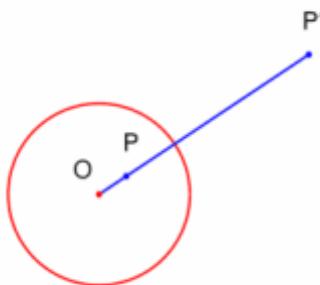
Transformation of geometric inversion (inversive geometry) - a special type of conversion points on the plane. The practical benefit of this transformation is that it often allows to reduce the solution of geometric problems **with the circles** to the solution of the corresponding problem **with straight**, 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 transformations.

### Determination

We fix a circle centered at  $O$  the radius  $r$ . Then the **inversion** point  $P$  on this circle is a point  $P'$  which lies on the line  $OP$ , and the distance imposed a condition:

$$OP \cdot OP' = r^2.$$



If it is assumed that the center of  $O$  the circle coincides with the origin, we can say that the point  $P'$  has the same polar angle as  $P$ , and the distance calculated by the above formula.

In terms of **complex numbers** inversion transformation is expressed simply, if we assume that the center of  $O$  the circle coincides with the origin:

### Contents [hide]

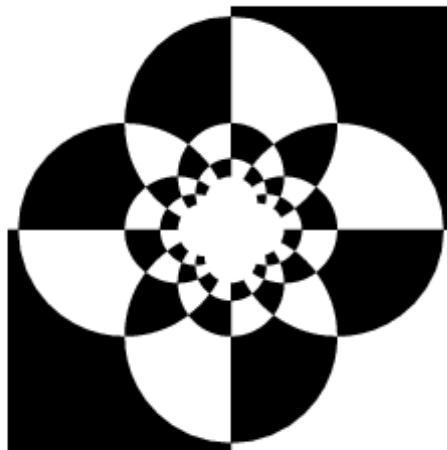
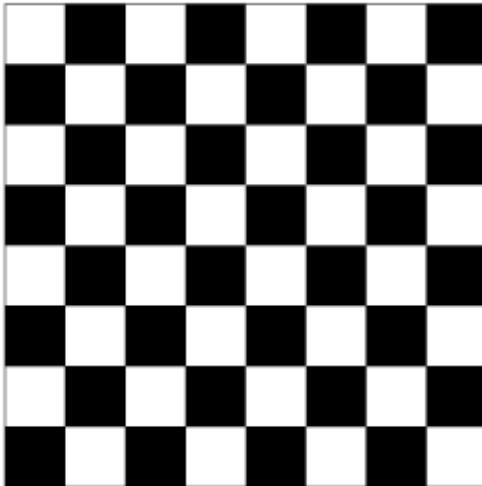
- Transformation of geometric inversion
  - Determination
  - Properties
    - Generalized circle
    - Inversion of the line passing through the point  $O$
    - Inversion line not passing through the point  $O$
    - Inversion of a circle passing through the point  $O$
    - Inversion circumference without passing through  $O$
    - Lemma on equal angles
      - Formulation
      - Proof
      - Consequence of Lemma
    - Conformality
    - Reflection property
  - Practical application
    - Constructing Shapes after inversion
    - The parameters of the circle after inversion
    - Use in evidence: the problem of partitioning points of the circle
    - Used to solve problems of computational geometry
    - Steiner chain
    - The application of the technique: direct Lipkin-Peaucellier

$$z' = r^2 \cdot \frac{z}{|z|^2}.$$

With the help of conjugate elements  $\bar{z}$  can get a simpler form:

$$z' = \frac{r^2}{\bar{z}}.$$

Application of inversion (at mid-board) to the image of the chessboard gives an interesting picture (right):



## Properties

It is obvious that any point lying **on the circle**, which is made with respect to the inversion transformation, the mapping transforms into itself same. Any point lying **inside the circle** moves in **the outer** region, and vice versa. It is believed that the center of the circle into the point "infinity"  $\infty$  and point "infinity" - on the contrary, in the center of the circle:

$$(O)' = \infty, \\ (\infty)' = O.$$

It is evident that repeated application of inversion **draws** its first application - all the points are returned:

$$(P')' \equiv P.$$

## Generalized circle

Generalized circle - it's either a circle or a straight line (it is believed that this is also a circle, but with an infinite radius).

The key property of inversion - that when used generalized circle **always becomes a generalized circle** (assuming conversion of pointwise inversion is applied to all points of the figure).

Now we'll see what happens with lines and circles under the inversion transformation.

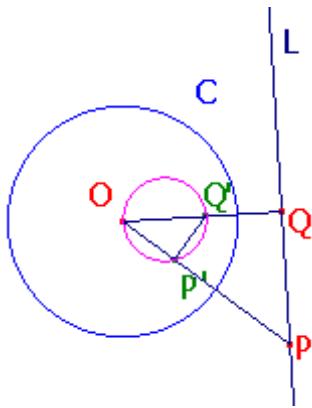
### Inversion of the line passing through the point $O$

It is argued that any line passing through  $O$ , after inversion **does not change**.

In fact, any point on this line, in addition to  $O$  and  $\infty$ , by definition, too, goes to the point of this line (and eventually get the point entirely fill the whole line, since the inversion transformation is reversible). The remaining points  $O$  and  $\infty$  but inversion they pass each other, so the proof is complete.

### Inversion line not passing through the point $O$

It is argued that any such line will move **in a circle** passing through  $O$ .



Consider any point  $P$  on this line, and we also consider the point  $Q$ - to the nearest  $O$  point of the line. It is clear that the segment  $OQ$  is perpendicular to the line, but because they formed the corner  $\angle PQO$ - line.

We now use **Lemma about equal angles**, which we shall prove later, this lemma gives us the equality:

$$\angle PQO = \angle Q'P'O.$$

Consequently, the angle of  $\angle Q'P'O$  the line too. As we take the point  $P$  of any, it turns out that the point  $P'$  lies on the circle, built on  $OQ'$  as diameter. It is easy to understand that in the end, all points on the line will cover the whole circumference of the whole, therefore, the assertion is proved.

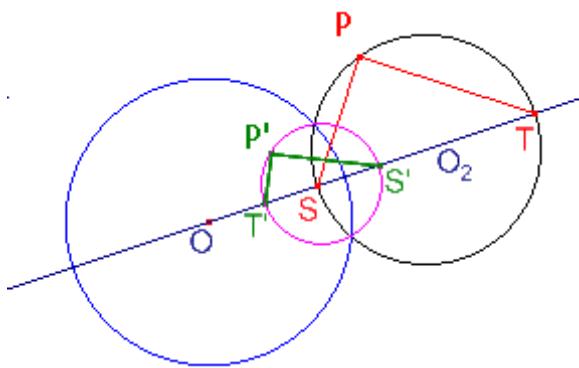
### Inversion of a circle passing through the point $O$

Any such circle will move **in a straight**, not passing through  $O$ .

In fact, this follows immediately from the preceding paragraph, if we remember about the reversibility of inversion.

### Inversion circumference without passing through $O$

Any such ring will move **in a circle**, are still passing through  $O$ .



In fact, consider any such circle  $Z$  centered at  $O_2$ . Connect the center  $O$  and  $O_2$  circle line; this line crosses the circle  $Z$  at two points  $S$  and  $T$  (obviously  $ST$ - diameter  $Z$ ).

Now consider any point  $P$  on the circle  $Z$ . The angle of  $\angle SPT$  the line for any such terms, but the corollary of **lemma equal angles** also must be direct and angle  $\angle S'P'T'$ , which implies that the point  $P'$  lies on the circle, built on the segment  $S'T'$  as diameter. Again, it is easy to understand that all the images  $P'$  eventually cover the circle.

It is clear that this new circle can not pass through  $O$ : otherwise the point  $\infty$  would have to belong to the old circle.

## Lemma on equal angles

This auxiliary property that was used in the above analysis results of inversion.

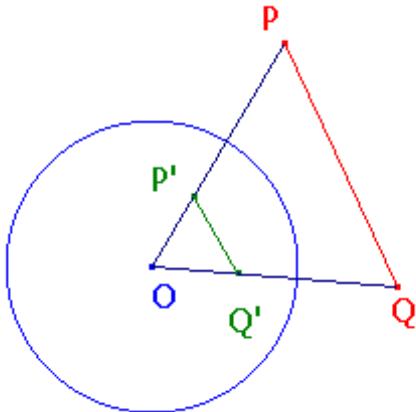
### Formulation

Consider any two points  $P$  and  $Q$ , and apply to them the inversion transformation, we get the point  $P'$  and  $Q'$ . Then the following angles are equal:

$$\begin{aligned}\angle PQO &= \angle Q'P'O, \\ \angle QPO &= \angle P'Q'O.\end{aligned}$$

### Proof

Prove that the triangle  $\triangle PQO$  and the  $\triangle Q'P'O$  like (the order of vertices is important!).



In fact, by the definition of inversion have:

$$OP \cdot OP' = r^2,$$

$$OQ \cdot OQ' = r^2,$$

whence we obtain:

$$\begin{aligned} OP \cdot OP' &= OQ \cdot OQ', \\ \frac{OP}{OQ} &= \frac{OQ'}{OP}. \end{aligned}$$

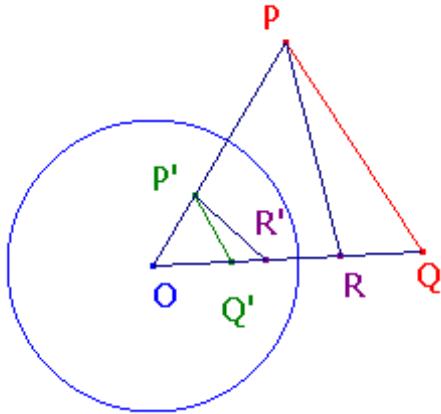
Thus, triangles  $\triangle P Q O$  and  $\triangle Q' P' O$  have a common angle and two sides adjacent thereto are proportional, therefore, these triangles are similar, and therefore the same respective angles.

### Consequence of Lemma

If there are any three points  $P, Q, R$ , with the point  $R$  lies on the segment  $OQ$ , then the:

$$\angle QPR = \angle Q'P'R',$$

and these angles are oriented in different directions (ie, if we consider these two angles as directed, they are of opposite sign).



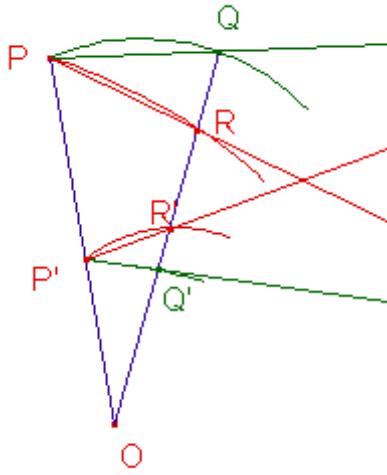
For the proof we note that  $\angle QPR$ - is the difference of two angles  $\angle QPO$  and  $\angle RPO$ , for each of which we can apply Lemma Equal angles:

$$\angle QPR = \angle QPO - \angle RPO = \angle P'Q'O - \angle P'R'O = \angle R'P'Q' = \angle Q'P'R'.$$

In the implementation of the last transition, we have 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 the angles at the points of intersection of the curves**. Here, if the angles considered as oriented, the orientation angles at applying inversion is reversed.



To prove this, consider two arbitrary curves intersecting at a point  $P$  and having it tangents. Let the first curve point will go  $Q$  on the second - a point  $R$  (which we rushed to the limit to  $P$ ).

It is obvious that after the application of the inversion curves will continue to overlap (unless, of course, they do not pass through the point  $O$ , but that we will not be), and their intersection point will be  $P'$ .

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 get:

$$\angle QPR = -\angle Q'P'R',$$

where, under the sign "minus" we mean that the angles are oriented in different directions.

Letting terms  $Q$  and  $R$  to the point  $P$ , we obtain in the limit that this equality - an expression of the angle between the intersecting curves, as required.

## Reflection property

If  $M$  - a generalized circle, when the inversion transformation it is stored only if and when the orthogonal circle with respect to which an inversion is made (and considered to be different).  $MCMC$

The proof of this property is interesting in that it demonstrates the use of geometric inversion care of circles and to simplify the problem.

The first step in the proof will be an indication of the fact that  $M$ , and  $C$  has at least two points of intersection. In fact, the transformation of inversion  $C$  maps the interior of a circle in its appearance, and vice versa. Time  $M$  after the conversion has not changed, it means that it contains a point from both domestic and from the exterior of the circle  $C$ . This implies that the two points of intersection (the one it can not be - it means two touching circles, but in this case, obviously, be the condition can not, be the same as the circumference can not by definition).

Denote one intersection point across  $A$ , the other - through  $B$ . Consider an arbitrary circle centered at the point  $A$ , and perform the conversion inversion it. Note that if and circumference  $C$ , and the generalized circle  $M$  must pass into the intersecting lines. Given the conformal transformation inversion, we obtain that  $M$  and  $M'$  coincide if and

only if the angle between the two intersecting lines line (in fact, the first inversion transformation - relatively  $C$ - changes the direction of the angle between the circles on the opposite, so if the circle coincides with its inversion, the angles between the intersecting lines on both sides must be identical, and equal  $\frac{180}{2} = 90$  degree).

## Practical application

Immediately it should be noted that when used in the calculation must take into account a large **error** introduced by the inversion transformation: fractional numbers may appear very small orders, and usually due to high error inversion method only works well with a relatively small coordinates.

## Constructing Shapes after inversion

In computing software is often more convenient and reliable to use no ready-made formulas for the coordinates and radii of the resulting generalized circles, and to restore every time direct / circle at two points. If the recovery is sufficient to take direct any two points and calculate their images and link line, then the circles it's much more difficult.

If we want to find the circumference, resulting in a direct inversion, according to calculations given above, it is necessary to find the nearest to the center of the inversion point of  $Q$  the line inversion applied to it (getting a certain point  $Q'$ ), and then the circle will have the desired diameter  $OQ'$ .

Suppose now that we want to find the circumference, resulting in an inversion of another circle. Generally speaking, the new center of the circle - not the same way the old center of the circle. To determine the center of the new circle can take this concept: to conduct through the inversion center and the center of the old circle line, see its point of intersection with the old circle - let it be the point  $S$  and  $T$ . The segment  $ST$  of the old circle diameter forms, and is easy to understand that after this inversion interval will still form diameter. Consequently, the center of the new circle can be defined as the arithmetic mean of the points  $S'$  and  $T'$ .

## The parameters of the circle after inversion

Required for a given circle (from the known coordinates of its center  $(x_0, y_0)$  and radius  $r_0$ ) to determine in what kind of circle it will go after the inversion with respect to a circle with center  $(x_c, y_c)$  and radius  $r$ .

If we solve the problem described in the previous paragraph, but want to get a closed form solution.

Response appears as 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

$$s = \frac{r^2}{(x_0 - x_c)^2 + (y_0 - y_c)^2 - r_0^2}.$$

Mnemonic these formulas can remember this: the center of the circle becomes "almost" as to transform inversion, only in the denominator in addition  $|z|^2 = (x_0 - x_c)^2 + (y_0 - y_c)^2$  have yet subtracted  $r_0^2$ .

We derive these formulas exactly as described in the preceding paragraph algorithm: find an expression for the two diametrical points  $S$  and  $T$  then applied to them inversion, and then taken the arithmetic mean of their origin. Similarly, we can calculate the radius as half the length of the segment  $ST$ .

## Use in evidence: the problem of partitioning points of the circle

Given  $2n$  different points on the plane, as well as any point  $O$  that is different from all the others. Prove that there exists a circle passing through the point  $O$ , such that the inside and outside of it will be based on the same number of points of the set, ie, to  $n$  pieces.

To prove proizvedëm transform inversion relative to the selected point  $O$  (with any radius, for example  $r = 1$ ). Then the desired circle will fit right, not passing through  $O$ . And on one side of the line is the half-plane corresponds to the inner circle, and on the other - appropriate appearance. It is understood that there will always be such a line, which divides the set of  $2n$  points into two halves by  $n$  points and not through the point  $O$  (e.g., a line can be obtained by turning the whole picture at any angle so that nor in any of the considered  $2n + 1$  point is not coincided coordinates  $x$ , and then just taking a vertical line between the  $n$ th and  $n + 1$ th points). This line corresponds to the desired circle passing through the point  $O$  and, therefore, the assertion is proved.

## Used to solve problems of computational geometry

A remarkable property of geometric inversion - that in many cases it allows simplified geometrical problem posed by replacing consideration circles only direct examination.

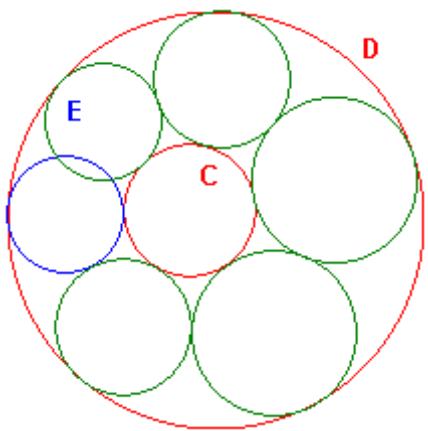
Ie if the problem is quite complicated operations with various circles, it makes sense to apply to the input data inversion transformation, try to solve the problem of obtaining modified without circles (or a smaller number of them), and then re-use inversion to obtain the solution of the original problem.

An example of this problem is described in the next section.

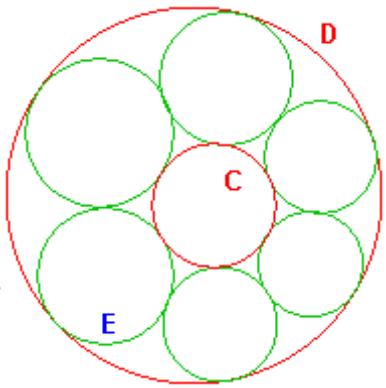
## Steiner chain

Given two circles  $C$  and  $D$  one is in the interior of the other. Then, the third circle is drawn  $E$  concerning these two circles, then the iterative process is started every time a new circle is drawn so that it **touches** the preceding drawn, and the first two. Sooner or later, another draw a circle crosses with some of the already delivered, or at least touches it.

Case intersection:



If you touch:



Accordingly, our goal - to put **as many** circles so that the intersection (ie, 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 tangent to the circle, then everything on the circle is uniquely placed.

In case of contact we obtain a chain of circles is called **a Steiner chain**.

With this so-called chain due **approval Steiner** (Steiner's porism): if there is at least one chain Steiner (ie, there is a corresponding provision relating to the starting circle, leading to a chain of Steiner), then for any other choice concerning the starting circle will also receive chain Steiner, with the number of circles in it will be the same.

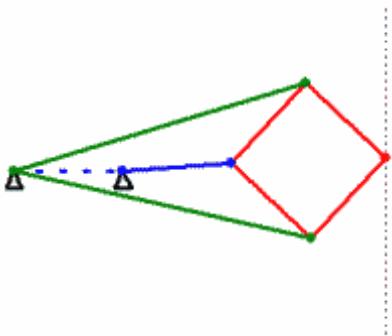
From this statement it follows that in solving the problem of maximizing the number of circles the answer does not depend on the position of the first set of the circle.

**Proof** and constructive algorithm for solving the following. Note that the problem has a very simple solution when the center 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 the circles have the same radius, and their number and coordinates of the centers can be calculated by a simple formula.

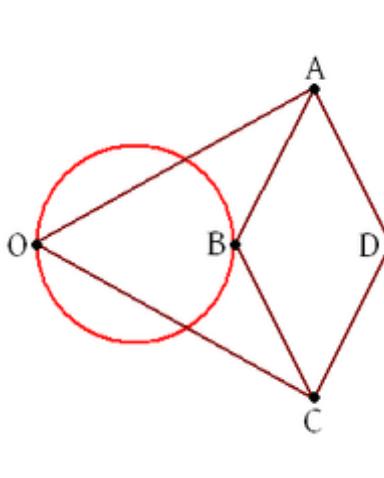
To go to this simple situation of any supplied to the input, apply the transformation inversion with respect to a circle. We need to center the inner circle and moved coincided with the center of the outside, so look for a point with respect to which we will take the inversion, it is necessary only to the line connecting the centers of the circles. Using the formulas for the coordinates of the circle center after applying inversion, can equate to the position of the center of inversion, and to solve this equation. Thus, we are of any situation can go to a simple, symmetric case, and, solving the problem for him, re-apply the transformation inversion and obtain the solution of the original problem.

## The application of the technique: direct Lipkin-Peaucellier

For a long time the task of converting the circular (rotational) motion into linear remained very challenging in engineering, managed to find at best approximate solutions. It was only in 1864 an officer of the French Army Corps of Engineers Nikola Peaucellier Charles (Charles-Nicolas Peaucellier) and in 1868, a student of Chebyshev Lipman Lipkin (Lipman Lipkin) invented a device based on the idea of geometric inversion. Device is called "direct-Lipkin Peaucellier" (Peaucellier-Lipkin linkage).



To understand the operation of the device, note it on several points:



Point  $B$  performs rotational movement around the circumference (red), whereby the point  $D$  necessary for moving the line (blue). Our task - to prove that point  $D$  - the essence of the inversion point  $B$  relative to the center  $O$  with a certain radius  $r$ .

Formalize the condition of the problem: that the point  $O$  rigidly fixed, lengths  $OA$  and  $OC$  the same, and also coincides quartet pieces  $AB, BC, CD, DA$ . A point  $B$  moves along a circle passing through the point  $O$ .

For the proof, we note first that point  $O, B$  and  $D$  lie on one line (this follows from the equality of triangles). We denote by  $P$  the intersection of the segments  $AC$  and  $BD$ . We introduce the following notation:

$$OB = x, \quad BP = y, \quad AP = h.$$

We need to show that the value of  $OB \cdot OD = \text{const}$ :

$$OB \cdot OD = x(x + 2y) = x^2 + 2xy.$$

Pythagorean theorem we get:

$$\begin{aligned}OA^2 &= (x+y)^2 + h^2, \\AD^2 &= y^2 + h^2.\end{aligned}$$

Take the difference between these two values:

$$OA^2 - AD^2 = x^2 + 2xy = OB \cdot OD.$$

Thus, we have proved that  $OB \cdot OD = \text{const}$ , which means that  $D$ - inversion point  $B$ .

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 15 Dec 2011 23:31  
 EDIT: 16 Dec 2011 0:20

## Search common tangents to two circles

Given two circles. You want to find all of their common tangents, ie all such lines, which relate to both circles at the same time.

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 for finding tangent to a circle passing through a given point.

### Contents [\[hide\]](#)

- Search common tangents to two circles
  - Number of common tangents
  - Algorithm
  - Implementation

## Number of common tangents

Just note that we do not consider **the degenerate** cases when the circle are the same (in this case, they have infinitely many common tangents), or one circle is inside the other (in this case they do not have common tangents, or if the circles are tangent, there is one common tangent).

In most cases, two circles are **four** common tangents.

If the circles **touch**, they will have three general industrial tangent, but it can be understood as a degenerate case: as if the two tangents coincide.

Moreover, the algorithm described below will work in the case where one or both of the radius of the circle have zero: in this case, respectively, one or two common tangent.

To summarize, we, except as described in the early cases, we will always look for **the 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 it is not, then this can be achieved by a simple shift of the entire picture, and then find a solution - a shift received direct feedback.)

We denote by  $r_1$  and  $r_2$  the radii of the first and second circles, and through  $v$  - the coordinates of the center of the second circle (point  $v$  differs from the origin, because we do not consider the case where the circumference of the same, or one circle is inside the other).

To solve the problem'll get to it purely **algebraically**. We want to find all direct type  $ax + by + c = 0$ , which lie at a distance  $r_1$  from the origin, and the distance  $r_2$  from point  $v$ . Furthermore, we impose the condition normalization direct sum of the squares of the

coefficients  $a$  and  $b$  must be equal to one (it must, otherwise the same line will correspond to an infinite number of types of representations  $ax + by + c = 0$ ). 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 modules, we note that in all there are four ways to reveal the modules in the system. All of these methods can be considered the 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 are moving 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 must solve the system:

$$\begin{cases} a^2 + b^2 = 1, \\ c = d_1, \\ a \cdot v_x + b \cdot v_y + c = d_2. \end{cases}$$

The solution of this system is reduced to solving a quadratic equation. We omit all the cumbersome calculations, and once we give a ready answer:

$$\begin{cases} a = \frac{(d_2 - d_1)v_x \pm v_y \sqrt{v_x^2 + v_y^2 - (d_2 - d_1)^2}}{v_x^2 + v_y^2}, \\ b = \frac{(d_2 - d_1)v_y \mp v_x \sqrt{v_x^2 + v_y^2 - (d_2 - d_1)^2}}{v_x^2 + v_y^2}, \\ c = d_1. \end{cases}$$

Total we've got 8 solutions instead 4. But it is easy to understand, in what place there are extra 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  $\pm r_1$  and  $\pm r_2$ , is clear: we actually sort out which side of each circle is a straight line. Therefore, two ways, arising in the solution of the latter system, redundant: just select one of the two solutions (only, of course, in all four cases, it is necessary to choose the same family of solutions).

Recently, we have not examined - this **move straight as** in the case where the circumference is not located initially at the origin. But here everything is simple: from the linear equation follows directly that the coefficient  $c$  is necessary to subtract the value  $a \cdot x_0 + b \cdot y_0$  (where  $x_0$  and  $y_0$  - the coordinates of the original center of 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 solution 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;
}

```

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 26 Mar 2012 1:00  
 EDIT: Mar 26 2012 1:00

## Search pair of intersecting segments algorithm swept out by lines of O (N log N)

Given  $n$  set of line segments. Required check whether intersect with each other at least two of them. (If the answer is yes - then bring this pair of intersecting segments; among several responses sufficient to choose any of them.)

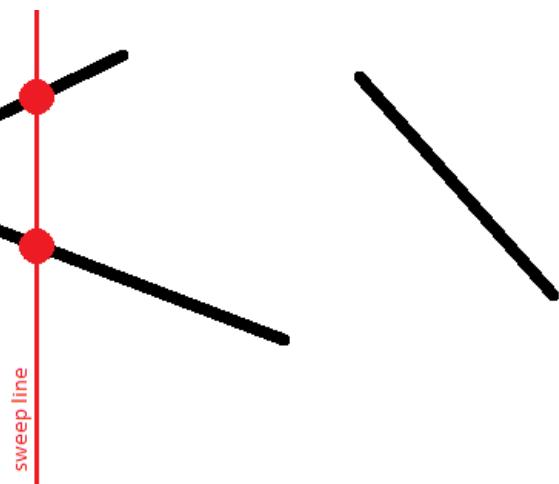
A naive algorithm for solving - iterate over  $O(n^2)$  all pairs of segments and check for each pair intersect or not. This article describes an algorithm with running time  $O(n \log n)$ , which is based on the principle of **scanning (sweep out) direct** (in English: "sweep line").

### Contents [hide]

- Search pair of intersecting segments algorithm swept out by lines of O (N log N)
  - Algorithm
  - Implementation

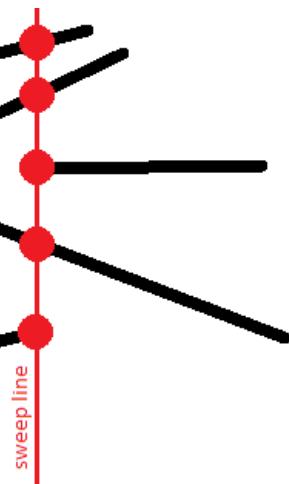
### Algorithm

Draw a vertical line mentally  $x = -\infty$  and begin to move this line to the right. In the course of its movement this line will meet with the segments, with at any given time, each segment will overlap with our direct at one point (we are going to assume that there is no vertical segments).

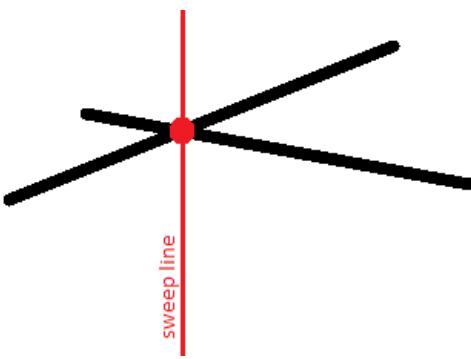


Thus, for each segment at some point in time it will be a point on the scanning line, then straight ahead and will move this point, and, finally, at some point in the segment will disappear from the line.

We are interested in the **relative order of the segments** vertically. Namely, we will keep a list of segments that cross the scanning line at a time, where the segments will be sorted by their  $y$ -coordinate on the scanning line.



This order is interesting in that overlapping segments will have the same  $y$ -coordinate at least one time:



We formulate the key statements:

- To find the intersecting pairs is sufficient to consider for each fixed position scanning direct **only adjacent segments**.
- It suffices to consider the scanning line is not valid in all possible positions  $(-\infty \dots +\infty)$ , but **only in those positions where there are new pieces of old or disappear**. In other words, it is sufficient to limit ourselves only to the provisions of equal abscissas-end segments.
- When a new segment is enough **to insert** it in the right place on the list prepared for the previous scanning line. Check for intersection is necessary **only adds segment with its immediate neighbors in the list of the top and bottom**.
- With the disappearance of the segment is sufficient **to remove** it from the current list. Thereafter, it is necessary **to check the intersection with the upper and lower neighbors** in the list.
- Other changes in the order of the segments in the list, other than those described, does not exist. Other checks on the crossing is not necessary to produce.

To understand the truth of these assertions quite the following remarks:

- Two disjoint segment never change their **relative order**.  
In fact, 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 identical  $y$ -coordinates of two disjoint intervals also can not.
- From this it follows that at the time the segment we can find a position in the queue for this segment, this segment and more swap queue does not have: his **order with respect to other segments in the queue will not change**.
- Two intersecting segment at the time point of its intersection will be **neighbors** to each other in turn.
- Therefore, in order to find a pair of intersecting segments is sufficient to check only at the intersection of all the pairs of segments that sometime during the scanning motion straight at least once **were neighbors of each other**.  
Easy to see that it is enough just to check the added segment with their 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 at a fixed position scanning the line, we **first** need to make **the addition** of all the segments are emerging, and only **then - the removal** of all endangered segments here.

Thus, we will not miss the intersection of segments on top: ie such cases, when the two segments have a common vertex.

- Note that the **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 comments, 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 some other open at this time segment (including vertical), it will be detected.

In what place queued vertical segments? After the vertical segment has no one specific  $y$ -coordinates of extending for a period of at  $y$ -coordinate. But it is easy to understand that as  $y$  the coordinates can take any coordinate of this segment.

Thus, the whole algorithm will make no more  $2n$  tests on the intersection of a pair of segments, and make  $O(n)$  transactions with burst lengths (for  $O(1)$  operations in the appearance and disappearance of each segment).

The resulting **asymptotic behavior** of the algorithm is thus  $O(n \log n)$ .

## Implementation

We give full realization of this 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 feature here - `solve()` which returns the number of intersecting segments found, or  $(-1, -1)$  if there is no intersection.

Check on the intersection of two segments performed the function `intersect()`, using [an algorithm based on oriented area of the triangle](#).

Of all segments in a global variable `s`- `set < event >`. Iterators indicating the position of each segment in the queue (for easy removal from the queue lengths) are stored in a global array `where`.

Also introduced two auxiliary functions `prev()` and `next()` which returns an iterator to the previous and next elements (or `end()`, if one does not exist).

The constant `EPS` represents the error of comparing two real numbers (basically it is used when checking the two segments at the intersection).

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:35  
EDIT: 26 Apr 2012 2:00

## Z-line function and its calculation

Given a string  $s$  of length  $n$ . Then the **Z-function** ("Z-function") from this line - an array length  $n$ ,  $i$ th element of which is the largest number of characters starting from a position  $i$  coinciding with the first character  $s$ .

In other words  $z[i]$  - this is the greatest common prefix string  $s$  and its  $i$ th suffix.

**Note**. In this paper, in order to avoid uncertainty, we assume a string of 0-indexed - ie the first character has an index 0, and the last -  $n - 1$ .

The first element of Z-function  $z[0]$ , generally considered uncertain. In this article, we will assume that it is equal to zero (although none in the algorithm or in the reduced implementation, this does not change anything).

This article provides an algorithm for calculating the Z-function of time  $O(n)$ , as well as various applications of this algorithm.

### Contents [hide]

- Z-line function and its calculation
  - Examples
  - A trivial algorithm
  - An efficient algorithm for calculating the Z-function
  - Implementation
  - Asymptotic behavior of the algorithm
  - Applications
    - Search substring
    - The number of different substrings in a string
    - Compression line
  - Tasks in the online judges

## Examples

Let us give an example computed Z-function for a few lines:

- "aaaaa":

$$\begin{aligned} z[0] &= 0, \\ z[1] &= 4, \\ z[2] &= 3, \\ z[3] &= 2, \\ z[4] &= 1. \end{aligned}$$

- "aaabaab":

$$\begin{aligned} z[0] &= 0, \\ z[1] &= 2, \\ z[2] &= 1, \\ z[3] &= 0, \end{aligned}$$

$$\begin{aligned}z[4] &= 2, \\z[5] &= 1, \\z[6] &= 0.\end{aligned}$$

- "abacaba":

$$\begin{aligned}z[0] &= 0, \\z[1] &= 0, \\z[2] &= 1, \\z[3] &= 0, \\z[4] &= 3, \\z[5] &= 0, \\z[6] &= 1.\end{aligned}$$

## A trivial algorithm

The formal definition can be represented as the following elementary implementation of  $O(n^2)$ :

```
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 simply for each position  $i$  iterate answer for it  $z[i]$ , starting from scratch, and as long as we do not find a mismatch or not we reach the end of the line.

Of course, this implementation is too inefficient, let's move now to the construction of an efficient algorithm.

## An efficient algorithm for calculating the Z-function

To obtain an efficient algorithm to calculate the value will  $z[i]$  in turn - by  $i = 1$  up  $n - 1$  and try wherein when calculating the next value of  $z[i]$  maximum use of the calculated values.

We call for the sake of brevity, the substring that matches the prefix of  $s$ , **coincidence segment**. For example, the value of the desired Z-function  $z[i]$ - a long stretch of matches, starting at the position  $i$  (and end it will be in a position  $i + z[i] - 1$ ).

To do this, we will support the coordinates  $[l; r]$  of the rightmost segment overlap

, ie of all detected segments will keep the one that ends just right. In a sense, the index  $r$ - this is such a boundary, to which our line has 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:

- $i > r$ - i.e. the current position is **outside** of what we have already processed.

Then we will look **trivial algorithm** , ie, a sampling values , etc. Note that, in the end, if you will , we will be required to update the coordinates of the rightmost segment - as would be guaranteed more  $.z[i] z[i] = 0 z[i] = 1 z[i]$   
 $> 0[l; r]i + z[i] - 1r$

- $i \leq r$ - i.e. the current position is inside the segment matches  $[l; r]$ .

Then we can use the already calculated **the previous** value of Z-functions to initialize the value  $z[i]$ is not zero, but somehow the greatest possible number.

To do this, we note that the substring  $s[l \dots r]$ and **match** . This means that for an initial approximation we can take the value of the corresponding segment , namely value  $.s[0 \dots r - l] z[i] s[0 \dots r - l] z[i - l]$

However, the value  $z[i - l]$ may be too large: so that when applied to position  $i$ it "come out" of bounds  $r$ . This should not be allowed because about the characters to the right  $r$ , we know nothing, and they may differ from those required.

Let us give **an example of** such a situation, the example of the line:

”aaaabaa”

When we get to the last position (  $i = 6$ ), the current will be the rightmost segment  $[5; 6]$ . Position 6in view of this segment will match the position  $6 - 5 = 1$ , in which the answer is  $z[1] = 3$ . It is obvious that such a value to initialize the  $z[6]$ impossible, it is absolutely incorrect. The maximum value of what we could initialize - this is 1because it is the largest value that does not climbs beyond the interval  $[l; r]$ .

Thus, as an **initial approximation** to  $z[i]$ safely take only an expression:

$$z_0[i] = \min(r - i + 1, z[i - l]).$$

Initialize  $z[i]$ this value  $z_0[i]$ , we will again act on **the trivial algorithm** - because after the border  $r$ , in general, is able to detect the segment continued coincidence predict that only one previous values Z-function, we could not.

Thus, the whole algorithm is a two cases that actually differ only in **the initial value**  $z[i]$  in the first case it is assumed to be zero, and the second - is determined by the previous values for this formula. After that, the two branches of the algorithm reduce to the **trivial algorithm** , starting immediately with the specified initial value.

The algorithm turned out very simple. Despite the fact that every time  $i$ there anyway performed trivial algorithm - we have made considerable progress, receiving algorithm runs in linear time. Why this is so, consider the following, after we give the

implementation of the algorithm.

## Implementation

Implementation get very succinctly:

```
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];
        if (i+z[i]-1 > r)
            l = i, r = i+z[i]-1;
    }
    return z;
}
```

Comment on this implementation.

Everything is packed in a decision function that returns an array of on line length  $n$ -the calculated Z-function.

The array  $z$  is initially filled with zeros. Current rightmost segment is assumed to be a match  $[0; 0]$ , ie deliberately small segment, which will not get none  $i$ .

Inside the cycle  $i = 1 \dots n - 1$ , we first algorithm described above to determine the initial value  $z[i]$  it will either zero or calculated based on the above formulas.

After that, the trivial algorithm that attempts to increase the value of  $z[i]$  as much as possible.

In the end, the current update of the rightmost segment coincidence  $[l; r]$ , of course, if this update is required - ie if  $i + z[i] - 1 > r$ .

## Asymptotic behavior of the algorithm

We prove that the above cited algorithm works 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 - only constant operations performed  $O(n)$  again.

We show that **each iteration** of this cycle while will lead to an increase in the right border  $r$  by one.

For this, we consider both branches of the algorithm:

- $i > r$

In this case, a cycle `while` will not make a single iteration (if  $s[0] \neq s[i]$ ), or else make several iterations, moving each time one character to the right, starting at the position  $i$ , and then - right border  $r$  necessarily updated.

Since  $i > r$ , then, we find that indeed each iteration of the loop increases the value of the new  $r$  unit.

- $i \leq r$

In this case, we initialize the value of the above formulas  $z[i]$  by some number  $z_0$ . Compare this to the initial value  $z_0$  with the value  $r - i + 1$ , we get three options:

- $z_0 < r - i + 1$

We prove that in this case no one iteration cycle `while` will not.

It is easy to prove, for example, by contradiction: if the cycle `while` made at least one iteration, it would mean that a certain value which we  $z_0$  have been inaccurate, less than the length of this match. But since row  $s[l \dots r]$  and  $s[0 \dots r - l]$  the same, then it means that the position  $z[i - l]$  stands the wrong value: less than it should be.

Thus, in this embodiment of the correctness values  $z[i - l]$ , and the fact that it is smaller  $r - i + 1$ , it follows that the value coincides with the desired value  $z[i]$ .

- $z_0 = r - i + 1$

In this case, the cycle `while` can make several iterations, but each of them will lead to an increase in the value of the new  $r$  unit: because we compare the first character is  $s[r + 1]$  who climbs out of the segment  $[l; r]$ .

- $z_0 > r - i + 1$

This option is basically impossible by the definition  $z_0$ .

Thus, we have proved that each iteration of the inner loop leads to the promotion of the pointer  $r$  to the right. Since  $r$  there could be more  $n - 1$ , it means that this cycle will likely not more  $n - 1$  iteration.

As the rest of the algorithm is obviously working for  $O(n)$ , we have proved that the whole algorithm to compute Z-function is performed in linear time.

## Applications

We consider several applications of Z-functions for specific tasks.

These applications will be largely similar applications [prefix function](#).

## Search substring

To avoid confusion, let's call a single line  $t$ , the other - a model  $p$ . Thus, the challenge is to find all occurrences of the pattern  $p$  in the text  $t$ .

To solve this problem, we form a line  $s = p + \# + t$ , 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 each  $i$  segment in  $[0; \text{length}(t) - 1]$  the respective value  $z[i + \text{length}(p) + 1]$  can be understood, if the sample is included  $p$  in the text  $t$ , starting from the position  $i$ : if this value is equal to the Z-function  $\text{length}(p)$ , then yes, included, otherwise - no.

Thus, the asymptotic behavior of the solution is obtained  $O(\text{length}(t) + \text{length}(p))$ . Memory consumption has the same asymptotic behavior.

## The number of different substrings in a string

A string  $s$  length  $n$ . Requires her to count the number of different substrings.

We will solve this problem iteratively. Namely, learn, knowing the current number of different substrings recalculate this number, when added to the end of one symbol.

So, let  $k$  - the current number of different substrings  $s$ , and we add to the end of the symbol  $c$ . Obviously, the result could appear some new substring ending in this new symbol  $c$  (namely, all the strings ending with this symbol, but never met before).

Take a string  $t = s + c$  and invert it (we write the characters in reverse order). Our task - to calculate how many lines  $t$  such prefixes that are not found it anywhere else. But if we think of the line  $t$  Z-function and find its maximum value  $z_{\max}$ , then, obviously, in the line  $t$  occurs (not at the beginning), its prefix length  $z_{\max}$ , but not greater length. Clearly, shorter length prefixes already occur exactly in it.

So, we have found that the number of new substrings that appear when you append the character  $c$  as well  $\text{len} - z_{\max}$ , where  $\text{len}$  - the current length of the string after the character attribution  $c$ .

Consequently, the asymptotic behavior of solutions for line length  $n$  is  $O(n^2)$ .

It is worth noting that in exactly the same can be recalculated for  $O(n)$  different number of substring and append a character to the beginning, as well as removing characters from the end or the beginning.

## Compression line

A string  $s$  length  $n$ . You want to find the shortest it "compressed" representation, ie, find a line  $t$  of shortest length that  $s$  can be represented as a concatenation of one or more copies  $t$ .

For solutions count function Z-line  $s$ , and find the first position  $i$  such that  $i + z[i] = n$ , while  $n$  divided into  $i$ . Then the string  $s$  can be compressed to a length

of string  $i$ .

Proof of such a decision does not differ from that of the solution using [a prefix function](#) .

## Tasks in the online judges

A list of tasks that can be solved using the Z-function:

- [UVA # 455 "Periodic Strings"](#) [Difficulty: Medium]
- [UVA # 11022 "String Factoring"](#) [Difficulty: Medium]

# Prefix function. Algorithm Knuth-Morris-Pratt

## Prefix function. Determination

Given a string  $s[0 \dots n - 1]$ . For it is necessary to calculate the prefix function, ie, an array of numbers  $\pi[0 \dots n - 1]$ , where  $\pi[i]$  is defined as follows: it is such a maximum length of the longest proper suffix substring  $s[0 \dots i]$  that matches its prefix (suffix own - so do not coincide with the whole line). In particular, the value  $\pi[0]$  is set to zero.

The mathematical 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:

- in line "a" no non-trivial prefix that matches the suffix;
- in line "ab" is no trivial prefix that matches the suffix;
- in line "abc" no non-trivial prefix that matches the suffix;
- in line "abca" prefix length 1 coincides with the suffix;
- in line "abcab" prefix length 2 coincides with the suffix;
- in line "abcabc" prefix length 3 coincides with the suffix;
- in line "abcabcd" no non-trivial prefix that matches the suffix.

Another example - for the string "aabaaaab" it is equal to: [0, 1, 0, 1, 2, 2, 3].

## A trivial algorithm

Immediately following the definition, it is possible to write such an algorithm for computing 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  $O(n^3)$  that too slow.

## Contents [hide]

- Prefix function. Algorithm Knuth-Morris-Pratt
  - Prefix function. Determination
  - A trivial algorithm
  - An efficient algorithm
    - The first optimization
    - The second optimization
    - The final algorithm
    - Implementation
  - Applications
    - Search substring. Algorithm Knuth-Morris-Pratt
    - Counting the number of occurrences of each prefix
    - The number of different substrings in a string
    - Compression line
    - The construction of an automaton from a prefix function
  - Tasks in the online judges

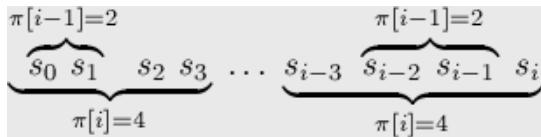
## An efficient algorithm

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 substring).

## The first optimization

The first important point - the value  $\pi[i + 1]$  is not more than one unit exceeds the value  $\pi[i]$  for each  $i$ .

Indeed, otherwise if  $\pi[i + 1] > \pi[i] + 1$ , we consider this suffix ending in the gap  $i + 1$  having a length and  $\pi[i + 1]$  - removing the last symbol from it, we obtain a suffix ending in the gap  $i$  having a length and  $\pi[i + 1] - 1$  that is better  $\pi[i]$ , i.e. a contradiction. Illustration of this contradiction (in this example,  $\pi[i - 1]$  must be equal to 3):



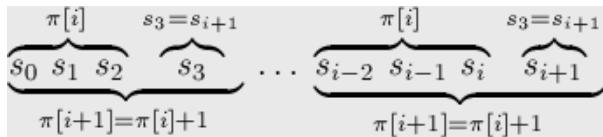
(In this scheme, the upper braces denote two identical substring of length 2, the lower braces - two identical substring of length 4)

Thus, the transition to the next position the next element prefix function could either increase by one, or do not change or reduced by any amount. This fact allows us to reduce the asymptotic behavior  $O(n^2)$  - as in one step value can grow up to one, the total for the whole line could have happened maximum  $n$  increases by one, and as a result (because the value could never be less than zero) maximum  $n$  reductions. That will have a  $O(n)$  string comparisons, ie we have already reached the asymptotic behavior  $O(n^2)$ .

## The second optimization

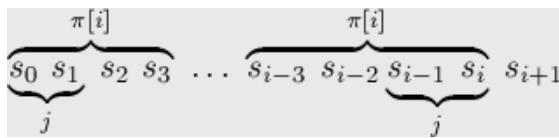
Come on - **get rid of the obvious comparisons substrings**. To do this, try to maximize the use of information calculated in the previous steps.

Thus, suppose we have calculated the value of the prefix function  $\pi[i]$  for some  $i$ . Now, if  $s[i + 1] = s[\pi[i]]$  we can say with certainty that  $\pi[i + 1] = \pi[i] + 1$  it illustrates the scheme:



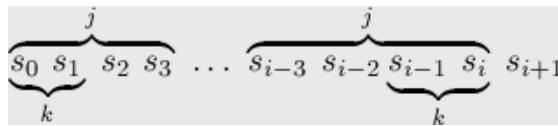
(In this scheme, again the same curly brackets denote the same substring)

Now suppose that, on the contrary, it appears that  $s[i + 1] \neq s[\pi[i]]$ . Then we have to try to try substring at length. In order to optimize I would like to go directly to a (maximum) length  $j < \pi[i]$  that is still running the prefix property is in position  $i$ , ie  $s[0 \dots j - 1] = s[i - j + 1 \dots i]$ :



Indeed, when we find such a length  $j$ , then we will again be sufficient to compare the characters  $s[i + 1]$  and  $s[j]$  - if they coincide, we can argue that  $\pi[i + 1] = j + 1$ . Otherwise, we will have to again find a smaller (next largest) value  $j$ , for which the prefix property, and so on. It may happen that such values  $j$  will end - it happens when  $j = 0$ . In this case  $s[i + 1] = s[0]$ , then  $\pi[i + 1] = 1$ , otherwise  $\pi[i + 1] = 0$ .

Thus, the general scheme of the algorithm we already have, there is only unresolved question of effective finding such lengths  $j$ . We pose this question formally, at the current length  $j$  and position  $i$  (for which the prefix property, that is  $s[0 \dots j - 1] = s[i - j + 1 \dots i]$ ) you want to find the largest  $k < j$  for which is still running the prefix property:



After such a detailed description is almost suggests that this value  $k$  is none other than the value of the prefix function  $\pi[j - 1]$ , which has been calculated previously (subtraction unit appears due to the 0-indexed rows). Thus, to find the lengths  $k$  we can for  $O(1)$  each.

## The final algorithm

So, we finally constructed an algorithm that does not contain explicit string comparisons and performs  $O(n)$  actions.

We give here a summary chart:

- Read the values of the prefix function  $\pi[i]$  will in turn: from  $i = 1$  to  $i = n - 1$  (meaning  $\pi[0]$  simply assign a zero).
- To calculate the current value of  $\pi[i]$  the variable we lead  $j$  for the length of the current sample under consideration. Originally  $j = \pi[i - 1]$ .
- Test the sample length  $j$ , which compare the characters  $s[j]$  and  $s[i]$ . If they match - that believe  $\pi[i] = j + 1$  and go to the next index  $i + 1$ . If the characters are different, then reduces the length  $j$  by putting it equal  $\pi[j - 1]$ , and repeat this step from the beginning of the algorithm.
- When we got to the length  $j = 0$  and have not found a match, then stop the process of sorting the samples and believe  $\pi[i] = 0$  and go to the next index  $i + 1$ .

## Implementation

The algorithm in the end turned out to be 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) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])  ++j;
        pi[i] = j;
    }
    return pi;
}
```

As you can see, this algorithm is **an online algorithm**, ie, it processes the data in the course of income - it is possible, for example, to read a string one character and immediately process the character, finding the answer to the next position. The algorithm requires the storage of the previous line and the calculated values of the prefix function, however, is easy to see, if we know in advance the maximum value that can take the prefix on the entire line, it will be sufficient to store only one more than the number of the first character and values prefix function.

## Applications

### Search substring. Algorithm Knuth-Morris-Pratt

This problem is a classical application of prefix function (and, indeed, it was discovered in this

connection).

Given text  $t$  and line  $s$ , you want to find and display the positions of all occurrences of the string  $s$  in the text  $t$ .

Denote for convenience through  $n$  the length of the string  $s$ , and through  $m$  - the length of the text  $t$ .

We form a line  $s + \# + t$ , where the symbol  $\#$  - is a separator, which should not occur anywhere else. Count for this line prefix function. Now consider its value, except the first  $n + 1$  (which, apparently, are string  $s$  and delimiter). By definition, the value  $\pi[i]$  displays of the longest length of the substring, ending in the position  $i$  and coincides with the prefix. But in this case  $\pi[i]$  - in fact, most of the length of the block matching with the string  $s$  and ending at the position  $i$ . More than  $n$  this length can not be - at the expense of the separator. But equality  $\pi[i] = n$  (where it is achieved), means that the position of  $i$  the desired ends occurrence of the string  $s$  (just do not forget that all positions are measured in the bonded line  $s + \# + t$ ).

Thus, if at some positions  $i$  proved  $\pi[i] = n$ , in the position of  $i - (n + 1) - n + 1 = i - 2n$  the line  $t$  begins the next occurrence of the string  $s$  in a string  $t$ .

As mentioned in the description of the algorithm for calculating the prefix function, if it is known that the value of the prefix function will not exceed a certain value, it is sufficient to store the entire string and not a prefix function, but only its beginning. In our case, this means that you need to keep in memory only the string  $s + \#$  and the value of the prefix functions on it, and then read one character string  $t$  and count the current value of the prefix function.

Thus, the algorithm Knuth-Morris-Pratt solves this problem in  $O(n + m)$  time and  $O(n)$  memory.

## Counting the number of occurrences of each prefix

Here we consider two problems at once. A string  $s$  length  $n$ . The first option is required for each prefix  $s[0 \dots i]$  count how many times it occurs in the very same line  $s$ . In the second version of the problem is given a different row  $t$ , and is required for each prefix  $s[0 \dots i]$  count how many times it occurs in  $t$ .

We solve the first problem first. We consider in any position  $i$  value prefix function in it  $\pi[i]$ . By definition, it means that the position of  $i$  the entry ends with a prefix  $s$  length  $\pi[i]$ , and no more prefix end up in the position  $i$  can not. At the same time, a gap  $i$  may be terminated, and the occurrence of prefixes shorter lengths (and, obviously, not necessarily length  $\pi[i] - 1$ ). However, it is easy to see, we have come to the same question, to which we have already answered when considering the algorithm for computing prefix function: for a given length  $j$  must say, some of the longest proper suffix it coincides with its prefix. We have found that the answer to this question will be  $\pi[j - 1]$ . But then in this problem, if in the position of  $i$  occurrence of the string ends in length  $\pi[i]$ , which coincides with the prefix, then  $i$  also ends occurrence of a string length  $\pi[\pi[i] - 1]$ , which coincides with the prefix, and for her apply the same reasoning, so  $i$  also ends and the length of the entry  $\pi[\pi[\pi[i] - 1] - 1]$ , and so on (until the index becomes zero). Thus, to calculate the response we must perform 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 we have for each value of the prefix function first count how many times he had met in the array  $\pi[]$ , and then felt such a kind of dynamics: if we know that the prefix length  $i$  occurs exactly  $ans[i]$  once, then it is necessary to add the number to the number of occurrences of its Length property suffix, which coincides with its prefix; then have this suffix (of course, smaller than the  $i$  length) Run "bouncing" of the amount of your suffix, etc.

Now consider the second problem. Using the standard method: assign to a string  $s$  line  $t$  across the separator, ie the resulting string  $s + \# + t$ , and count it for a prefix function. The only difference from the first task will be that it is necessary to take into account only the values prefix functions that belong to the line  $t$ , i.e. all  $\pi[i]$  for  $i \geq n + 1$ .

## The number of different substrings in a string

A string  $s$  length  $n$ . Requires her to count the number of different substrings.

We will solve this problem iteratively. Namely, learn, knowing the current number of different substrings recalculate this number, when added to the end of one symbol.

So, let  $k$  - the current number of different substrings  $s$ , and we add to the end of the symbol  $c$ . Obviously, the result could appear some new substring ending in this new symbol  $c$ . Namely, are added as new ones substring ending on the symbol  $c$  and not seen before.

Take a string  $t = s + c$  and invert it (we write the characters in reverse order). Our task - to calculate how many lines  $t$  such prefixes that are not found it anywhere else. But if we think of a string  $t$  prefix-function and find its maximum value  $\pi_{\max}$ , then, obviously, in the line  $t$  occurs (not at the beginning), its prefix length  $\pi_{\max}$ , but not greater length. Clearly, prefixes shorter length certainly found it.

So, we have found that the number of new substrings that appear when you append the character  $c$  as well  $s.length() + 1 - \pi_{\max}$ .

Thus, for each character is appended for we  $O(n)$  can count the number of different substrings. Consequently, for  $O(n^2)$  we can find a number of different substrings for any given line.

It is worth noting that in exactly the same, you can count the number of different substrings and append a character to the beginning, as well as removing characters from the end or the beginning.

## Compression line

A string  $s$  length  $n$ . You want to find the shortest it "compressed" representation, ie, find a line  $t$  of shortest length that  $s$  can be represented as a concatenation of one or more copies  $t$ .

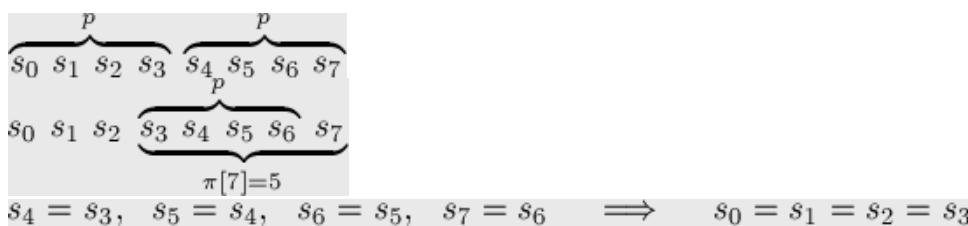
Clearly, the problem is to find the length of the search string  $t$ . Knowing the length of the answer to the problem would be, for example, the prefix of  $s$  this length.

Count on the line  $s$  prefix function. Consider it the last value, ie  $\pi[n - 1]$ , and introduce notation  $k = n - \pi[n - 1]$ . We show that if  $n$  divisible by  $k$ , then it  $k$  will be a long answer, otherwise effective compression does not exist, and the answer is  $n$ .

Indeed, suppose that  $n$  is divided into  $k$ . Then the string can be represented as a series of blocks of length  $k$ , and, by the definition of the prefix function, the prefix length  $n - k$  is the same as its suffix. But if the last block must coincide with the penultimate penultimate - with predpredposlednim, etc. In the end, it turns out that all the blocks are the same blocks, and is  $k$  really fit the answer.

We show that this response is optimal. In fact, otherwise, if there was minimal  $k$ , and that the prefix to the end of the function would be greater than  $n - k$ , i.e. a contradiction.

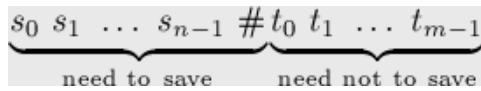
Suppose now that  $n$  is not divisible by  $k$ . We show that this implies that the length of the response is  $n$ . We prove by contradiction - assume that the answer exists, and has a length  $p$  ( $p$  divider  $n$ ). Note that the prefix function must be larger  $n - p$ , i.e. this suffix is to partially cover the first block. Now consider the second block row; because prefix matches with a suffix and a prefix, suffix and the cover unit, and their displacement relative to each other  $k$  does not divide the block length  $p$  (otherwise be  $k$  shared a  $n$ ), all the symbols are the same unit. But then the line consists of the same character, here  $k = 1$ , and the answer must exist, ie, so we come to a contradiction.



## The construction of an automaton from a prefix function

Let's go back to the reception have been used repeatedly concatenating two strings through the

separator, ie data lines  $s$  and  $t$  prefix computation functions for string  $s + \# + t$ . Obviously, since character  $\#$  is a delimiter, then the value of the prefix function will never exceed  $s.length()$ . It follows that, as mentioned in the description of the algorithm for calculating the prefix function is sufficient to store only the string  $s + \#$  and prefix value function for it, and for all subsequent characters prefix function to calculate on the fly:



Indeed, in such a situation, knowing the current symbol  $c \in t$  and the value in the prefix function previous position can calculate a new value prefix function, does not at all using the previous string code  $t$  prefix and values in their function.

In other words, we can build a **machine**: it is a state of the current value of the prefix functions, transitions from one state to another will be implemented under the symbol:



Thus, even with no more rows  $t$ , we can construct a pre-transition table  $(old\_pi, c) \rightarrow new\_pi$  with the same prefix computation algorithm functions:

```
string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

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) {
        int j = 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  $O(n^2k)$  ( $k$ - cardinality of the alphabet). But note that, instead of the inner loop `while`, which gradually shortens response, we can use the already computed the table: moving from value  $j$  to value  $\pi[j - 1]$ , we actually say that the transition from the state  $(j, c)$  will result in the same state as the transition  $(\pi[j - 1], c)$ , and to have an answer accurately counted (as  $\pi[j - 1] < j$ ):

```
string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

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 construction machine working for  $O(nk)$ .

Can be useful when such a machine? To begin with let us recall that we consider prefix function for a string  $s + \# + t$ , and its value is usually used for the sole purpose: to find all occurrences of  $s$  in string  $t$ .

Therefore, the most obvious benefits of constructing such a machine - **acceleration calculation function prefix** string  $s + \# + t$ . Is built on the line  $s + \#$  machine, we no longer need any string  $s$  or 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 another, less obvious use. This is the case when the string is a **giant string constructed by any rule**. This may be, for example, Gray string or a string formed of a recursive combination of several short lines fed to the input  $t$ .

Suppose for definiteness we solve the **following problem**: given a number of  $k \leq 10^5$  lines of Gray, and given string  $s$  length  $n \leq 10^5$ . Required to count the number of occurrences of a string  $s$  in  $k$ th row Gray. Recall Gray lines are defined as follows:

```

 $g_1 = "a"$ 
 $g_2 = "aba"$ 
 $g_3 = "abacaba"$ 
 $g_4 = "abacabadabacaba"$ 
...

```

In such cases, even just building line  $t$  will be impossible because of its astronomical length (eg,  $k$  Gray th row has length  $2^k - 1$ ). Nevertheless, we can calculate the value of the prefix function at the end of the line, knowing the value of the prefix function, which was before the line.

So, in addition to the machine also calculate such quantities:  $G[i][j]$ - the value of the machine, achieved after "feeding" him line  $g_i$ , if before the machine was in a state  $j$ . The second value -  $K[i][j]$ - the number of occurrences of a string  $s$  in a string  $g_i$  if to "feeding" of the line  $g_i$  machine is in state  $j$ . In fact,  $K[i][j]$ - the number of times that the automatic take a value  $s.length()$  in a time "feeding" line  $g_i$ . It is clear that the answer to the problem would be the value  $K[k][0]$ .

How to calculate these values? First, the basic values are  $G[0][j] = j$ ,  $K[0][j] = 0$ . And all subsequent values can be calculated by using the previous value and the machine. So, to calculate these values for some  $i$  we remember that the line  $g_i$  consists of  $g_{i-1}$  plus  $i$ th character of the alphabet plus again  $g_{i-1}$ . Then, after "feeding" the first piece ( $g_{i-1}$ ) will go into a state machine  $G[i-1][j]$ , then after "feeding" the character  $\text{char}_i$  he will go to state:

$\text{mid} = \text{aut}[ G[i-1][j] ][\text{char}_i]$

After this machine "fed" the last piece, ie  $g_{i-1}$ :

$G[i][j] = G[i-1][\text{mid}]$

Number of  $K[i][j]$  easily considered as the sum of the amounts in three pieces  $g_i$ : a string  $g_{i-1}$ , a symbol  $\text{char}_i$ , and then a string  $g_{i-1}$ :

$K[i][j] = K[i-1][j] + (\text{mid} == s.length()) + K[i-1][\text{mid}]$

So, we have solved the problem for strings Gray, similarly you can solve a class of problems. For example, exactly the same method solves the **following problem**: given a string  $s$ , and samples  $t_i$ , each of which is defined as follows: a string of ordinary characters, among which may occur recursive insert other lines in the form  $t_k[\text{cnt}]$ , which means that in this place is to be inserted  $\text{cnt}$ copies the string  $t_k$ . An example of such a scheme:

```

 $t_1 = "abdeca"$ 
 $t_2 = "abc" + t_1[30] + "abd"$ 
 $t_3 = t_2[50] + t_1[100]$ 
 $t_4 = t_2[10] + t_3[100]$ 

```

It is guaranteed that this description does not contain circular dependencies. Constraints are such that if explicitly disclose recursion and find the line  $t_i$ , their length can reach about  $100^{100}$ .

Required to find the number of occurrences of a string  $s$  in each of the rows  $t_i$ .

The problem is solved in the same building machine prefix function, then it is necessary to calculate and add it transitions to a whole row  $t_i$ . In general, it's just a more general case in comparison to the problem of lines Gray.

## Tasks in the online judges

List of tasks that can be solved by using a 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]

Поделиться >

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:39  
EDIT: 4 Apr 2012 20:11

## Hashing algorithms in problems on the line

Hashing algorithms lines help solve a lot of problems. But they have a big flaw that often they are not 100% - ny, because there are many rows that have the same hash. Another thing is that in the majority of tasks it can be ignored, since the probability of coincidence hashes still very small.

### Contents [hide]

- Hashing algorithms in problems on the line
  - Determination of the hash and its calculation
  - An example of the problem. Search for duplicate rows
  - Hash string and its fast computation
  - The use of hashing
  - Determination of the number of different substrings

### Determination of the hash and its calculation

One of the best ways to determine the hash function from a string S is as follows:

$$h(S) = S[0] + S[1] * P + S[2] * P^2 + S[3] * P^3 + \dots + S[N] * P^N$$

where P - a number.

It is reasonable to choose a prime number P, approximately equal to the number of characters in the input alphabet. For example, if only line predpolayutsya composed of small letters, then a good choice P = 31. If the letter may be capitalized and small, then, for example, P = 53 possible.

All pieces of code in this article will be used P = 31.

The very desirable to store the hash value in the largest numeric type - int64, he's long long. It is obvious that the length of the string of about 20 characters will already be overrun value. The key point - that we do not pay attention to these overflow, as if taking a hash modulo  $2^{64}$ .

An example of calculating the hash, if permitted 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 'aaaaa' hash was non-zero
    hash += (s[i] - 'a' + 1) * p_pow;
    p_pow *= p;
}
```

In most applications it makes sense to first calculate all the necessary degree of P in any array.

### An example of the problem. Search for duplicate rows

Already now we are able to effectively solve this problem. Given a list of strings S [1..N], each no longer than M symbols. Suppose you want to find all duplicate lines and divide them into groups that each group had only the same line.

Normal sorting lines we would have an algorithm with complexity  $O(NM \log N)$ , while using hashes, we get  $O(NM + N \log N)$ .

Algorithm. Calculate the hash of each line, and sort the words in this hash.

```

vector <string> s (n);
// ... Reading the lines ...

// Assume all powers of p, say, up to 10,000 - the maximum length of lines
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;

// Calculate the hashes of all rows
// Array store the hash value and the line number 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 ());

// Display the answer
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 string and its fast computation

Suppose we are given a string S, and are given indices I and J. required to find a hash of substring S [I..J].

By definition, we have:

$$H [I..J] = S [I] + S [I + 1] * P + S [I + 2] * P ^ 2 + \dots + S [J] * P ^ (JI)$$

from 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 string S, we can in O (1) to receive a hash of any substring**.

Only a problem - is that we must be able to divide by P [I]. In fact, it's not so simple. Since we compute hash modulo  $2 ^ {64}$ , then dividing by P [I], we have to find the inverse element thereto in the field (e.g., via [the Extended Euclidean algorithm](#) ), and perform the multiplication by the inverse.

However, there is an easier way. In most cases, instead of hashing to divide degree P, it is possible, on the contrary, these multiply their extent .

Assume we are given two hash one multiplied by P [I], and the other - for P [J]. If I < J, then we multiply hash pery on P [JI], or else we multiply the second hash to P [IJ]. Now, we have led the hashes to the same extent and can easily compare them.

For example, the code that calculates hashes of all prefixes, and then O (1) compares two strings:

```

string s; int i1, i2, len; // Input

// Assume all powers of 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;

// Calculate the hashes of 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];
}

// Get the hashes of two substrings
long long h1 = h [i1 + len-1];
if (i1) h1 -= h [i1-1];
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";

```

## The use of hashing

Here are some typical applications hash:

- Rabin-Karp algorithm search substring of O (N)
- Determination of the number of different substrings of O (N ^ 2 log N) (See below.)
- Determination of the number of palindromes in a string

## Determination of the number of different substrings

Given a string S of length N, consisting only of lowercase Latin letters. Required to find the number of different substrings in this line.

To solve the brute over by one length of the substring: L = 1 .. N.

For each L we construct an array of hashes substrings of length L, moreover we give hashes to the same extent, and we can sort this array. The number of different elements in the array we add to the answer.

Implementation:

```

string s; // Input string
int n = (int) s.length ();

// Assume all powers of 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;

// Calculate the hashes of 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 over the length of the substring
for (int l = 1; l <= n; ++ l)
{
    // Looking for the answer to the current length

    // Get the 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-l-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 ()), hs.end ());
    result += (int) hs.size ();
}

cout << result;

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:41  
EDIT: 11 Jun 2008 10:42

## Rabin-Karp algorithm search substring of O (N)

### Contents [hide]

- Rabin-Karp algorithm search substring of O (N)
  - Implementation

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 Carp (Karp), 1987.

Given a string S and the text T, consisting of small Latin letters. You want to find all occurrences of the string S in the text T for time  $O(|S| + |T|)$ .

Algorithm. Calculate the hash of a string S. Calculate the hash value for all prefix of T. Now brute over all length of the substring  $T[|S|]$  and every comparable with  $|S|$  in  $O(1)$ .

### Implementation

```
string s, t; // Input

// Assume all powers of 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;

// Calculate the hashes of all prefixes of strings 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];
}

// Calculate the hash of a string 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 over all length of the substring 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 << ' ';
    }
```

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 10:42  
EDIT: 10 Sep 2012 14:0

## Expression parsing. Reverse Polish Notation

A string that is a mathematical expression containing numbers, variables, various operations. You want to calculate the value of  $O(n)$  which  $n$ - the length of the string.

Here we describe an algorithm that translates this into a so-called **Reverse Polish Notation** (explicitly or implicitly), and already it evaluates the expression.

### Contents [hide]

- Expression parsing. Reverse Polish Notation
  - Reverse Polish Notation
  - Analysis of a simple expression
  - Unary operations
  - Right-associative

### Reverse Polish Notation

Reverse Polish Notation - a form of writing mathematical expressions in which operators are placed after its 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 Polish mathematician Jan Lukasiewicz.

Ease reverse polish notation is that the expressions presented in a form that is very **easy to calculate**, with linear time. Zaveděm stack is initially empty. Will move from left to right on the reverse polish notation; if the current element - a number or a variable, then we place on top of the stack, its value; if the current element - an operation that We reach of the top two elements of the stack (or one if the operation is a unary), apply operations to them, and we place the result back on the stack. In the end, the stack will be exactly one element - the value of the expression.

Clearly, this simple algorithm is executed for  $O(n)$ , that the order of the expression.

### Analysis of a simple expression

While we consider only the simplest case: all operations **are binary** (ie, two arguments), and all **left associative** (ie, with equal priority are executed from left to right). Parentheses are allowed.

Zaveděm two stacks: one for numbers and one for operations and parentheses (ie, stack characters). Initially, both the stack is empty. For the second stack will support the precondition that all operations are ordered in descending it on a strict priority, if we move from the top of the stack. If the stack has the parentheses, then each ordered unit operations, between parentheses, and the entire stack in this case is not necessarily ordered.

Let's go on a line from left to right. If the current element - a figure or a variable, then put on the stack the value of this number / variable. If the current element - opening parenthesis, we put it on the stack. If the current element - a closing parenthesis, we will push the stack and perform all operations until such time as we do not izvlechěm opening bracket (ie, in other words, meeting the closing parenthesis, we perform all operations inside this bracket). Finally, if the current element - an operation that, until the top of the stack is the operation with the same or higher priority will be to push and execute it.

After we process the entire string, the stack operations can still remain some of the operations that have not yet been calculated, and the need to do all of them (ie, act similarly to the case when we meet 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;
    }
}
```

```

        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 (i < s.length() && isalnum (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 of  $O(n)$ , and at the same time we have implicitly used the Reverse Polish Notation: We are located operations in that order, when the time of calculation of the next operation both its operands are already calculated. Slightly modifying the above algorithm, we 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, a 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 always stands or other operation, or an opening bracket, or nothing at all (if it stands at the beginning of the line). Before binary operation, in contrast, is always either operand (number / variable) or a closing bracket. 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 unary and binary operations when removed from the stack and calculating. You can, for example, for unary operations instead of the symbol  $s[i]$  to put on the stack  $-s[i]$ .

Priority for unary operations should be chosen such that it was more priorities of all binary operations.

In addition, it should be noted that the unary operations actually associates to the right - if a row are a few unary operators, they must be processed from right to left (for a description of this case, see. Below; The above code already takes into account the right-associative).

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 4; // op == '-' || op == '-'
    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 = -curop;
                while (!op.empty() && (
                    curop >= 0 && priority(op.back()) >= priority(curop)
                    || curop < 0 && priority(op.back()) > priority(curop)
                ))
                    process_op (st, op.back()), op.pop_back();
                op.push_back (curop);
                may_unary = true;
            }
            else {
                string operand;
                while (i < s.length() && isalnum (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));
                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 permitted only  $+$ and  $-$ , right associative plays no role, so in such cases no complications in the scheme can not enter. Ie cycle:

```

while (!op.empty() && (
    curop >= 0 && priority(op.back()) >= priority(curop)
    || curop < 0 && priority(op.back()) > priority(curop)
))
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-associative

Right-associative operator means that with equal priority operators 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 operation is considered right-associative (really,  $a^b^c$  is generally perceived as  $a^{(b^c)}$ , rather than  $(a^b)^c$ ).

What are the differences you need to make in the algorithm to correctly handle right-associative? In fact, the most minimal changes are needed. The only difference will be shown only at equal priorities, and it is that the operation of equal priority, located on the 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)))
    ...
}
```

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 22 Jun 2008 13:37  
edited 1 Oct 2012 12:46

## Suffix array

A string  $s[0 \dots n - 1]$  length  $n$ .

of th **suffix** string is called a substring  $s[i \dots n - 1]$ ,  $i = 0 \dots n - 1$ .

Then the **suffix array** of strings scalled the permutation of the indices of suffixes  $p[0 \dots n - 1]$ ,  $p[i] \in [0; n - 1]$  which specifies the order of suffixes in lexicographic sorting order. In other words, you want to sort all suffixes specified string.

For example, for a string  $s = abaab$  suffix array will be equal to:

(2, 3, 0, 4, 1)

### Contents [hide]

- Suffix array
  - Building for  $O(n \log n)$
  - Applications
    - Finding the smallest cyclic shift line
    - Search substring
    - A comparison of two substrings
    - The greatest common prefix of two strings: a method with additional memory
    - The greatest common prefix of two strings: a way without additional memory. The greatest common prefix of two adjacent suffixes
    - The number of different substrings
  - Tasks in the online judges

## Building for $O(n \log n)$

Strictly speaking, the algorithm described below will not perform sorting suffixes, and **cyclic shifts** of the line. However, this algorithm is easily obtained and an algorithm for sorting suffixes concatenate the end of an arbitrary character string, which is certainly less than any symbol which may consist of a string (for example, it may be a buck or Sharp, in the C language may be used for these purposes are already available null character).

Immediately, we note that since we sort of cyclic shifts, then the substring we consider **cyclical** : a substring  $s[i \dots j]$ , when  $i > j$  understood substring  $s[i \dots n - 1] + s[0 \dots j]$ . In addition, all pre-indices are taken modulo the length of the string (in order to simplify the formulas, I will omit the subscript explicit modulo).

The issue before us algorithm consists of approximately  $\log n$  phases. On  $k$ th phase ( $k = 0 \dots \lceil \log n \rceil$ ) sorted cyclic length of the substring  $2^k$ . At last,  $\lceil \log n \rceil$ th phase will be sorted substring length  $2^{\lceil \log n \rceil} > n$ , which is equivalent to sorting cyclic shifts.

At each phase of the algorithm in addition to the rearrangement  $p[0 \dots n - 1]$  of cyclic indexes substrings will be maintained for each cyclic substring starting at the position  $i$  with the length  $2^k$ , **number of  $c[i]$  equivalence classes**, which belongs to this substring. In fact, among the substrings may be the same, and the algorithm will need information about this. Furthermore, non  $c[i]$  equivalence classes will give so that they retain the order information and, if one is less than another suffix, the class number and he must get smaller. Classes will be numbered for convenience from scratch. Number of equivalence classes will be stored in a variable **classes**.

Let us give **an example**. Consider the string  $s = aaba$ . Values array  $p[]$  and  $c[]$  for each stage with zero for the second as follows:

0 :	$p = (0, 1, 3, 2)$	$c = (0, 0, 1, 0)$
1 :	$p = (0, 3, 1, 2)$	$c = (0, 1, 2, 0)$
2 :	$p = (3, 0, 1, 2)$	$c = (1, 2, 3, 0)$

It is worth noting that in the array  $p[]$  may ambiguity. For example, a zero phase array can be:  $p = (3, 1, 0, 2)$ . Then what option will depend on the particular implementation of the algorithm, but all options are equally valid. At the same time, the array  $c[]$  is no ambiguity could not be.

Let us now turn to the construction of **the algorithm**. Input data:

```
char *s; // входная строка
int n; // длина строки

// константы
const int maxlen = ...; // максимальная длина строки
const int alphabet = 256; // размер алфавита, <= maxlen
```

At **zero phase** we must sort cyclic substring length 1, ie, individual characters in a string, and divide them into equivalence classes (just the same characters should be assigned to the same equivalence class). This can be done trivially, for example, Counting sort. For each character count how many times he met. Later on this information rebuild the array  $p[]$ . After that, the passage through the array  $p[]$  and comparing characters built array  $c[]$ .

```
int p[maxlen], cnt[maxlen], c[maxlen];
```

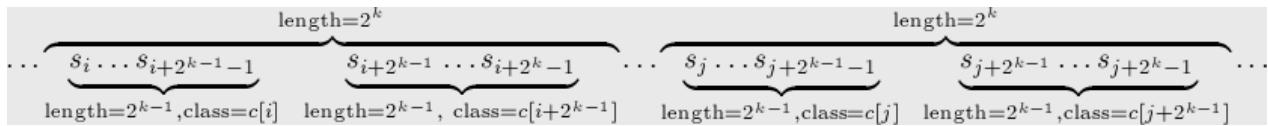
```

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 that we have fulfilled  $k - 1$ th phase (ie the calculated values of arrays  $p$  and  $c$  for it), and now have learned over  $O(n)$  perform the following,  $k$ th, phase. Since the phases of all  $O(\log n)$ , it will give us the desired algorithm with time  $O(n \log n)$ .

To do this, we note that the length of the cyclic substring  $2^k$  consists of two substrings of length  $2^{k-1}$ , which we can compare with each other for  $O(1)$  using the information from the previous phase - number  $c$  of equivalence classes. Thus, for the length of the substring  $2^k$  starting at the position  $i$ , all the necessary information is contained in a pair of numbers  $(c[i], c[i + 2^{k-1}])$  (again, we use an array  $c$  with the previous phases).



This gives us a very simple solution: sort substring length  $2^k$  simply by these pairs of numbers, it will give us the correct order, ie, array  $p$ . However, the conventional sorting, runs in time  $O(n \log n)$ , we are not satisfied - it will give an algorithm for constructing suffix array with the time  $O(n \log^2 n)$  (but this algorithm is somewhat easier to write than described below).

How to quickly perform a sorting pairs? Since the elements of couples do not exceed  $n$ , then you can sort counting. However, to achieve the best hidden in the asymptotic constants instead of sorting couples arrive at a sort of prime numbers.

We use here reception, which founded the so-called **digital sorting**: to sort couples sort them first to the second item, and then - on the first element (but certainly stable sort, that does not disrupt the relative order of elements in the equation). However, a separate second elements are already in order - this order is set in the array  $p$  from the previous phase. Then, to arrange for a second pair of elements, you just have to each array element  $p$  to take away  $2^{k-1}$  - it will give us the sort order for the second pairs of elements (in fact  $p$  gives ordering substring length  $2^{k-1}$ , and the transition to the line twice the length of the substring these become their soul mate, so from position of the second half the length of the first half is subtracted).

Thus, with just a subtraction of elements in the array  $p$ , we produce sorting by second pairs of elements. Now we have to produce a stable sort on the first element pairs it already can be done in  $O(n)$  using the Counting sort.

It remains only to count the number  $c$  of equivalence classes, but they were easy to get, just go to get a new permutation  $p$  and comparing adjacent elements (again, comparing two numbers as a pair).

We present the **implementation** of all phases of the implementation of the algorithm, in addition to zero. Introduce 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));
}

```

This algorithm requires  $O(n \log n)$  time and  $O(n)$  memory. However, given the size of even  $k$  the alphabet, while the work becomes  $O((n+k) \log n)$ , and the size of 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), and therefore  $p[0]$  yields the desired position of the smallest cyclic shift. Hours -  $O(n \log n)$ .

### Search substring

Let it be required in the text  $t$  string to search  $s$  online (ie, pre-line  $s$  should be considered unknown). We construct the suffix array for text  $t$  over  $O(|t| \log |t|)$ . Now substring  $s$  will look as follows: we note that the required entry must be prefixed with any suffix  $t$ . Since suffixes we ordered (it gives us suffix array), then the substring  $s$  you can search binary search suffix string. A 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 search substring in the text becomes  $O(|p| \log |t|)$ .

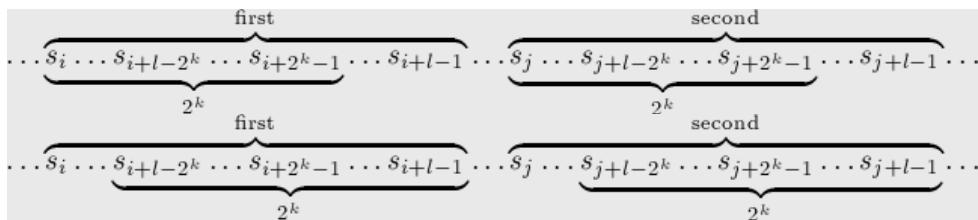
### A comparison of two substrings

Required for a given string  $s$ , producing some of its preprocessing learn for  $O(1)$  responding compare two arbitrary substrings (ie checking that the first substring equal / less than / greater than the second).

We construct the suffix array for  $O(|s| \log |s|)$  while keeping the intermediate results: we need arrays  $c[]$  from each phase. Therefore, the memory is also required  $O(|s| \log |s|)$ .

Using this information, we can for the  $O(1)$  compare any two substrings of length equal to a power of two: it is enough to compare the number of equivalence classes of the corresponding phase. Now we have to generalize this method to a substring of arbitrary length.

Suppose now received another request for comparison of two substrings of length  $l$  with origins in the indexes  $i$  and  $j$ . We find the greatest length of the block, placed inside a substring of this length, ie, most  $k$  such that  $2^k \leq l$ . Then the comparison of two substrings can be replaced by comparing two pairs of overlapping blocks of length  $2^k$ : first it is necessary to compare the two sets of starting positions  $i$  and  $j$ , as with equal - compare two blocks ending in positions  $i + l - 1$  and  $j + l - 1$ :



Thus, the implementation is roughly this (it is assumed here that the calling procedure itself calculates  $k$ , as do it in constant time is not so easy (apparently, most likely - predposchëtom), but in any case this is not related 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)]);
    pair<int,int> b = make_pair (c[k][j], c[k][j+l-(1<<k)]);
    return a == b ? 0 : a < b ? -1 : 1;
}

```

### The greatest common prefix of two strings: a method with additional memory

Required for a given string  $s$ , producing some of its preprocessing learn for  $O(\log |s|)$  responding greatest common prefix (longest common prefix, lcp) for two arbitrary suffix with the positions  $i$  and  $j$ .

The method described here requires  $O(|s| \log |s|)$  additional memory; another method using linear memory space, but non-

constant response time is described in the next section.

We construct the suffix array for  $O(|s| \log |s|)$  while keeping the intermediate results: we need arrays  $c[]$  from each phase. Therefore, the memory is also required  $O(|s| \log |s|)$ .

Suppose now received another request: a pair of indices  $i$  and  $j$ . We use the fact that we can for the  $O(1)$  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 level to check if a substring of this length are the same, then the answer has to add this power of two, and the greatest common prefix will continue to look to the right of equal 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, through the  $\log_n$  designated constant equal to the logarithm  $n$  to the base 2, rounded down.

## The greatest common prefix of two strings: a way without additional memory. The greatest common prefix of two adjacent suffixes

Required for a given string  $s$ , producing some of its preprocessing learn responding greatest common prefix (longest common prefix, lcp) for two arbitrary suffix with the positions  $i$  and  $j$ .

Unlike the previous method described herein will perform preprocessing line for  $O(n \log n)$  time  $O(n)$  memory. The result will be an array of preprocessing (which in itself is an important source of information on line, and therefore be used for other tasks). The answer to the request will be made as a result of the query RMQ (at least on the interval, range minimum query) in the array, so the different implementations can be obtained as logarithmic and constant time operation.

The base for this algorithm is the following idea: find some way most common prefixes for each **adjacent pair in the sort order of suffixes**. In other words, we construct an array  $lcp[0 \dots n - 2]$ , where  $lcp[i]$  is the greatest common prefix suffix  $p[i]$  and  $p[i + 1]$ . This array will give us the answer to any two adjacent suffixes line. Then the answer for any two suffixes, not necessarily adjacent, you can get through this array. In fact, let received a request from a certain number of suffixes  $i$  and  $j$ . Let us find these codes in suffix array, ie, let  $k_1$  and  $k_2$  - their position in the array  $p[]$  (order them, ie let  $k_1 < k_2$ ). Then the answer to this query will be at least in the array  $lcp$ , taken in the interval  $[k_1; k_2 - 1]$ . In fact, the transition from a suffix  $i$  to the suffix  $j$  can replace an entire chain of transitions, starting with a suffix  $i$  and ending in the suffix  $j$ , but includes all intermediate suffixes that are in the sort order between them.

Thus, if we have an array  $lcp$ , then the response to any request for the longest common prefix is the request **for the minimum interval** of the array  $lcp$ . This classic problem of the minimum on the interval (range minimum query, RMQ) has a set of solutions with different asymptotics described [here](#).

So, our main task - **to build** the array  $lcp$ . Build it, we will in the course of the algorithm for constructing suffix array: each current iteration will build an array of  $lcp$  substrings for cyclic current length.

After zero iteration array  $lcp$  must obviously be zero.

Suppose now that we have completed  $k - 1$ th iteration, received from her array  $lcp'$ , and should at the current  $k$ th iteration recalculate the array, it received a new value  $lcp$ . As we remember, in the algorithm for constructing suffix array cyclic substring length  $2^k$  broke in half into two substrings of length  $2^{k-1}$ ; use this same technique for the construction of the array  $lcp$ .

Thus, even for the current iteration algorithm for computing the suffix array done his job, found a new meaning permutation  $p[]$  substrings. We will now go through this array and watch a pair of adjacent substrings:  $p[i]$  and  $p[i + 1]$ ,  $i = 0 \dots n - 2$ . Breaking each substring in half, we have two different situations: 1) the first halves of the substring in the positions  $p[i]$  and  $p[i + 1]$  different, and 2) the first halves of the same (recall such a comparison can be easily produced by simply comparing the numbers of the classes  $c[]$  from the previous iteration). Let us consider each of these cases separately.

- 1) The first half of substrings different. Note that while in the previous step, these first halves must have been nearby. In fact, the equivalence classes could not disappear (and may only appear), so all the different length of the substring  $2^{k-1}$  will (as the first halves) in the current iteration of the different length of the substring  $2^k$ , and in the same order. Thus, to determine  $lcp[i]$  in this case it is necessary only to take the corresponding value of the array  $lcp'$ .
- 2) The first halves of the same. Then the second half could be the same as or different; 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  $lcp[i]$ . To define it, we must do the same, as we are going to then calculate the greatest common prefix for any two suffixes: it is necessary to query the minimum (RMQ) on the corresponding interval of the array  $lcp'$ .

We estimate **the asymptotic behavior** of the algorithm. As we have seen in the analysis of these two cases, only the second case gives an increase in the number of equivalence classes. In other words, we can say that each new equivalence class appears with a single request RMQ. Because of all equivalence classes can be up to  $n$ , then we should at least look at the asymptotic behavior  $O(\log n)$ . And for this we need to use already some data structure to a minimum on the interval; this data structure will have to be rebuilt at each iteration (which only  $O(\log n)$ ). A good option is a data structure **segment tree**: it can be built in  $O(n)$ , and then perform searches for  $O(\log n)$  that just gives us the asymptotic behavior of the final  $O(n \log n)$ .

### 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;

    ... все действия по построению суфф. массива, кроме последней строки (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])
        lcnp[i] = lcp[rpos[c[a]]];
    else {
        int aa = (a + (1<<h)) % n, bb = (b + (1<<h)) % n;
        lcnp[i] = (1<<h) + rmq(lpos[c[aa]], rpos[c[bb]]-1);
        lcnp[i] = min(n, lcnp[i]);
    }
}
memcpy(lcp, lcnp, (n-1) * sizeof(int));
memcpy(c, cn, n * sizeof(int));
}

```

Here, in addition to the array `lcp` introduced a temporary array `lcnp` with its new value. It also supports an array `pos` that stores for each substring of 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, the size of it is passed as the second argument. The function `rmq` returns the minimum of the interval: the first argument by the second, inclusive.

Most of the algorithm for constructing suffix array had only to make up the array `c`, because during the calculation `lcp` we need the old values of the array.

It is worth noting that our implementation is the length of the common prefix for **cyclic substring**, while in practice more often desired length of the common prefix for the suffixes in their usual 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 the longest common prefix can now be found at least in the respective segment of the array `lcp`:

```

for (int i=0; i<n; ++i)
    pos[p[i]] = i;
rmq_build(lcp, n-1);

... поступил запрос (i,j) на нахождение LCP ...
int result = rmq(min(i,j), max(i,j)-1);

```

### The number of different substrings

Perform **preprocessing** described in the previous section: for the  $O(n \log n)$  time and  $O(n)$  memory, we each pair of adjacent in the sort order suffixes find the length of the longest common prefix. We now find this information on the number of different substrings in a string.

For this we consider substrings which start at the new position `p[0]`, then the position `p[1]`, etc. In fact, we take another suffix in the sort order and look what it prefixes give new substring. Thus, we obviously do not lose sight of any of the substrings.

Using the fact that the suffixes we have already sorted, it is easy to understand that the current suffix  $p[i]$  will as new substrings all its prefixes other than coinciding with the prefix suffix  $p[i - 1]$ . i.e all of its prefixes, except  $\text{lcp}[i - 1]$  the first, will give new substring. Since the length of the current suffix is  $n - p[i]$ , then we conclude that the current suffix  $p[i]$  gives  $n - p[i] - \text{lcp}[i - 1]$  new substring. Summing it all suffixes (for the very first,  $p[0]$  takes away nothing - just be added  $n - p[0]$ ), we get **the answer** to the problem:

$$\sum_{i=0}^n (n - p[i]) - \sum_{i=0}^{n-1} \text{lcp}[i]$$

## Tasks in the online judges

Tasks that can be solved using the suffix array:

- [UVA # 10679 "I Love Strings !!!"](#) [Difficulty: Medium]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 10 Aug 2008 19:50  
EDIT: 2 May 2012 22:52

## Suffix Machine

**Suffix machine** (or a directed acyclic graph of words) - is a powerful data structure that allows you 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.

Intuitively, suffix automaton can be understood as concise information about **all substrings** of the string. An impressive fact is that the suffix automaton contains all the information in a compressed form so that the line length  $n$ , it requires only a  $O(n)$  memory. Moreover, it can also be constructed in a time  $O(n)$  (if we consider the size of the alphabet  $k$  constant, otherwise - a time  $O(n \log k)$ ).

**Historically**, the first time the linear dimensions suffix automaton was opened in 1983 by Blumer et al., and in 1985 - 1986. were presented the first algorithm for its construction in linear time (Crochemore, Blumer et al.). For more details - see. List of references at the end.

English suffix automaton is called "suffix automaton" (in the plural - "suffix automata"), a directed acyclic graph of words - "directed acyclic word graph" (or simply "DAWG").

## Definition suffix automaton

**Definition.** **suffix automaton** for a given row  $s$  is called a minimal deterministic finite automaton that accepts all suffixes line  $s$ .

Decipher this definition.

- Suffix machine is a directed acyclic graph in which the vertices are **states** and the arcs of the graph - it **transitions** between these states.
- One of the states  $t_0$  is called **the initial state**, and it must be the source of the graph (ie reachable from all other states).
- Each **transition** in the machine - this arc labeled by some symbol. All transitions originating from any state, must have **different** labels. (On the other hand, the state transition can not be on any characters).
- One or more states are marked as **the terminal state**. If we will walk from the initial state  $t_0$  to any path to any terminal state, and write with the labels of all arcs passed, you get a string that is bound to be one of the suffix string  $s$ .
- Suffix machine contains the minimum number of vertices among all the machines that meet the conditions described above. (Minimum number of transitions is not required as subject to a minimum number of states in the automaton can not be "extra" ways - otherwise it would break the previous property.)

## The simplest properties of suffix automaton

The simplest, yet most important property suffix automaton is that it contains information about all the substrings  $s$ . Namely, **any path** from the initial state  $t_0$ , if we write the label arcs along this path, there is necessarily a **substring of** a string  $s$ . Conversely, any substring  $s$  matches a path starting in the initial state  $t_0$ .

In order to simplify the explanation, we shall say that the substring **matches** the path from the initial state, the labels along which form this substring. Conversely, we say that any path **corresponds to** that line, which is formed by its label arcs.

In each state suffix automaton leads one or more paths from the initial state. We say that a state **corresponds to** a

## Contents [hide]

- Suffix Machine
  - Definition suffix automaton
    - The simplest properties of suffix automaton
    - Examples of construction of suffix automata
  - An algorithm for constructing suffix automaton in linear time
    - Position endings *endpos*, their properties and connection with suffix automaton
    - Suffix links
    - Subtotal
    - An algorithm for constructing suffix automaton in linear time
    - The proof of the correctness of the algorithm
    - Proof linear number of operations
  - Implementation of the algorithm
  - Additional properties suffix automaton
    - The number of states
    - The number of transitions
    - Communication with the suffix tree. Construction of suffix tree on the suffix automaton and vice versa
  - Application in solving problems
    - Verification of entry
    - The number of different substrings
    - The total length of different substrings
    - Lexicographically  $k$ -th substring
    - The smallest cyclic shift
    - Number of occurrences
    - The position of the first occurrence
    - The positions of all occurrences
    - Search the shortest line, not included in this
    - Of the longest common substring of two strings
    - Longest common substring problem a few lines.
  - Tasks in the online judges
  - Literature

set of rows that meet all these paths.

### Examples of construction of suffix automata

Give examples suffix automata constructed for a few simple lines.

The initial state will be denoted here by  $t_0$ , and terminal states - mark with an asterisk.

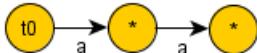
String  $s = " "$ :



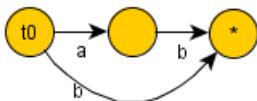
String  $s = "a"$ :



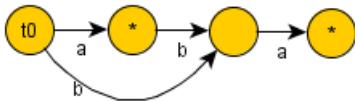
String  $s = "aa"$ :



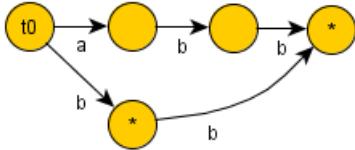
String  $s = "ab"$ :



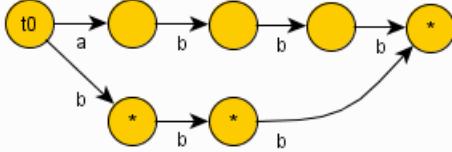
String  $s = "aba"$ :



String  $s = "abb"$ :



String  $s = "abbb"$ :



### An algorithm for constructing suffix automaton in linear time

Before you go directly to the description of the algorithm for constructing, it is necessary to introduce some new concepts and prove simple, but very important for understanding the suffix automaton of the lemma.

#### Position endings $\text{endpos}$ , their properties and connection with suffix automaton

Consider any nonempty substring of  $t$  a string  $s$ . Then we call **the set of endings**  $\text{endpos}(t)$  set of all positions in the line  $s$ , which end with occurrence of the string  $t$ .

We will call the two substrings  $t_1$  and  $t_2$  equivalent if they are the same set of endings. Thus, all non-empty substring of string can be divided into several **equivalence classes** according to their sets  $\text{endpos}(t_1) = \text{endpos}(t_2)$

It turns out that the suffix automaton ***endpos-equivalent substring match the same condition***. In other words, the number of states in the suffix automaton is equal to the number of classes of *endposequivalence* among all substrings, plus one initial state. Each of suffix automaton correspond to one or more substrings have the same value *endpos*.

**This statement we take for granted**, and we describe an algorithm for constructing suffix automaton, on that assumption - as we shall see later, all of the required properties of suffix automaton, except for minimal, will be fulfilled. (And the minimum follows from Theorem Nerode - see. Bibliography.)

We give some simple but important statements about values *endpos*.

**Lemma 1**. Two non-empty substring *u* and *w* ( $\text{length}(u) \leq \text{length}(w)$ ) are *endposequivalent* if and only if the string *u* in the string *s* only as a suffix string *w*.

The proof is almost obvious. In one direction, if *u* and *w* like numerals have terminations joining, it *u* is a suffix *w* and is present in *s* only a suffix *w*. Conversely, if *u* a suffix *w* and is only as this suffix, then their values *endpos* are equal by definition.

**Lemma 2**. Consider two nonempty substring *u*, and *w* ( $\text{length}(u) \leq \text{length}(w)$ ). Then a lot of them *endpos* either do not intersect or *endpos(w)* is contained in *endpos(u)*, with it depends on whether the *u* suffix *w* or not:

$$\begin{cases} \text{endpos}(w) \subset \text{endpos}(u) & \text{if } u \text{ — suffix } w, \\ \text{endpos}(u) \cap \text{endpos}(w) = \emptyset & \text{otherwise.} \end{cases}$$

Proof. Suppose that the sets *endpos(u)* and *endpos(w)* have at least one common element. Then this means that row *u* and *w* terminate at the same place, i.e. *u*- the suffix *w*. But then every occurrence of the string *w* contains on its end of occurrence of the string *u*, which means that a lot of it *endpos(w)* is entirely embedded in the set *endpos(u)*.

**Lemma 3**. Consider a class of *endposequivalence*. Substring sort all included in this class, in decreasing length. Then, the resulting sequence each substring will be shorter than the previous one, and thus is a suffix of the previous one. In other words, **the substring included in one equivalence class, in fact, are suffixes of each other, and make all sorts of different lengths in a certain interval**  $[x; y]$ .

Proof.

We fix a class *endpos*-equivalence. If it contains only one row, then the correctness of the lemma is obvious. Suppose now that the number of rows more than one.

According to Lemma 1, two different *endpos*-equivalent string is always such that one is a proper suffix of another. Therefore, in the same class *endpos*-equivalence can not be strings of the same length.

Denoted by *w* length, and through *u*- the shortest line in the equivalence class. According to Lemma 1, the string *u* is a proper suffix string *w*. Consider now any suffix string *w* with a length of the segment  $[\text{length}(u); \text{length}(w)]$ , and show that it is contained in the same equivalence class. In fact, the suffix can enter *s* only a suffix string *w* (as shorter suffix *u* included only as a suffix string *w*). Therefore, according to Lemma 1, this suffix *endpos*-equivalent line *w*, as required.

## Suffix links

Consider a state machine  $v \neq t_0$ . As we now know, the state *v* corresponds to a certain class of rows with the same values *endpos*, with if we denote the *w* longest of these lines, all the rest will suffixes *w*.

We also know that the first few lines of suffixes *w* (if we consider the suffixes in order of length) are in the same equivalence class, and all other suffixes (at least the empty suffix) - in some other classes. Denoted by *t* the first such suffix - it we spend suffix link.

In other words, **the suffix link**  $\text{link}(v)$  leads to a condition which corresponds to the **longest suffix** string *w*, which is in a different class of *endposequivalence*.

Here we assume that the initial state  $t_0$  corresponds to a single equivalence class (containing only the empty string), and believe  $\text{endpos}(t_0) = [-1 \dots \text{length}(s) - 1]$ .

**Lemma 4**. Suffix links form a **tree** whose root is the initial state  $t_0$ .

Proof. Consider an arbitrary state  $v \neq t_0$ . Suffix link  $\text{link}(v)$  from it leads to a condition which correspond strictly less than the length of the line (this follows from the definition of suffix links and Lemma 3). Therefore, moving suffix links, we will sooner or later will come from the state *v* to the initial state  $t_0$ , which corresponds to an empty string.

**Lemma 5**. If we construct from all available sets **tree** (on the "many-to-parent contains as a subset of all his children"), it will be the same as the structure of the tree of suffix links.*endpos*

Proof.

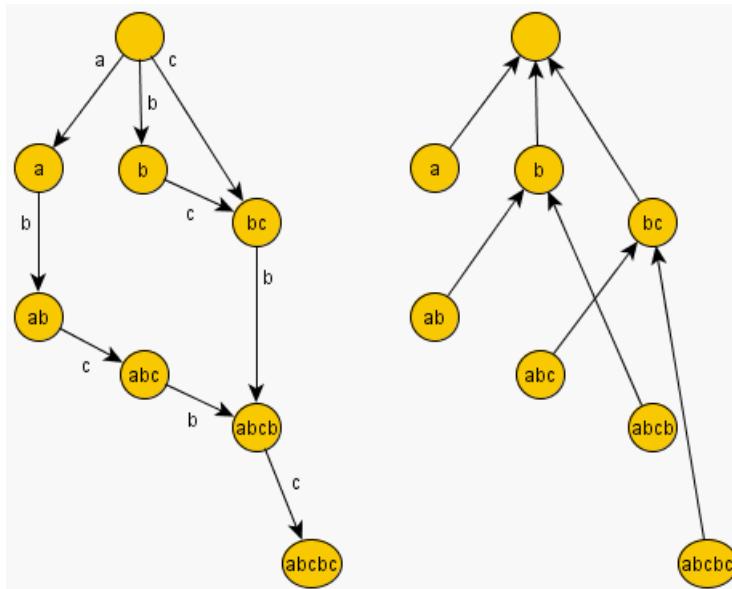
The fact that the sets  $\text{endpos}$  can construct a tree, it follows from Lemma 2 (that any two sets  $\text{endpos}$  are either disjoint, or one contained in the other).

We now consider an arbitrary state  $v \neq t_0$  and its suffix link  $\text{link}(v)$ . From the definition of suffix links and Lemma 2 follows:

$$\text{endpos}(v) \subset \text{endpos}(\text{link}(v)),$$

which, coupled with the previous lemma proves our assertion: word suffix links is essentially a timber is laid sets  $\text{endpos}$ .

Here is an example of a tree suffix suffix links in the machine, built for the line "abcabc":



## Subtotal

Before you begin to algorithm systematize the accumulated knowledge of the above, and introduce a pair of auxiliary notation.

- Set of substrings  $\text{sc}$  can be divided into equivalence classes according to their end sets  $\text{endpos}$ .
- Suffix machine consists of an initial state  $t_0$ , and one of each class of  $\text{endpos}$  equivalence.
- Each state  $v$  corresponds to one or more lines. Denoted by  $\text{longest}(v)$  the longest of these lines, through  $\text{len}(v)$  its length. Denoted by  $\text{shortest}(v)$  the shortest of these lines, and its length through  $\text{minlen}(v)$ .

Then all rows corresponding to this state are different suffixes line  $\text{longest}(v)$  and have all sorts of lengths in the interval  $[\text{minlen}(v); \text{len}(v)]$ .

- For each state  $v \neq t_0$  is defined suffix link leading to a state that corresponds to the suffix of the string  $\text{longest}(v)$  length  $\text{minlen}(v) - 1$ . Suffix links form a tree rooted at  $t_0$ , with this tree, in fact, is a tree inclusion relation between sets  $\text{endpos}$ .
- Thus,  $\text{minlen}(v)$  for the  $v \neq t_0$  suffix expressed by reference  $\text{link}(v)$  as:

$$\text{minlen}(v) = \text{len}(\text{link}(v)) + 1.$$

- If we start from an arbitrary state  $v_0$  and will go on suffix links, sooner or later you'll get to the initial state  $t_0$ . In this case, we obtain a sequence of disjoint segments  $[\text{minlen}(v_i); \text{len}(v_i)]$ , which the union will give one continuous segment.

## An algorithm for constructing suffix automaton in linear time

Proceed to a description of the algorithm. The algorithm will be **online**, ie will add one character string  $s$  rearranging accordingly the current machine.

To achieve linear memory consumption in each state, we will only store the value  $\text{len}, \text{link}$  and a list of transitions from this state. Tags terminal states, we will not support (we'll show you how to place these labels after construction

of suffix automaton if there is a need for them).

**Initially** the machine consists of a single state  $t_0$ , which we agree to assume zero state (the remaining states will receive numbers  $1, 2, \dots$ ). Assign this state  $\text{len} = 0$ , and the value  $\text{link}$  we assign to convenience  $-1$  (a reference to a fictitious, non-existent state).

Accordingly, the whole task now is to ensure that the process to implement **the addition of one character  $c$**  to the end of the current line. We describe this process:

- Let  $\text{last}$  - a state corresponding to the entire current line to add the character  $c$ . (Initially  $\text{last} = 0$ , and after the addition of each character we will change the value  $\text{last}$ .)
- Create a new state  $\text{cur}$  by putting him  $\text{len}(\text{cur}) = \text{len}(\text{last}) + 1$ . Value  $\text{link}(\text{cur})$  until consider uncertain.
- Let's make this cycle: initially we are able  $\text{last}$ ; if he does not go out to the letter  $c$ , then add the transition in the letter  $c$  to the state  $\text{cur}$ , and then go on suffix link, check again - if there is no transition, then add. If at some point it happens that such a transition is already there, then stop - and denote  $p$  number of the state in which it happened.
- If you have never happened that the shift in the letter  $c$  already had, and we did come down to a fictitious state  $-1$  (in which we have come to the suffix link from the initial state  $t_0$ ), we can simply assign  $\text{link}(\text{cur}) = 0$  and exit.
- Suppose now that we stopped at a certain state  $p$ , from which there was a shift in the letter  $c$ . We denote by  $q$  the state, which leads the existing transition.
- Now we have two cases depending on whether  $\text{len}(p) + 1 = \text{len}(q)$  or not.
  - If  $\text{len}(p) + 1 = \text{len}(q)$  we can simply assign  $\text{link}(\text{cur}) = q$  and exit.
  - Otherwise, all the more complicated. It is necessary to produce "**cloning**" state  $q$ : create a new state  $\text{clone}$ , copy the data from the top  $q$  (suffix link transitions), except for the value  $\text{len}$ : it is necessary to assign  $\text{len}(\text{clone}) = \text{len}(p) + 1$ .

After cloning, we spend suffix link of  $\text{cur}$  this state  $\text{clone}$ , and redirect the suffix link from  $q$  within  $\text{clone}$ .

Finally, the last thing we need to do - is to go from state  $p$  to suffix links, and for each of the next state to check if there was a passage in the letter  $c$  to the state  $q$ , then redirect it to the state  $\text{clone}$  (and if not, then stop).

- In any case, whatever the performance of this procedure has ended, we update the value at the end of  $\text{last}$  assigning it  $\text{cur}$ .

If we also need to know which are the top of the **terminal**, and what - no, we can find all terminal nodes after building a suffix automaton for the entire line. For this we consider the state corresponding to the entire string (which, obviously, we have stored in a variable  $\text{last}$ ), and we will walk in his suffix links, until we reach the initial state, and mark each traveled as a terminal condition. Easy to understand, thus we mark the states corresponding to all suffixes line  $s$  that we required.

In the next section we consider in detail 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 automaton. Thus, **the linearity of the number of states** is evident.

The linearity of the number of transitions, and generally linear time algorithm works less well understood, and they will be proved below, after the proof of the correctness of the algorithm.

## The proof of the correctness of the algorithm

- We call the transition **solid** if . Otherwise, i.e. when the transition will be called **discontinuous** . $(p, q)$   
 $\text{len}(p) + 1 = \text{len}(q)$   
 $\text{len}(p) + 1 < \text{len}(q)$

As can be seen from the description of the algorithm, continuous and non-continuous transitions lead to different branches of the algorithm. Solid transitions are so named because, appearing for the first time, they will never be changed. In contrast, non-continuous transitions can be changed by the addition of new characters to the string (can change the end of the arc transition).

- To avoid ambiguity, under the line  $s$  we mean the line 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  $\text{cur}$ , which will match the entire string  $s + c$ . It is clear why we have to create a new state - as together with the addition of a new character, a new equivalence class - a class of strings ending in add characters  $c$ .
- After creating a new state of the algorithm traverses the suffix links, starting with the state corresponding to the entire line  $s$ , and trying to add a transition on the symbol  $c$  in the state  $\text{cur}$ . Thus, we assign to each line

suffix *s* character *c*. But adding 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 immediately have to stop.

- The simplest case - if we did come down to a fictitious state — 1, adding anywhere along the transition to the new symbol *c*. This means that the character *c* in the string *s* has not previously met. We have successfully added all the transitions, it remains only to put down suffix link from the state *cur* - it obviously must be equal 0, as the state *cur* in this case correspond to all the suffixes of a string *s + c*.
- The second case - when we ran into an existing transition  $(p, q)$ . This means that we are trying to add to the machine line  $x + c$  (where  $x$ - some suffix line *s* having a length  $\text{len}(p)$ ), and this line **has been previously added** to the machine (ie, the string  $x + c$  is already included as a substring in the string *s*). Since we assume that the automaton for the string *s* is built correctly, the new transitions we should not add more.

However, there is a complexity to where lead suffix link from the state *cur*. We need to spend suffix link in a state in which the length of the string will be just this same  $x + c$ , that  $\text{len}$  for this state must be equal  $\text{len}(p) + 1$ . However, such a state could not exist: in this case, we need to make a "**splitting**" of the state.

- So, one of the possible scenarios, the transition  $(p, q)$  proved to be solid, that is  $\text{len}(q) = \text{len}(p) + 1$ . In this case, everything is simple, no splitting is not necessary to make, and we just spend suffix link from state *cur* to state *q*.
- Another, more complex version - when a discontinuous transition, that is  $\text{len}(q) > \text{len}(p) + 1$ . This means that the state *q* corresponds not only necessary to us substring  $w + c$  length  $\text{len}(p) + 1$ , but also the greater length of the substring. We have no choice but to make a "**split**" status *q*: split segment lines corresponding to her two subsegment, so the first will end up just long  $\text{len}(p) + 1$ .

How to make this split? We "**clone**" state *q*, making a copy of *clone* a parameter  $\text{len}(\text{clone}) = \text{len}(p) + 1$ . We copy in *clone* from *q* all the transitions, because we do not want in any way to change the path passed through *q*. Suffix link from *clone* we carry wherever led suffix link from the old *q*, and a link from *q* a direct *clone*.

After cloning, we spend suffix link from *cur* within *clone* - something for which we cloned.

Was the last step - to redirect some members of the *q* transitions, redirect them to *clone*. Which incoming transitions must redirect? Just redirect the only transitions corresponding to all suffixes line  $w + c$ , ie we need to continue to move along suffix links, starting from the top *p*, and as long as we do not reach the state of a fictitious — 1 or not we reach the state transition from which leads to a state other than *q*.

## Proof linear number of operations

First, once the reservation that we consider the alphabet size **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 B-tree, allows fast searches on the key and the key is added. Therefore, if we denote the *k* size of the alphabet, the asymptotic behavior of the algorithm will be  $O(n \log k)$  at  $O(n)$  the memory. However, if the alphabet is small enough, it is possible, sacrificing memory, avoid balanced lists, and keep the transitions at each vertex as an array of length *k* (for quick search on the key) and a dynamic list (to quickly bypass all existing keys). Thus we reach  $O(n)$  the time of the algorithm, but at the cost  $O(nk)$  of memory consumption.

So, we assume that the alphabet size constant, ie, each search operation on the transition character add transition, searching for the next transition - all of these operations, we believe working for  $O(1)$ .

If we look at all parts of the algorithm, it contains three places, the linear asymptotic behavior is not obvious:

- The first place - it is run on suffix links from the state *last* with the addition of ribs on the symbol *c*.
- Second place - up transitions in cloning state *q* to a new state *clone*.
- Third place - redirect transitions leading *q* on *clone*.

We use the well-known fact that the size of the suffix automaton (as the number of states, and the number of transitions) **is linear**. (The proof of the linearity of 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 asymptotic behavior of the total **of the first and second place** : for each operation here adds a new automatic transition.

It remains to estimate the asymptotic behavior of the total **in the third place** - in fact, where we redirect the transitions leading to *q* on *clone*. Denote  $v = \text{longest}(p)$ . This suffix strings *s*, and with each iteration of its length decreases - and, hence, the position *v* as a suffix string *s* monotonically increases with each iteration. In this case, if before the first iteration cycle corresponding row *v* has a depth *k* ( $k \geq 2$ ) of *last* (assuming a depth of suffix number of links that must pass), after the last iteration of the row *v + c* will 2suffix -th link of the path *cur* (which will become the new value *last*).

Thus, each iteration of the loop results in the fact that the string position  $\text{longest}(\text{link}(\text{link}(\text{last}))$  as suffix entire current line will increase monotonically. Consequently, all this cycle could not work more  $n$  iterations, **as required**.

(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 ( $\text{len}$ ,  $\text{link}$ , Jump List). If necessary, here you can add the flag of the terminal, as well as other required information. List of transitions we store in the form of a standard container  $\text{map}$  that allows for total  $O(n)$  memory and  $O(n \log k)$  processing time across the line.

```
struct state {
    int len, link;
    map<char,int> next;
};
```

Suffix machine itself will be stored in an array of these structures  $\text{state}$ . As proved in the next section if  $\text{MAXN}$  is the maximum possible length of the string in the program, it is sufficient to have a memory for the  $2 \cdot \text{MAXN} - 1$  states. Also, we store the variable  $\text{last}$ - state corresponding to the entire line at the moment.

```
const int MAXLEN = 100000;
state st[MAXLEN*2];
int sz, last;
```

We give function initializes Suffix Machine (Machine creates a single initial state):

```
void sa_init() {
    sz = last = 0;
    st[0].len = 0;
    st[0].link = -1;
    ++sz;
    /*
    // этот код нужен, только если автомат строится много раз для разных строк:
    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 Machine:

```
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 memory (up to  $O(nk)$  where  $k$ - the size of the alphabet), the time can be achieved by constructing the machine  $O(n)$  even for all  $k$ - but it will have to be stored in each array of state  $k$  (transition to quickly search for desired letter), and a list of all transitions (for fast traversal or copy all transitions).

## Additional properties suffix automaton

### The number of states

The number of states in the suffix automaton built for line  $s$  length  $n$ , **no more than** $2n - 1$  (for  $n \geq 3$ ).

This is proved by the algorithm described above (as originally machine consists of a single initial state, the first and second steps are added in exactly the same state, and each of the other  $n - 2$  steps could be added on top of two of the splitting condition).

However, this estimate is **easy to show, and without the knowledge of the algorithm**. Recall that the number of states equal to the number of different sets of values  $endpos$ . In addition, these sets  $endpos$  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 there is an interior vertex with one son, then it means that  $endpos$  this son does not contain at least one number of  $endpos$  the parent; then create a virtual vertex with  $endpos$ , equal to this number, and weight gain of the son to the parent. As a result, we obtain a tree in which every internal vertex has degree greater than one, and the number of leaves at most  $n$ . Consequently, such a tree only a maximum  $2n - 1$  peak.

Thus we have shown that assessment independently, without knowledge of the algorithm.

It is interesting to note that this estimate is sharp, ie there is a **test in which it is achieved**. This test is as follows:

"*abbb...b*"

When the processing of the rows on each iteration, beginning with the third split state will occur, and thus, evaluation is achieved  $2n - 1$ .

### The number of transitions

The number of transitions in the suffix automaton built for line  $s$  length  $n$ , **no more than** $3n - 4$  (for  $n \geq 3$ ).

**Let us prove it.**

We estimate the number of continuous transitions. Consider a spanning tree of the longest routes in the machine, starting in state  $t_0$ . This framework will consist of continuous ribs, which means that 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; Suppose that the transition - a transition  $(p, q)$  from the symbol  $c$ . Put it in the appropriate line  $u + c + w$ , where the string  $u$  corresponds to the length of the path from the initial state in  $p$ , and  $w$ - a long way from  $q$  a person into a terminal state. On the one hand, all these lines  $u + c + w$  for all non-continuous transition will be different (as a string  $u$  and  $w$  formed only by continuous transitions). On the other hand, each of these rows  $u + c + w$ , according to the terminal state determination, the entire line will be a suffix  $s$ . Since non-empty suffix in line  $s$  all  $n$  the pieces, and besides, the whole row  $s$  among these lines  $u + c + w$  could not be contained (as the entire line  $s$  corresponds to the path of  $n$  continuous edges), then the total number of non-continuous transitions at most  $n - 1$ .

Adding these two estimates, we obtain an estimate  $3n - 3$ . However, remembering that the maximum number of states can only be achieved on the test species "abbb...b", and to estimate it  $3n - 3$  clearly is not reached, we obtain a final assessment  $3n - 4$ , as required.

Interestingly, there is also a **test on which this estimate is achieved**:

"*abbb...bbbc*"

### Communication with the suffix tree. Construction of suffix tree on the suffix automaton and vice versa

We prove two theorems that establish the mutual relationships between the machine and the suffix suffix tree .

Outset that we believe that the input line is that each has its own suffix suffix tree at the top (as for arbitrary strings, generally speaking, is not true: for example, for a string "aaa..."). Typically, this is achieved by assigning to the end of the line of a special character (usually denoted by a dollar sign).

For convenience, we introduce the following notation:  $\bar{s}$ - a string  $s$  written in reverse order  $DAWG(s)$ - a suffix automaton built for the line  $s$ ,  $ST(s)$  a suffix tree line  $s$ .

We introduce the concept of **expanding links**: fix the top of the suffix tree  $v$  and symbol  $c$ ; then extending the link  $ext[c, v]$  leads to the top of the tree, the corresponding line  $c + v$  (if the path  $c + v$  ends in the middle of the edge, then spend the link in the lower end of the rib); If such a path  $c + v$  does not exist in the tree, then expanding the link is not defined. In a sense, the opposite of expanding links suffix links.

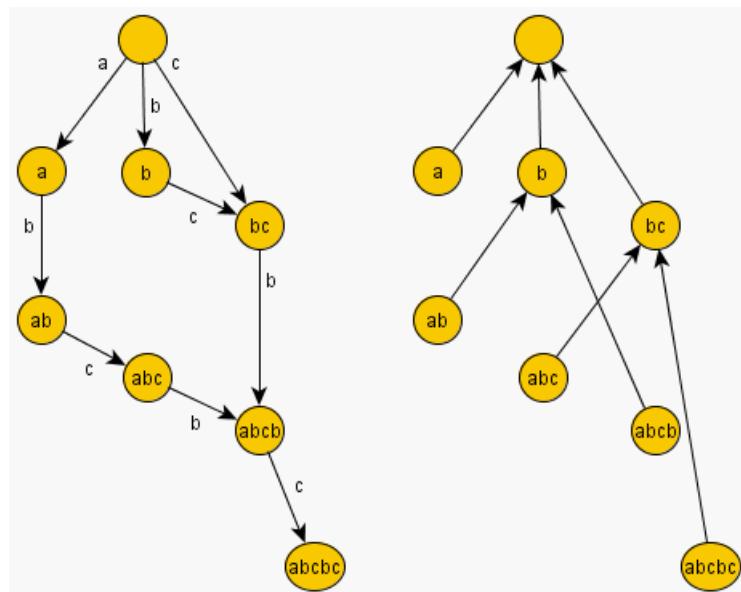
**Theorem 1**. The tree formed by the suffix link  $DAWG(s)$  is a suffix tree  $ST(\bar{s})$ .

**Theorem 2**.  $DAWG(s)$ - a graph expanding links suffix tree  $ST(\bar{s})$ . Furthermore, in the solid edges  $DAWG(s)$  is inverted suffix links  $ST(\bar{s})$ .

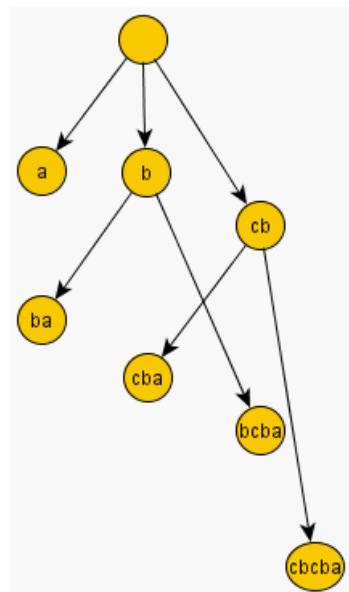
These two theorems allow one of the structures (suffix tree or the suffix automaton) to build another in time  $O(n)$ - these two simple algorithm will be discussed below us in Theorems 3 and 4.

For illustrative purposes, we present automatic suffix tree with its suffix links and the corresponding suffix tree for the inverted row. For example, take the line  $s = "abcbc"$ .

$DAWG("abcbc")$  suffix tree and its references (for clarity we sign each state its *longest*-row):



$ST("cbcba")$ :



**Lemma**. The following three statements are equivalent for any two substrings  $u$  and  $w$ :

- $\text{endpos}(u) = \text{endpos}(w)$  in line  $s$
- $\text{firstpos}(\bar{u}) = \text{firstpos}(\bar{w})$  in line  $\bar{s}$
- $\bar{u}$  and  $\bar{w}$  lie on the same path from the root to the suffix tree  $ST(\bar{s})$ .

The proof of it is pretty obvious: if the beginning of occurrences of the two strings are the same, then one string is a prefix of the other, which means that one line is in the suffix tree in the way of another string.

#### **Proof of Theorem 1 .**

State suffix automaton correspond to the vertices suffix tree.

Consider an arbitrary suffix link  $y = \text{link}(x)$ . According to the definition of suffix links,  $\text{longest}(y)$  is a suffix  $\text{longest}(x)$ , with among all such  $y$  selected the one with the  $\text{len}(y)$  maximum.

In terms of the inverted row  $\bar{s}$ , this means that the suffix link  $\text{link}[x]$  leads to a longest prefix string corresponding to the state  $x$  to this prefix correspond to a different state  $y$ . In other words, the suffix link  $\text{link}[x]$  leads to the top of the ancestor  $x$  in the suffix tree, as required.

#### **The proof of Theorem 2 .**

State suffix automaton correspond to the vertices suffix tree.

Consider an arbitrary shift  $(x, y, c)$  in the suffix automaton  $DAWG(s)$ . The presence of this transition means that  $y$ - is that state the equivalence class that contains the substring  $\text{longest}(x) + c$ . In the inverted row  $\bar{s}$ , this means that  $y$  it is a condition which corresponds to a substring  $\text{firstpos}$  from which (in the text  $\bar{s}$ ) coincides with the  $\text{firstpos}$  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 automaton correspond to the suffix links in the tree. Characterized by a continuous transition discontinuous in that  $\text{length}(y) = \text{length}(x) + 1$ , that after attributing the character  $c$  we were in a state with a string of maximum equivalence class of this state. This means that in the calculation of the corresponding spreading links  $\text{ext}[c, \text{longest}(x)]$  we immediately hit the top of the tree, rather than going down to the nearest tree tops. Thus, assigning a single character in the beginning, we were top of the tree to another - hence, if it is inverted suffix link in the tree.

The theorem is completely proved.

**Theorem 3 .** Having a suffix automaton  $DAWG(s)$  can be a time  $O(n)$  to build a suffix tree  $ST(\bar{s})$ .

**Theorem 4 .** Having a suffix tree  $ST(\bar{s})$  can be a time  $O(n)$  to build a suffix automaton  $DAWG(s)$ .

#### **The proof of Theorem 3 .**

Suffix tree  $ST(\bar{s})$  will contain the same number of vertices as in the states  $DAWG(s)$ , with the top of the tree, obtained from the state  $v$  machine corresponds to the length of the string  $\text{len}(v)$ .

According to Theorem 1, the edges in the tree are formed as inverted suffix links, and arc labels can be found on the basis of the difference between the  $\text{len}$  states, and further knowing for each state of the automaton any one element of his sets  $\text{endpos}$  (this is one element of the set  $\text{endpos}$  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 solid transitions in the machine, and for each such transition  $(x, y)$  to add a link  $\text{link}(y) = x$ .

Thus, over time,  $O(n)$  we can build a suffix tree with suffix links in it.

(If we consider the size  $k$  of the alphabet is not constant, then everything will take time to rebuild  $O(n \log k)$ .)

#### **The proof of Theorem 4 .**

Suffix machine  $DAWG(s)$  will contain the same number of states as vertices  $ST(\bar{s})$ . Each state of  $v$  this long string  $\text{longest}(v)$  will match the inverted path from the root of the tree to the top  $v$ .

According to Theorem 2, to build all the transitions in the suffix automaton, we need to find all references expand  $\text{ext}[c, v]$ .

First, we note that some of these links extend obtained directly from the suffix links in the tree. In fact, if for any vertex  $x$  we consider its suffix link  $y = \text{link}(x)$ , then it means that it is necessary to carry out a spreading of the link  $y$  to  $x$  the first character string corresponding to the vertex  $x$ .

However, since we do not find all references expand. Additionally, it is necessary to go through the suffix tree from

the leaves to the root, and for each vertex  $v$  view of all her sons, for each son see all expanding links  $ext[c, w]$  and copy this link into the top  $v$  if this symbol  $c$  from the top link  $v$  has not yet been found:

$$ext[c, v] = ext[c, w], \quad \text{if } ext[c, w] = nil.$$

This process will work for the time  $O(n)$ , if we consider the size of the alphabet constant.

Finally, it remains to construct suffix links in the machine, however, according to Theorem 1, these suffix links are obtained simply as a suffix tree edges  $ST(\bar{s})$ .

Thus, the algorithm described in the time  $O(n)$  machine for building suffix suffix tree for the inverted row.

(If, however, we believe that the size of  $k$  the alphabet - as a variable, then the asymptotic increase to  $O(n \log k)$ .)

## Application in solving problems

Below we look at what can be done with the help of suffix automaton.

For simplicity, we will assume the alphabet size  $k$  constant, which will allow us to consider the asymptotic behavior of constructing suffix automaton and pass through it constant.

### Verification of entry

**Condition**. Dan text  $T$ , and receives requests in the form: given a string  $P$ , you want to check whether or not the line is included  $P$  in the text  $T$  as a substring.

**Asymptotic behavior**. Preprocessing  $O(\text{length}(T))$  and  $O(\text{length}(P))$  one request.

**Solution**. We construct the suffix automaton text  $T$  during  $O(\text{length}(T))$ .

As now respond to a single request. Suppose that the state - is a variable  $v$ , it is initially equal to the initial state  $t_0$ . Let's go for a character string  $P$ , accordingly making the transition from the current state  $v$  to a new state. If at some time it happened that the transition from the current state to the desired character was not - the answer to the query "no." If we were able to handle the whole line  $P$ , the response to "yes."

It is clear that this will work for the time  $O(\text{length}(P))$ . Moreover, the algorithm actually looking length of the longest prefix  $P$  in the text - and if the input patterns 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  $S$ . You want to know the number of its various substrings.

**Asymptotics**.  $O(\text{length}(S))$ .

**Solution**. We construct the suffix automaton on the line  $S$ .

In any machine suffix substring  $S$  corresponds to a path in the automaton. Since duplicate rows in the machine can not be, then the answer is - it **many different ways** in the machine, starting at the initial vertex  $t_0$ .

Given that the suffix automaton is acyclic graph, the number of different ways it can be considered in using dynamic programming.

Namely, let  $d[v]$  - a number of different ways, starting with the state  $v$  (including the path length of zero). Then right:

$$d[v] = 1 + \sum_{\substack{w : \\ (v, w, c) \in \text{DAWG}}} d[w],$$

that  $d[v]$  can be expressed as the sum of the responses for all possible transitions from the state  $v$ .

The answer to the problem is the value  $d[t_0] - 1$  (unit is taken away, to ignore the 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**.  $O(\text{length}(S))$ .

**Solution**. Solution of the problem is similar to the previous one, only now we have to consider the dynamics of two

quantities: the number of different substrings  $d[v]$  and their total length  $ans[v]$ .

How to count  $d[v]$ , described in the previous problem, but the value  $ans[v]$  can be calculated as follows:

$$ans[v] = \sum_{\substack{w \\ (v, w, c) \in DAWG}} d[w] + ans[w],$$

ie we take the answer for each vertex  $w$ , and add to it  $d[w]$ , as if thereby attributing to the beginning of each line of a single character.

## Lexicographically k-th substring

**Condition**. Given a string  $S$ . Receives requests - the number  $K_i$  and need to find  $K_i$  in the sort order of the substring  $S$ .

**Asymptotics**.  $O(\text{length}(ans) \cdot \text{Alphabet})$  per request (where  $ans$ - is a response to this request,  $\text{Alphabet}$ - the size of the alphabet).

**Solution**. The solution to this problem is based on the same idea as the previous two tasks. Lexicographically  $k$  Star substring - a lexicographical  $k$ th path suffix automaton. Therefore, considering for each state the number of paths from it, we will be able to easily search for  $k$ th path, moving from the root machine.

## The smallest cyclic shift

**Condition**. Given a string  $S$ . Required to find the lexicographically smallest of its cyclic shift.

**Asymptotics**.  $O(\text{length}(S))$ .

**Solution**. We construct the suffix automaton of a string  $S + S$ . Then the machine will contain all the way as the cyclic shifts of the line  $S$ .

Consequently, the problem is reduced to the fact to find in the machine lexicographically minimum path length  $\text{length}(S)$  that is a trivial way: we start in the initial state and each time act greedily, passing the transition with minimal character.

## Number of occurrences

**Condition**. Dan text  $T$ , and receives requests in the form: given a string  $P$ , you want to know how many times the line  $P$  is included in the text  $T$  as a substring (occurrences may overlap).

**Asymptotic behavior**. Preprocessing  $O(\text{length}(T))$  and  $O(\text{length}(P))$  one request.

**Solution**. We construct the suffix automaton in the text  $T$ .

Next we need to do a preprocessing: for each state  $v$  machine to count the number  $cnt[v]$  equal to the size of the set  $\text{endpos}(v)$ . In fact, all the rows corresponding to the same state are included in  $T$  the same number of times equal to the number of positions in the set  $\text{endpos}$ .

However, clearly supports a variety  $\text{endpos}$  for all the states we can not, therefore, learn to count only their size  $cnt$ .

To do this, proceed as follows. For each state, if it has been obtained by cloning (initial state and  $t_0$  we also do not count), initially assign  $cnt = 1$ . Then we will go over all the states in order of their length  $\text{len}$  and are forwarding the current value  $cnt[v]$  on the suffix link:

$$cnt[\text{link}(v)] += cnt[v].$$

It is argued that in the end we did count for each state the correct values  $cnt$ .

Why is this true? All states obtained by cloning is not, exactly  $\text{length}(S)$ , and  $i$  retracement of them came when we added the first  $i$  character. Therefore, each of these states, we associate this position in the processing of which it appeared. Therefore, each such initial state  $cnt = 1$ , and all the other states  $cnt = 0$ .

Then we carry out for each  $v$  such operation:  $cnt[\text{link}(v)] += cnt[v]$ . The meaning of this is that if the row corresponding to the states  $v$ , met  $cnt[v]$  again, all its suffixes will meet the same.

Why thus we will take into account the same position a few times? Because of the value of each state "are forwarding" only once, so there could well be that one state of its value "icing" to some other state twice, in two different ways.

Thus, we have learned to count these values  $cnt$  for all states of the automaton.

After that, the answer to the query is trivial - just back  $cnt[t]$  where  $t$  - the state corresponding to the model  $P$ .

## The position of the first occurrence

**Condition**. Dan text  $T$ , and receives requests in the form: given a string  $P$ , you want to know the position of the start of the first occurrence of the string  $P$ .

**Asymptotic behavior**. Preprocessing  $O(\text{length}(T))$  and  $O(\text{length}(P))$  one request.

**Solution**. We construct the suffix automaton in the text  $T$ .

To solve the problem, we also need to add a preprocessing to find positions  $\text{firstpos}$  for all the states of the automaton, ie, for each state, we want to find the position of  $\text{firstpos}[v]$  the end of the first occurrence. In other words, we want to find the minimum element in advance of each of the sets  $\text{endpos}(v)$  (because clearly support all the sets  $\text{endpos}$  we can not).

Maintain these positions  $\text{firstpos}$  easiest right during the construction of the machine, when we create a new state  $cur$  when entering a function  $sa\_extend()$ , then expose him:

$$\text{firstpos}(cur) = \text{len}(cur) - 1$$

(If we work in 0-indexed arrays).

When cloning tops  $q$  in  $clone$  we put:

$$\text{firstpos}(clone) = \text{firstpos}(q),$$

(Because the other option values, only one - is  $\text{firstpos}(cur)$  that clearly more).

Thus, the answer to the inquiry - it's just  $\text{firstpos}(t) - \text{length}(P) + 1$  where  $t$  - the state corresponding to the model  $P$ .

## The positions of all occurrences

**Condition**. Dan text  $T$ , and receives requests in the form: given a string  $P$ , you want to display the position of all its occurrences in a row  $T$  (entry can overlap).

**Asymptotic behavior**. Preprocessing  $O(\text{length}(T))$ . Response 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 in time order of the input and output.

**Solution**. We construct the suffix automaton in the text  $T$ . Similarly to the previous problem, count in the process of constructing an automaton for each state position  $\text{firstpos}$  the end of the first occurrence.

Suppose now received a request - a string  $P$ . We find, what state  $t$  it corresponds.

It is understood that  $\text{firstpos}(t)$  should definitely go back. What other positions need to find? We took into account the state of the automaton containing a string  $P$ , but does not take into account other conditions, which correspond to lines like that  $P$  is their suffix.

In other words, we need to find all the states from which the **suffix links achievable by state  $t$** .

Therefore, to solve the problem, we need to save for each state list of suffix links leading into it. The answer to the query then would be to make a **detour in the depth / width** of these inverted suffix links, starting from the condition  $t$ .

This tour will run during  $O(\text{answer}(P))$  because we do not visit the same state twice (because of the state of each suffix link goes only one, so it can not be two paths leading to the same state).

However, we must remember that the two states of their values **can be the same** : if one state was obtained by cloning the other. However, this does not impair the asymptotic behavior as each non-cloned maximum peak can be one clone.  $\text{firstpos}$

Moreover, you can easily get rid of repetitive output positions, if we do not add to the response  $\text{firstpos}$  from the states clones. In fact, at any state-clone leads suffix link from the initial state, which is a state of cloned. Thus, if we are able to remember for each flag  $is\_clon$ , and will not add to the response  $\text{firstpos}$  of the states for which  $is\_clon = true$  we thus obtain all the required  $\text{answer}(P)$  positions without repetition.

We give an outline of the implementation:

```
struct state {
    ...
    bool is_clon;
```

```

        int first_pos;
        vector<int> inv_link;
    };

    ... после построения автомата ...
    for (int v=1; v<sz; ++v)
        st[st[v].link].inv_link.push_back (v);
    ...

    // ответ на запрос - вывод всех вхождений (возможно, с повторами)
    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);
    }
}

```

## Search the shortest line, not included in this

**Condition**. Given a string  $S$ , and assign specific alphabet. Required to find a string of minimal length that it does not occur in  $S$  as a substring.

**Asymptotic behavior**. Solution for  $O(\text{length}(S))$ .

**Solution**. Will solve the dynamic programming for machine constructed for the row  $S$ .

Let  $d[v]$  - this is the answer for the top  $v$ , ie, We have already typed in a part of the substring, being able to  $v$ , and want to find the minimum number of characters that must still add to the machine to go beyond finding a nonexistent transition.

Considered to be  $d[v]$  very simple. If, there is no transition at least one character from the alphabet, then  $d[v] = 1$  we can assign 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 equal  $d[t_0]$ , and the string 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 their general of the longest substring, ie a string  $X$  that is a substring of it, and  $S$ , and  $T$ .

**Asymptotic behavior**. Solution for  $O(\text{length}(S) + \text{length}(T))$ .

**Solution**. We construct the suffix automaton on the line  $S$ .

We now go on line  $T$  and look for each prefix longest suffix of the prefixes in  $S$ . In other words, we have for each position in the string  $T$  we want to find the total of the longest substring  $S$ , and  $T$  ending it in this position.

To do this, we will support two variables: **the current state  $v$**  and **the current length  $l$** . These two variables will describe the current matching part: its length and condition, which corresponds to it (without storage length can not be avoided, since one state can match multiple strings of different lengths).

Initially  $p = t_0, l = 0$  ie Match empty.

Now let us consider the character  $T[i]$  and want to recalculate the answer for him.

- If the state of  $v$  the automaton is a transition from the symbol  $T[i]$ , we simply accomplish this transition and increase  $l$  by one.
- If the state of  $v$  the transition not required, we should try to shorten the current matching part, which is necessary to go to suffix link:

$$v = \text{link}(v).$$

In this case, the current length must be shortened, but leave as much as possible. Obviously, this should be assigned  $l = \text{len}(v)$ , as after a rush down the suffix link us to satisfy any substring of length corresponding to this state:

$$l = \text{len}(v).$$

Unless the new state will not go back to the desired character, then again we have to go to the link and reduce the suffix  $l$ , and so on, until we find the transition (then move on to item 1), or we're not busy in the fictional state  $-1$ (which means that symbol  $T[i]$ does not occur in the  $S$ so assign  $v = l = 0$ and move to the next  $i$ ).

The answer to the problem will be the maximum of the values  $l$ for all the time round.

Asymptotic passage is such  $O(\text{length}(T))$ as one move, we can either increment  $l$ or to make multiple passes over the suffix link, each of which would severely reduce the value  $l$ . Therefore, the decrease could not be greater  $\text{length}(T)$ , 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)
        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);
}
```

## Longest common substring problem a few lines.

**Condition** . Given  $K$ lines  $S_i$ . Required to find their general of the longest substring, ie a string  $X$ that is a substring of it all  $S_i$ .

**Asymptotic behavior** . Solution for  $O(\sum \text{length}(S_i) \cdot K)$ .

**Solution** . Glue together all the rows  $S_i$ in one line  $T$ , attributing after each line  $S_i$ its own delimiter character  $D_i$ (ie typing  $K$ extra special. characters  $D_i$ ):

$$T = S_1 D_1 S_2 D_2 \dots S_k D_k.$$

We construct a string  $T$ suffix automaton.

Now we need to find the following line in the machine, which is contained in all rows  $S_i$ , and this will help us spices. symbols. Note that if a substring occurs in some string  $S_j$ , the suffix automaton of this substring exists a path containing the character  $D_j$ , and does not contain other characters  $D_1, \dots, D_{j-1}, D_{j+1}, \dots, D_k$ .

Thus, we need to calculate the reachability: for each state of the automaton, and each character  $D_i$ is there a path that contains a separator  $D_i$ , and containing no other separators. It's easy to make a detour in depth / width or lazy dynamics. After that, the answer to the problem is the string  $\text{longest}(v)$ for the condition  $v$ from which were found in all the way characters.

## Tasks in the online judges

Tasks that can be solved with suffix automaton:

- SPOJ # 7258 SUBLEX "Lexicographical Substring Search" [Difficulty: Medium]

## Literature

We give first a list of the first works related to the suffix automata:

- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, R. McConnell. **Linear Finite Automata Size 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 Text of A** [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 theme has been examined 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 lines** [2006]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 23 Aug 2008 21:00  
EDIT: 5 Apr 2012 23:14

## Finding all subpalindromes

### Statement of the Problem

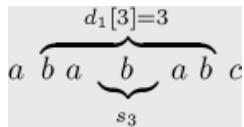
A string  $s$  length  $n$ . You want to find all pairs  $(i, j)$  where  $i < j$  that substring  $s[i \dots j]$  is a palindrome (ie reads the same from left to right and right to left).

### Clarification of statement

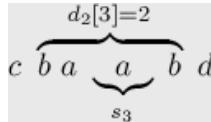
It is clear that in the worst case, such substring-palindromes can be  $O(n^2)$ , and at first glance it seems that the algorithm with linear asymptotic behavior can not exist.

However, information about the found palindromes can return more **compact**: for each position  $i = 0 \dots n - 1$  we find values  $d_1[i]$  and  $d_2[i]$  indicating the number of palindromes respectively odd and even length with center position  $i$ .

For example, in a row  $s = abababc$  there are three odd palindrome length in the center of the symbol  $s[3] = b$ , i.e. value  $d_1[3] = 3$ :



A row  $s = cbaabd$  has two palindrome even length with center symbol  $s[3] = a$ , i.e. value  $d_2[3] = 2$ :



The idea is that if there is a palindrome length  $l$  with center at any position  $i$ , i.e. also subpalindromes length  $l - 2$ ,  $l - 4$  etc. with centers  $i$ . Therefore, two such arrays  $d_1[i]$  and  $d_2[i]$  large enough to store information about all subpalindromes this line.

Rather unexpected fact is that there is a fairly simple algorithm that calculates these "palindromic array"  $d_1$  and  $d_2$  in linear time. This algorithm is described in this article.

### Solution

Generally speaking, this problem has several known solutions: using [hashing technique](#) it can be overcome  $O(n \log n)$ , but with the help of [suffix trees](#) and [fast algorithm LCA](#) this problem can be solved  $O(n)$ .

However, this article describes a method is much simpler and has fewer hidden constants in the asymptotic time and memory. This algorithm was discovered by **Glenn Manakerom (Glenn Manacher)** in 1975

### A trivial algorithm

To avoid ambiguities in further description of the conditions, what is there "trivial algorithm."

This algorithm, which is to find an answer in a position  $i$  repeatedly tries to increase the response unit, each time by comparing a pair of corresponding characters.

This algorithm is too slow, the whole answer as he may deem just in time  $O(n^2)$ .

### Contents [hide]

- Finding all subpalindromes
  - Statement of the Problem
    - Clarification of statement
  - Solution
    - A trivial algorithm
    - Algorithm Manakera
    - Evaluation of the asymptotic behavior of the algorithm Manakera
    - Implementation of the algorithm Manakera
  - Tasks in the online judges

We give clarity 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

Learn how to find all subpalindromes odd length, ie, calculate the array  $d_1[]$ ; solution for palindromes of even length (ie, finding an array  $d_2[]$ ) A slight modification of this.

For a quick calculation will support **the border** $(l, r)$  of the right of the detected subpalindromes (ie subpalindromes with the highest value  $r$ ). Initially, you can put  $l = 0, r = -1$ .

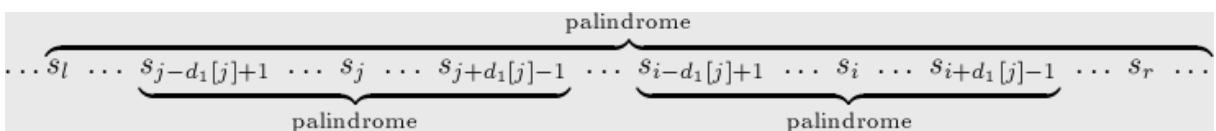
Thus, suppose we want to calculate the value  $d_1[i]$  for the next  $i$ , all previous values  $d_1[]$  already calculated.

- If  $i$  not within the current subpalindromes, ie  $i > r$ , simply do the trivial algorithm.

Ie will gradually increase the value  $d_1[i]$ , and check each time - though whether the current substring  $[i - d_1[i]; i + d_1[i]]$  is a palindrome. When we first find a discrepancy, or when we get to the borders line  $s$ - stop: we finally counted value  $d_1[i]$ . After that, we must not forget to update the values  $(l, r)$ .

- Consider now the case when  $i \leq r$ .

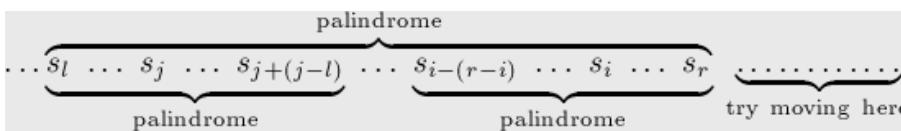
Let us try to extract some information from the already calculated values  $d_1[]$ . Namely, to reflect the position  $j$  in subpalindromes  $(l, r)$ , ie obtain the position  $j = l + (r - i)$  and consider the value  $d_1[j]$ . Because  $j$ - a position symmetrical position  $i$ , then **almost always** we can simply assign  $d_1[i] = d_1[j]$ . An illustration of this reflection (palindrome around  $j$  actually "copied" into a palindrome around  $i$ ):



However, there is a **subtlety** that must be processed correctly when "internal palindrome" reaches the boundary of the outer or climbs over it, that is  $j - d_1[j] + 1 \leq l$  (or what is the same thing  $i + d_1[j] - 1 \geq r$ ). 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 the position  $i$  palindrome is the same length.

In fact, in order to properly handle such situations, it is necessary to "trim" the length subpalindromes, ie assign  $d_1[i] = r - i$ . After that you should let the trivial algorithm that will try to increase the value  $d_1[i]$  as long as possible.

An illustration of this case (the one palindrome with center  $j$  depicts already "cut off" to such a length that it is placed right next to the outside palindrome):



(This illustration shows that, while the center of the palindrome in position  $j$  could be longer, beyond the outer limits of the palindrome, - but in the position  $i$  we use only the portion that fits completely into the outer palindrome. But the answer for the position  $i$  can be greater than this part, so we have to run on a trivial search that will try to push it beyond the outer palindrome, that is to "try moving here".)

At the end of the description of the algorithm became just remind you that we must not forget to update the values  $(l, r)$  after calculating the next value  $d_1[i]$ .

Also reiterated that we have described above reasoning to calculate the array of odd palindromes  $d_1[]$ ; for an

array of even palindromes  $d_2$  completely analogous.

## Evaluation of the asymptotic behavior of the algorithm Manakera

At first glance it is not obvious that this algorithm has a linear asymptotics in the calculation of the response to a particular position in it often starts trivial algorithm to search for palindromes.

However, a more careful analysis shows that the algorithm is still linear. (It should refer to a well-known [algorithm for constructing the Z-line function](#) that is internally 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  $r$  in the course of the algorithm can not occur. Consequently, the trivial algorithm to make a sum of  $O(n)$  actions.

Given that, apart from trivial quest, all other parts of the algorithm Manakera obviously work in linear time, we get the final asymptotics  $O(n)$ .

## Implementation of the algorithm Manakera

For the case subpalindromes odd length, ie, to calculate the array  $d_1$ , we get 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, to compute the array  $d_2$  is 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;
}
```

## Tasks in the online judges

List of tasks that can be taken using this algorithm:

- [UVA # 11475 "Extend to Palindrome"](#) [Difficulty: Low]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 7 Sep 2008 14:40  
EDIT: 24 Aug 2011 18:36

## Lyndon decomposition. Duval algorithm. Finding the smallest cyclic shift

### Contents [hide]

- Lyndon decomposition. Duval algorithm. Finding the smallest cyclic shift
  - The concept of decomposition Lyndon
  - Algorithm Duval
    - Implementation
    - Asymptotics
  - Finding the smallest cyclic shift
  - Tasks in the online judges

## The concept of decomposition Lyndon

Define the concept of **decomposition Lyndon** (Lyndon decomposition).

Line is called **simple** if it is strictly **less than** any of its own **suffix**. Examples of simple lines: *a*, *b*, *ab*, *aab*, *abb*, *ababb*, *abcd*. It can be shown that the line is simple if and only if it is strictly **less than** all of its non-trivial **cyclic shifts**.

Further, suppose that given a string *s*. Then **Lyndon decomposition** line *s* called its decomposition  $s = w_1 w_2 \dots w_k$ , where the lines  $w_i$  are simple, and at the same time  $w_1 \geq w_2 \geq \dots \geq w_k$ .

It can be shown that for any line *s* this expansion exists and is unique.

## Algorithm Duval

**Algorithm Duval** (Duval's algorithm) is on that line length *n* during Lyndon decomposition  $O(n)$  using  $O(1)$  additional memory.

Will work with strings in the 0-indexed.

We introduce the auxiliary concept predprostoy line. Row *t* called **predprostoy** if it has the form  $t = wwww \dots w\bar{w}$  where *w*- some simple string,  $\bar{w}$ - a certain prefix string *w*.

Duval algorithm is greedy. At any point in his work a string *S* is actually divided into three lines  $s = s_1 s_2 s_3$ , where line  $s_1$  Lyndon decomposition has already been found and  $s_1$  is no longer used by the algorithm; line  $s_2$ - a line predprostaya (and the length of simple lines inside it we also remember); line  $s_3$ - this is not treated part of the string *s*. Each time the algorithm Duval takes the first character

$s_3$  and tries to add it to the line  $s_2$ . 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**. Firstly, the pointer will be supported  $i$  on top of the line  $s_2$ . The outer loop algorithm will run until  $i < n$ , that is, until the entire string  $s$  will not go to the line  $s_1$ . Within this cycle are two pointers: a pointer  $j$  to the beginning of the line  $s_3$  (actually a pointer to the next character candidate) and a pointer  $k$  to the current character in the string  $s_2$  with which to compare. Then we will try to add in a cycle symbol  $s[j]$  to the line  $s_2$ , which is necessary to make comparisons with the symbol  $s[k]$ . Here we have three different cases arise:

- If  $s[j] = s[k]$  we can finish the symbol  $s[j]$  to the line  $s_2$  without breaking it "predprostoty." Consequently, in this case, we simply increase the pointers  $j$  and  $k$  one.
- If  $s[j] > s[k]$ , then, obviously, the string  $s_2 + s[j]$  will be easy. Then we increase  $j$  by one, and  $k$  we move back on  $i$  to the next character compared with the first character  $s_2$ .
- If  $s[j] < s[k]$ , then the string  $s_2 + s[j]$  can not be predprostoy. Therefore, we split predprostoyu line  $s_2$  on simple strings plus "residue" (prefix simple string may be empty); simple lines are added in response (ie, derive their position, simultaneously moving the pointer  $i$ ), and "residue" with the symbol  $s[j]$  goes back to a string  $s_3$ , and stops the execution of the inner loop. Thus, we at the next iteration of the outer loop to re-process residue, knowing that he could not form a predprostoyu line with previous simple strings. It remains only to note that in the derivation of simple lines of products we need to know their length; but it obviously is  $j - k$ .

## Implementation

We give an algorithm implementation Duval, which will output the desired decomposition Lyndon line  $s$ :

```
string s; // входная строка
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 a three-pointer  $i, j, k$ .

We now estimate the time of the algorithm.

**The outer loop while** doing no more  $n$ iterations as the end of each iteration, it displays at least one symbol (total symbols and displayed clearly even  $n$ ).

We now estimate the number of iterations of the first inner loop while . For this we consider the second nested loop while - it each time you run your prints a number of  $r \geq 1$ copies of the same simple string of some length  $p = j - k$ . Note that the string  $s_2$ is predprostoy, with its simple lines have a length of just  $p$ , ie its length does not exceed  $rp + p - 1$ . Since the length of the string  $s_2$ is equal  $j - i$ , and the pointer  $j$ is increased by one at each iteration of the first inner loop while, then this cycle will perform no more  $rp + p - 2$ iterations. The worst case is the case  $r = 1$ , and we find that the first nested while loop executes every time no more  $2p - 2$ iterations. Recalling that all displayed  $n$  characters, we find that to output  $n$ characters requires no more  $2n - 2$ iterations of the first nested while-a.

Therefore, the algorithm runs in Duval $O(n)$ .

Easy to estimate the number of character comparisons performed by the algorithm Duval. Since each iteration of the while loop first nested comparisons produces two characters and one comparison is performed after the last iteration of the loop (to understand that the cycle has to stop), then the total **number of comparisons characters** at most  $4n - 3$ .

## Finding the smallest cyclic shift

Given a string  $s$ . We construct a string  $s + s$ Lyndon decomposition (we can do it in  $O(n)$ time and  $O(1)$ memory (if you do not perform concatenation explicitly)).

Let us find predprostoy block that starts at the position at  $n$ (ie, in the first instance of the string  $s$ ), and ends at a position greater than or equal to  $n$  (ie, in the second instance). It is alleged that the starting position of this unit and will be the beginning of the desired cyclic shift (this is easily seen, using the definition of Lyndon decomposition).

Start block predprostogo find easy - just noticed that the pointer  $i$ at the beginning of each iteration of the outer while loop indicates the beginning of the current predprostogo block.

Total we get a realization (to simplify the code it uses  $O(n)$ memory explicitly appending a string to itself):

```
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);
}
```

## Tasks in the online judges

A list of tasks that can be solved using the algorithm Duval:

- UVA # 719 "Glass Beads" [Difficulty: Low]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 14 Sep 2008 14:08  
EDIT: 24 Aug 2011 18:21

## Algorithm Aho-Korasik

Suppose we are given a set of rows in the alphabet size of  $k$  the total length  $m$ . Aho-Korasik algorithm builds for this rowset data structure "Forest" and then on the forest machine building, all of the  $O(m)$  time and  $O(mk)$  memory. The resulting machine can already be used in a variety of tasks, the simplest of them - is to find all occurrences of each line of a given set in some text in linear time.

This algorithm has been proposed by the Canadian scientist Alfred Aho (Alfred Vaino Aho) and scientist Margaret Korasik (Margaret John Corasick) in 1975

### Contents [hide]

- Algorithm Aho-Korasik
  - Bor. Construction of boron
  - Building machine
  - Applications
    - Search from all the rows in a given set of text
    - Finding the lexicographically smallest string of a given length that does not contain any of these samples
    - Finding the shortest line containing occurrences of all samples simultaneously
    - Finding the lexicographically smallest string length  $L$ , containing the data samples in the amount of  $k$  time
  - Tasks in the online judges

## Bor. Construction of boron

Formally, **boron** - the tree rooted at a vertex **Root**, and each edge of the tree signed a letter. If we look at the list of edges emanating from that vertex (except the ribs, leading to the ancestor), all edges must have different labels.

Consider in boron any path from the root; we write the row labels of edges of this path. As a result, we obtain a line that corresponds to this path. If we look at the top of any of boron, then it assign the string corresponding to the path from the root to this node.

Each vertex also has a flag **boron leaf** that is **true**, if the top ends of any row of the set.

Accordingly, **boron build** on the set of strings - then build a forest that each **leaf** vertex of will match any string from the set, and vice versa, each line of the set will fit some kind of **leaf** vertex.

We now describe **how to build a forest** 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 keep boron in the form of an array  $t$  (number of elements in the array - it  $sz$ ) structures **vertex**. The structure **vertex** contains a flag **leaf**, and edges in an array **next**[], where **next[i]** - the index of the top, which leads the edge on the symbol  $i$ , or  $-1$  if there is no such edge.

Initially forest consists of only one vertex - root (agree that the root is always in the array  $t$  index 0). Therefore, the **initialization of boron** is as follows:

```
memset (t[0].next, 255, sizeof t[0].next);
sz = 1;
```

Now implement a function that will **add boron** specified string  $s$ . The implementation is very simple: we get up to the root of boron, look, if there is a transition from the root to the letter  $s[0]$ : if the transition is, then just go for it in other vertex, otherwise create a new node and add a transition to this vertex in the letter  $s[0]$ . Then we stood at some top repeat the process for letters  $s[1]$ , etc. After completing the mark last visited vertex flag  $\text{leaf} = \text{true}$ .

```
void add_string (const string & s) {
    int v = 0;
    for (size_t i=0; i<s.length(); ++i) {
        char c = s[i] - 'a'; // в зависимости от алфавита
        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 ( $O(n)$ ), but due to the increase of the asymptotics of work before  $O(n \log k)$ . It's enough to keep the transitions `next` are not array, and mapping `map < char, int >`.

## Building machine

Suppose we have constructed a forest for a given set of strings. Look at him now, some on the other side. If we consider any vertex, the line that corresponds to it, is a prefix of one or more lines of a set; ie each vertex of boron can be understood as a position in one or more rows in the set.

In fact, the top of the boron can be understood as the state of a **finite deterministic automaton**. Being in any condition, we exposed some input letters go to a different state - ie to another position in the set of strings. For example, if boron is only a line "abc" and we are in a state 2 (which corresponds to the line "ab"), then under the influence of the letters "c" we move into the state 3.

If 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 nevertheless need to go to some state.

More strictly, even if we are in a position  $p$  that corresponds to a particular line  $t$ , and we want to make the transition from the symbol  $c$ . If the boron from the top  $p$  there is a transition in the letter  $c$ , we simply pass on the edge and get to the top, which corresponds to the line  $tc$ . If such an edge is not, then we have to find a state corresponding to proper suffix of the longest line  $t$  (of the longest available in boron), and try to make the transition to the letter  $c$  from him.

For example, suppose that boron is built on the lines "ab" and "bc", and we are under the influence of lines "ab" crossed in a state that is a leaf. Then under the influence of the letters "c" we have to go to the state corresponding to the line "b", and only from there to transition to the letter "c".

**Suffix link** for each node  $p$ - is the pinnacle to which ends of the longest proper suffix of the string corresponding to the vertex  $p$ . The only special case - the root of boron; for convenience 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 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 the suffix links to all the vertices of boron. However, to build these suffix links, we will, oddly enough, on the contrary, with the help of built in the machine transitions.

Note that if we want to know suffix link of a vertex  $v$ , then we can move on to the ancestor of  $p$  the current node (even if  $c$ - the letter on which of  $pa$  transition in  $v$ ), and then go to its suffix link from it and then make the transition from the machine on letter  $c$ .

Thus, the problem of finding the transition has been reduced to that of finding the suffix links, and the problem of finding suffix links - to the problem of finding the suffix links and go, but for the closer to the root vertex. We got a recursive relationship, but not infinite, and, moreover, which can solve the linear time.

We now turn to **implementation**. Note that we now need to store each vertex its ancestor  $p$ , as well as a symbol  $pch$  for which there is a transition from an ancestor in our top. Also, at each node will store int  $link$ - suffix link (or  $-1$ , if it is not calculated), and an array int  $go[k]$  transitions in the machine for each of the characters (again, if an array element is equal  $-1$ , it is not calculated). We give now 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 not difficult to understand that, due to memorize found suffix links and transitions, the total time finding all suffix links and transitions will be linear.

## Applications

### Search from all the rows in a given set of text

Given a set of lines, and given text. You want to display all occurrences of a set of all the rows in the text for the time  $O(\text{Len} + \text{Ans})$  where  $\text{Len}$ - the length of the text  $\text{Ans}$ - the size of the response.

We construct from a given set of strings boron. We now process text one letter, moving accordingly on a tree, in fact - as of the machine. Initially, we are at the root of the tree. Let us at the next step, we are in a position  $v$ , and the next letter of the text  $c$ . Then make the transition to the state  $\text{go}(v, c)$ , thereby increasing or on the length of the current matching substring, or reducing it, passing through the suffix link.

As it is now found on the current state  $v$ , there is a match with some lines from the set? Firstly, it is clear that if we stand in marked vertex ( $\text{leaf} = \text{true}$ ), then there is a match with the sample, which ends at the top of boron  $v$ . However, this is not the only case of coincidence achieve if we, moving suffix links, we can achieve one or more of the labeled peaks, the agreement will also be, but the sample ending in these states. A simple example of such a situation - when a set of rows - this  $\{"dabce", "abc", "bc"\}$ , and the text - it is  $"dabc"$ .

Thus, if each labeled vertices stored sample number that ends in it (or a list of numbers, if allowed duplicate samples), we can for the current state of the  $O(n)$  room to find all of the samples, which reached a match, simply click on the suffix links from the current to the root apex. However, this is not enough effective solution, since the amount of the asymptotics is obtained  $O(n \cdot \text{Len})$ . However, you may notice that traffic on the suffix links can be optimized previously counted for each vertex closest to the marked vertices reachable by suffix links (this is called "output function"). This value can be considered lazy dynamics in linear time. Then for the current node we can for  $O(1)$  finding a suffix following the path marked with the top, ie, next match. Thus, for each match will be spent  $O(1)$  actions and obtain the asymptotic sum

$O(\text{Len} + \text{Ans})$ .

In a simple case, when it is necessary to find not the entry, but only their number, you can substitute the output function to count the number of labeled lazy dynamics of vertices reachable from the current node  $v$  for suffix links. This value can be calculated for  $O(n)$  an amount and then the current state  $v$  we can for the  $O(1)$  number of occurrences found in the text of all the samples, ending at the current position. 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 any of these samples

Given a set of samples, and given length  $L$ . You want to find the length of the string  $L$  that does not contain any of the samples, and of all such strings lexicographically smallest output.

We construct from a given set of strings boron. Recall now that the vertices, of which suffix links can be achieved marked peaks (peaks such as can be found for  $O(n)$  example, lazy dynamics) can be interpreted as the occurrence of any of a set of rows in the specified text. Because of 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 from the machine all the "bad" vertices, and in the remaining graph automaton is required to find the lexicographically smallest path length  $L$ . This problem can be solved already  $O(L)$ , such as depth-first search .

## Finding the shortest line containing occurrences of all samples simultaneously

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 ( $v = \text{Root}$ ,  $\text{Msk} = 0$ ) is required to reach the state ( $v$ ,  $\text{Msk} = 2^n - 1$ ) where  $n$ - the number of samples. Transitions from one state will be adding one letter to the text, ie, transition along the edge machine to another top with a corresponding change in the mask. Running round in width at this column, we'll find the way to a state ( $v$ ,  $\text{Msk} = 2^n - 1$ ) of minimal length that we just needed to.

## Finding the lexicographically smallest string length $L$ , containing the data samples in the amount of $k$ time

As in the previous tasks, for each vertex count the number of times that corresponds to it (ie, the number of labeled vertices reachable from it by suffix links). Reformulate the problem in this way: the current status is determined by three numbers ( $v$ ,  $\text{Len}$ ,  $\text{Cnt}$ ), and requires from the state ( $\text{Root}$ ,  $0$ ,  $0$ ) to come to a state ( $v$ ,  $L$ ,  $k$ ) where  $v$ - any vertex. Transitions between states - it's just the ribs machine transitions from the current node. Thus, it is enough just to find a circuit in depth way between these two states (if the bypass will look in depth letters in their natural order, then found a way to automatically lexicographically least).

## Tasks in the 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]

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: Mar 1  
EDIT: 4 F

## Suffix tree. Ukkonen's algorithm

This article - a temporary cap, and contains no descriptions.

Ukkonen algorithm description can be found, for example, in the book Gasfilda "strings, trees and sequences in the algorithms."

### Contents [hide]

- Suffix tree. Ukkonen's algorithm
  - Implementation of the algorithm Ukkonen
  - Compressed implementation
  - Tasks in the online judges

## Implementation of the algorithm Ukkonen

This algorithm builds a suffix tree for a given string  $s$  and its length  $n$ , in a time  $O(n \log k)$  where  $k$  - the size of the alphabet (if it is assumed constant, the asymptotic behavior is obtained  $O(n)$ ).

The inputs to the algorithm are the string  $s$  and its length  $n$ , which are transmitted in the form of global variables.

The main feature - `build_tree` builds a suffix tree. The tree is stored as an array of structures `node`, where `node[0]` - the root suffix tree.

In order to ease code edges are stored in the same structures: for each vertex in its structure `node` contains data about the edges contained in its parent. Total in each `node` store:  $(l, r)$  defining the label  $s[l..r - 1]$ ; edges in ancestor `par`; the top-ancestor `link`; suffix link `next`; list of outgoing edges `next`.

```

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 give also the following compact implementation of the 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,la;

void ukkadd (int c) {
    suff:;
    if (r[tv]<tp) {
        if (t[tv][c]==-1) { t[tv][c]=ts; l[ts]=la;
            p[ts+1]=tv; tv=s[tv]; tp=r[tv]+1; goto suff; }
        tv=t[tv][c]; tp=l[tv];
    }
    if (tp==-1 || c==a[tp]-'a') tp++; else {
        l[ts+1]=la; 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]-'a']=tv;
        l[tv]=tp; p[tv]=ts; t[p[ts]][a[l[ts]]-'a']=ts; ts+=2;
        tv=s[p[ts-2]]; tp=l[ts-2];
        while (tp<=r[ts-2]) { tv=t[tv][a[tp]-'a']; 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() {
    ts=2;
    tv=0;
    tp=0;
    fill(r,r+N,(int)a.size()-1);
    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 (la=0; la<(int)a.size(); ++la)
        ukkadd (a[la]-'a');
}

```

The same code, commented:

```

const int N=1000000,      // максимальное число вершин в суффиксном дереве
        INF=1000000000, // константа "бесконечность"
string a;                // входная строка, для которой надо построить дерево
int t[N][26],           // массив переходов (состояние, буква)
    l[N],               // левая
    r[N],               // и правая границы подстроки из a, отвечающие ребру, входящему в вершину
    p[N],               // предок вершины
    s[N],               // суффиксная ссылка
    tv,                 // вершина текущего суффикса (если мы посередине ребра, то нижняя вершина ребра)
    tp,                 // положение в строке соответствующее месту на ребре (от l[tv] до r[tv] включительно)
    ts,                 // количество вершин
    la;                 // текущий символ строки

void ukkadd(int c) { // дописать к дереву символ с
    suff;;           // будем приходить сюда после каждого перехода к суффиксу (и заново добавлять символ)
    if (r[tv]<tp) { // проверим, не вылезли ли мы за пределы текущего ребра
        // если вылезли, найдем следующее ребро. Если его нет - создадим лист и прицепим к дереву
        if (t[tv][c]==-1) {t[tv][c]=ts;l[ts]=la;p[ts++]=tv;tv=s[tv];tp=r[tv]+1;goto suff;}
        tv=t[tv][c];tp=l[tv]; // в противном случае просто перейдем к следующему ребру
    }
    if (tp===-1 || c==a[tp]-'a') tp++; else { // если буква на ребре совпадает с с то идем по ребру, ε
        // разделяем ребро на два. Посередине - вершина ts
        l[ts]=l[tv];r[ts]=tp-1;p[ts]=p[tv];t[ts][a[tp]-'a']=tv;
        // ставим лист ts+1. Он соответствует переходу по с.
        t[ts][c]=ts+1;l[ts+1]=la;p[ts+1]=ts;
        // обновляем параметры текущей вершины. Не забыть про переход от предка tv к ts.
        l[tv]=tp;p[tv]=ts;t[p[ts]][a[l[ts]]-'a']=ts;ts+=2;
        // готовимся к спуску: поднялись на ребро и перешли по суффиксной ссылке.
        // tp будет отмечать, где мы в текущем суффиксе.
        tv=s[p[ts-2]];tp=l[ts-2];
        // пока текущий суффикс не кончился, топаем вниз
        while (tp<=r[ts-2]) {tv=t[tp][a[tp]-'a'];tp+=r[tp]-1[tp]+1;}
        // если мы пришли в вершину, то поставим в нее суффиксную ссылку, иначе поставим в ts
        // (ведь на след. итерации мы создадим ts).
        if (tp==r[ts-2]+1) s[ts-2]=tv; else s[ts-2]=ts;
        // устанавливаем tp на новое ребро и идем добавлять букву к суффиксу.
        tp=r[tp]-(tp-r[ts-2])+2;goto suff;
    }
}

void build() {
    ts=2;
    tv=0;
    tp=0;
    fill(r,r+N,(int)a.size()-1);
    // инициализируем данные для корня дерева
    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);
    // добавляем текст в дерево по одной букве
    for (la=0; la<(int)a.size(); ++la)
        ukkadd (a[la]-'a');
}

```

## Tasks in the online judges

Tasks that can be solved using the suffix tree:

- UVA # 10679 "I Love Strings !!!" [Difficulty: Medium]

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: Sep 6, 2011 1:  
EDIT: 11 Feb 2012 15

## Search all tandem repeats in the row. Algorithm Maine-Lorentz

A string strength  $n$ .

**Tandem repeats** (tandem repeat) it called two occurrences of a substring in a row. In other words, the tandem repeat described by a pair of indices  $i < j$  such that substring  $s[i \dots j]$  - two identical strings recorded in a row.

The challenge is to **find all tandem repeats**. A simplified version of this problem: to find **any** tandem repeat or to **find the longest** tandem repeat.

**Note**. To avoid confusion, all the lines in the article, we will assume a 0-indexed, ie the first character has an index of 0.

Algorithm described here was published in 1982, Maine and Lorenz (see. References).

### Example

Consider the example 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 length  $n$  may be of the order  $O(n^2)$ .

An obvious example is a string composed of  $n$  identical letters - in this line of tandem repeats is any substring of even length, which is about  $n^2/4$ . In general, any string is periodic with a 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 asymptotic behavior  $O(n \log n)$  as the algorithm can produce tandem repeats in some compressed form, in groups of several pieces at once.

Moreover, there is the concept of **the series** - quadruples of numbers that describe the entire group of the periodic substrings. It has been proved that the number of runs in each line are linearly with respect to the 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 give here some other interesting results related to the number of tandem repeats:

- We know that if we consider only primitive tandem repeats (ie, those halves which are not multiples of strings), their number in any row -  $O(n \log n)$ .
- If encode tandem repeats of three numbers (a triple Krochemora (Crochemore))  $(i, p, r)$  (where  $i$  - starting position,  $p$  - the length of the repeating substring  $r$  - the number of repeats), all tandem repeats any line can be derived with the help of  $O(n \log n)$  such triples. (This is the result obtained at the output of Algorithm C find all tandem repeats.)
- Fibonaccis defined as follows:

$$\begin{aligned} t_0 &= b, \\ t_1 &= a, \\ t_i &= t_{i-1} + t_{i-2}, \end{aligned}$$

are "strongly" periodical.

The number of tandem repeats in the  $i$ -Fibonacci th row length  $f_i$ , even compressed using triple Krochemora amounts  $O(f_n \log f_n)$ .

Number of primitive tandem repeats in the Fibonacci lines - also of the order  $O(f_n \log f_n)$ .

### Contents [hide]

- **Search all tandem repeats in the row. Algorithm Maine-Lorentz**
  - Example
  - The number of tandem repeats
  - Algorithm Maine-Lorentz
    - Search crossing of tandem repeats
      - Right and left tandem repeats
      - The central position of  $cntr$ -tandem repeat
      - The criterion of the presence of tandem repeat with a given center  $cntr$
    - Algorithm for finding the length  $k_1$  and  $k_2$ 
      - Search Rules of tandem repeats
      - Asymptotics
  - Implementation
  - Literature

## Algorithm Maine-Lorentz

The idea of the algorithm Maine-Lorentz is fairly standard: an algorithm "**divide-and-rule**".

In brief, it consists in the fact that the original string is divided in half, the solution runs from each of the two halves separately (in this way we find all of the tandem repeats, which are located only in the first or only in the second half). Then there is the hardest part - is finding tandem repeats beginning in the first half and the second ending (we call these tandem repeats for the convenience of **crossing**). How it's done - and is the very essence of the algorithm Maine-Lorentz; what we describe in detail below.

Asymptotic behavior of the 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  $n$  of  $O(n)$ , the final asymptotic behavior of the whole algorithm will  $O(n \log n)$ .

### Search crossing of tandem repeats

Thus, the algorithm Maine-Lorentz solves to ensure that on a given line  $s$  all learn to look for tandem repeats, ie those that begin in the first half of the line, and an end - in the second.

We denote by  $u$  and  $v$  the two halves of the string  $s$ :

$$s = u + v$$

(Their length approximately equal to the length of the string  $s$  one, divided in half).

#### Right and left tandem repeats

Consider an arbitrary tandem repeat and look at its middle symbol (or rather, to that symbol, which begins the second half of the tandem, ie, if the tandem repeat - a substring  $s[i \dots j]$ , the middle character is  $(i + j + 1)/2$ ).

Then we call tandem repeat **left** or **right** depending on which is the symbol - in the line  $u$  or string  $v$ . (You can say so: tandem repeat is called left if most of it is in the left side of the line  $s$ , otherwise - tandem repeat is called the right.)

Learn to look for **all the left tandem repeats**; for right everything will be similar.

#### The central position of $cntr$ tandem repeat

We denote the length of the desired left through the tandem repeat  $2k$  (ie, the length of each half tandem repeat - it  $k$ ). Consider the first character of the tandem repeat entering the line  $v$  (he stands in line  $s$  at the position  $\text{length}(u)$ ). It coincides with the character standing in the  $k$  position before him; denote this position through  $cntr$ .

**Search all tandem repeats we will be going over this position  $cntr$** : that is, first find all tandem repeats with one value  $cntr$ , then with another value, etc. - Through all the possible values  $cntr$  of  $0$  up to  $\text{length}(u) - 1$ .

**For example**, consider the following line:

$$s = "cac|ada"$$

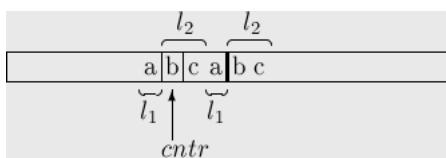
(The vertical line character separates the two halves  $u$  and  $v$ )

Tandem repeat "caca" contained in this line will be found when we view the value  $cntr = 1$  - because it is in a position 1 worth symbol 'a', coinciding with the first character of a tandem repeat themselves in half  $v$ .

#### The criterion of the presence of tandem repeat with a given center $cntr$

So we must learn to fix the value  $cntr$  to quickly search all tandem repeats, corresponding to it.

Obtain such a scheme (for the abstract line, which contains a tandem repeat "abcabc"):



Here through  $l_1$  and  $l_2$  we denote the length of the two pieces of tandem repeat:  $l_1$  - the length of the tandem repeat to the position  $cntr - 1$ , and  $l_2$  - this is the length of the tandem repeat of the  $cntr$  half until the end of the tandem repeat. Thus,  $l_1 + l_2 + l_1 + l_2$  - the length of the tandem repeat.

Looking at this picture, we can see that a **necessary and sufficient** condition that a center position  $cntr$  is tandem repeat length  $2l = 2(l_1 + l_2) = 2(\text{length}(u) - cntr)$ , is the following condition:

- Let  $k_1$  - this is the largest number such that  $k_1$  symbols before the position  $cntr$  coincide with the last  $k_1$  character string  $u$ :

$$u[cntr - k_1 \dots cntr - 1] == u[\text{length}(u) - k_1 \dots \text{length}(u) - 1].$$

- Let  $k_2$  - this is the largest number such that  $k_2$  characters from position  $cntr$ , same as the first  $k_2$  character string  $v$ :

$$u[cntr \dots cntr + k_2 - 1] == v[0 \dots k_2 - 1].$$

- Then it must be true:

$$\begin{cases} l_1 \leq k_1, \\ l_2 \leq k_2. \end{cases}$$

This criterion can be **reformulated** in such a way. We fix a specific value  $cntr$ , then:

- All tandem repeats, which we will now discover, will have a length  $2l = 2(\text{length}(u) - cntr)$ .

However, these tandem repeats may 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 length of the pieces  $l_1$  and  $l_2$  satisfy the 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 length $k_1$ and $k_2$

So the whole problem is reduced to a rapid calculation of lengths  $k_1$  and  $k_2$  for each value  $cntr$ .

We recall their definition:

- $k_1$  - 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:

$$u[cntr \dots cntr + k_2 - 1] == v[0 \dots k_2 - 1].$$

On both of these requests can be responsible for  $O(1)$  using the **algorithm for finding the Z-function**:

- To quickly find the values of  $k_1$  pre-count function for Z-line  $\bar{u}$  (ie, line  $u$ , written out in reverse order).

Then the value  $k_1$  for a particular  $cntr$  will simply equal to the corresponding value of the array Z-function.

- To quickly find the values of  $k_2$  pre-count function for Z-line  $v + \# + u$  (ie, the string  $v$  assigned to the line  $v$  through the separator character).

Again, the value  $k_2$  for a particular  $cntr$  need to take a cell from the corresponding Z-functions.

### Search Rules of tandem repeats

Up to this point we have only worked with the left tandem repeats.

To search for the right tandem repeats, act the same way: we define the center  $cntr$  as a symbol corresponding to the last character of the tandem repeat, were in the first row.

Then the length  $k_1$  will be determined as the number of characters to the position  $cntr$  inclusive, coinciding with the last character  $u$ . The length  $k_2$  will be determined as the maximum number of characters from  $cntr + 1$  coinciding with the first character  $v$ .

Thus, to quickly find  $k_1$  and  $k_2$  will have to count predetermined function for Z-lines  $\bar{u} + \# + \bar{v}$  and  $v$ , respectively. Thereafter, going through the specific value  $cntr$ , we the same criteria will find all the upright tandem repeats.

### Asymptotics

Asmptotika algorithm Maine-Lorentz be thus  $O(n \log n)$ : Since this algorithm is an algorithm "divide-and-conquer", each recursive invocations which operates in a time linear in the length of the string: for four lines in linear time looking for them **Z-function**, and then moves the value  $cntr$  and displays all groups detected tandem repeats.

Tandem repeats found algorithm Maine-Lorentz in the form of original **groups**: those fours  $(cntr, l, k_1, k_2)$ , each of which represents a group tandem repeats of length  $l$ , the center  $cntr$  and all sorts of lengths of 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 give an algorithm implementation Maine-Lorentz, which for the time  $O(n \log n)$  is all tandem repeats this line in a compressed form (as a group described by fours numbers).

For demonstration purposes, tandem repeats found during  $O(n^2)$  "unchain" and displayed separately. This conclusion is in solving real-world problems easily be replaced by some other, more effective action, such as the search of the longest tandem repeat, or counting 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 ll, 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 ll=1; ll<=l; ++ll) {
            if (left && ll == l) break;
            if (ll <= k1 && l-ll <= k2)
                output_tandem (s, shift, left, cntr, l, ll, l-ll);
        }
    }

    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 lines** [2006]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:02  
EDIT: 13 Jun 2008 1:37

## The task RMQ (Range Minimum Query - at least in the interval)

### Contents [hide]

- The task RMQ (Range Minimum Query - at least in the interval)
  - Apps
  - Solution

Given an array A [1..N]. Receives requests like (L, R), for each query is required to find the minimum of the array A, starting from a position of L and ending at R.

### Apps

In addition to the direct application in a variety of tasks are the following:

- The task of LCA (lowest common ancestor)

### Solution

RMQ task is solved by the data structures.

Described on the website of the data structures can be selected:

- **Sqrt-decomposition** - responds to a request for O ( $\sqrt{N}$ ), preprocessing of O (N).  
The advantage is that it is a very simple data structure. Disadvantage - asymptotics.
- **Segment tree** - responds to a request for O ( $\log N$ ), preprocessing of O (N).  
The advantage of - good asymptotic behavior. Drawback - a large amount of code compared to other data structures.
- **Fenwick tree** - responds to a request for O ( $\log N$ ), preprocessing of O ( $N \log N$ )  
Advantage - very quickly written up and running is also very fast. But a significant disadvantage - Fenwick tree can only respond to inquiries with L = 1, which for many applications is not applicable.

Note. "Preprocessing" - a pre-processing of the array A, in fact, is to build a data structure for a given array.

Now suppose that the array A **can be changed** during operation (that will also do requests for change values in some interval [L; R]). Then the resulting problem can be solved by **Sqrt-decomposition** and the **segment tree**.

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 11 Jun 2008 11:06  
EDIT: 7 Nov 2011 15:45

## Finding of the longest increasing subsequence

**Condition of the problem** as follows. Given an array of  $n$  integers:  $a[0 \dots n - 1]$ . To find in this sequence is strictly increasing subsequence of maximal length.

**Formally**, it looks like this: you want 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 different algorithms for solving 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 - this is a very common technique that allows to solve a huge class of problems. Here we consider this technique as applied to our particular problem.

Learn how to look for **the length** of the longest increasing subsequence, and the restoration of the subsequence zaymëmsya later.

### Dynamic programming to find the length of the response

To do this, let us learn to consider an array  $d[0 \dots n - 1]$  where  $d[i]$  is the length of the longest increasing subsequence, ending it at index  $i$ . This array (he is - the very dynamics), we assume gradually: first  $d[0]$ , and then  $d[1]$ , etc. In the end, when the weight is calculated by us, the answer to the problem will be equal to the maximum in the array  $d[]$ .

So, let the current index -  $i$  ie we want to calculate the value  $d[i]$ , and all the previous values  $d[0] \dots d[i - 1]$  already calculated. Then we note that we have two options:

- either  $d[i] = 1$ , i.e. desired subsequence consists of numbers only  $a[i]$ .
- either  $d[i] > 1$ . Then before the number  $a[i]$  in the desired subsequence worth some other number.  
Let's brute over this number: it can be any element , but so what . Let us consider some current index . Since the dynamics for it already calculated, it turns out that this number with the number gives the answer . Thus, we can assume by the following formula:  

$$a[j] (j = 0 \dots i - 1) a[j] < a[i] d[j] a[j] a[i] \\ d[j] + 1 d[i]$$

$$d[i] = \max_{\substack{j=0 \dots i-1, \\ a[j] < a[i]}} (d[j] + 1).$$

By combining the two in one embodiment obtain a final algorithm for computing  $d[i]$ :

$$d[i] = \max \left( 1, \max_{\substack{j=0 \dots i-1, \\ a[j] < a[i]}} (d[j] + 1) \right).$$

This algorithm - is the very dynamics.

### Contents [hide]

- Finding of the longest increasing subsequence
  - Solution for  $O(n^2)$ : dynamic programming method
    - Dynamic programming to find the length of the response
    - Implementation
    - Restoring response
    - Implementation of the recovery response
    - An alternative method for restoring a response
  - Solution for  $O(n \log n)$ : dynamic programming with binary search
    - Implementation of the  $O(n \log n)$
    - Restoring response
  - Solution for  $O(n \log n)$ : data structures
  - Related tasks
    - Of the longest non-decreasing subsequence
    - Number of the longest increasing subsequence
    - The smallest number of non-increasing subsequences covering this sequence
  - Tasks in the online judges

## Implementation

We present the implementation of the above algorithm, which finds and displays the length of the longest increasing subsequence:

```

int d[MAXN]; // константа MAXN равна наибольшему возможному значению 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;

```

## Restoring response

So far we have only learned to look for the length of the answer, but the very of the longest subsequence, we can not withdraw because do not store any additional information about where reaches its maximum.

To be able to restore the response, in addition to the dynamics  $d[0 \dots n - 1]$  also need to keep an auxiliary array  $p[0 \dots n - 1]$  - then, in what place peaked for each value  $d[i]$ . In other words, the index  $p[i]$  will be denoted by the same index  $j$  at which get the highest value  $d[i]$ . (This array  $p$  in dynamic programming is often called "an array of ancestors".)

Then, in order to deduce the answer, you just have to go on the item with the maximum value  $d[i]$  for his ancestors as long as we do not derive all subsequence, ie until we reach the elements having a value  $d = 1$ .

## Implementation of the recovery response

So, we will change the dynamics of the code, and add the code that produces the conclusion of the longest subsequence (output index subsequence of elements in the 0-indexed).

For convenience, we initially put codes  $p[i] = -1$ : for elements whose dynamics turned out to be unity, this value will remain ancestor minus one, that a little bit easier to answer when restoring.

```

int d[MAXN], p[MAXN]; // константа MAXN равна наибольшему возможному значению 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 method for restoring a response

However, as is almost always the case in dynamic programming, to restore the response, you can not store an additional array of ancestors  $p[]$ , but simply re-counting the current element dynamics and looking at what is the index maximum was reached.

This method leads to the implementation of a little more than a long code, but instead we get memory savings and absolute coincidence of the program logic in the process of calculating the dynamics and in 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 will understand how it is possible to speed up this option  $O(n \log n)$ .

**Dynamics** will now be as follows: let - is the number by which ends increasing subsequence of length (and if such numbers are few - then the least of them). $d[i](i = 0 \dots n)$

Initially, we believe  $d[0] = -\infty$ , and all the other elements  $d[i] = \infty$ .

Consider this momentum, we will gradually, the number of processed  $a[0]$ , then  $a[1]$ , etc.

We present the implementation of this dynamic  $O(n^2)$ :

```

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];

```

We now note that this dynamic is one **very important property** :  $d[i - 1] \leq d[i]$  for all  $i = 1 \dots n$ . Another property - that each element  $a[i]$  updates a maximum of one cell  $d[j]$ .

Thus, it means that the next process  $a[i]$ , we can for  $O(\log n)$  making a binary search through the array  $d[]$ . In fact, we're just looking for in the array  $d[]$  the first number, which is strictly more  $a[i]$ , and try to update this element similarly found above implementation.

## Implementation of the $O(n \log n)$

Using standard in C ++ binary search algorithm *upper\_bound*(which returns the position of the first element is strictly greater than this), 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];
}

```

## Restoring response

According to this speaker, too, you can restore a response, which again, in addition to the dynamics  $d[i]$  also need to store an array of "ancestors"  $p[i]$  - is on the element with which the index ends the optimal subsequence length  $i$ . In addition, for each element of the array  $a[i]$  will have to keep his "ancestor" - ie, the index of the element to face the  $a[i]$  best in the subsequence.

By keeping these two arrays during the computation of the dynamics, at the end it would be easy to restore the desired subsequence.

(It is interesting to note that in relation to the dynamics of the response can be restored only way through the array of ancestors - and without them recover after calculating the dynamics of response 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  $O(n \log n)$  very beautiful, but it's not trivial ideological, then there is another way: use one of the known simple data structures.

In fact, let's go back to the very first dynamic, where the state is simply the current position. Current dynamics value  $d[i]$  is calculated as a maximum value  $d[i]$  among all elements  $j$  that  $a[j] < a[i]$ .

Therefore, if we through  $t[]$  denote such an array , which will record the values of the dynamics of the numbers:

$$t[a[i]] = d[i],$$

it turns out that all we should be able to - is to seek **maximum prefix** array  $t: t[0 \dots a[i] - 1]$ .

The task of searching for the maximum prefix array (given the fact that the array may vary) solved many standard data structures, such as [wood pieces](#) or [wood Fenwick](#) .

Using any such data structure, we obtain a solution for  $O(n \log n)$ .

In this method, the solution is a clear **shortcomings** : the length and complexity of the implementation of this road will in any case worse than described above for the dynamics  $O(n \log n)$ . In addition, if the input number  $a[i]$  can be quite large, it is likely that they will have to compress (ie renumbering from 0 to  $n - 1$ ) - without this many standard data structures can not work due to the high memory consumption.

On the other hand, in this way there are **the advantages** . Firstly, with this method of solution will not have to worry about the tricky dynamics. Secondly, this method allows us to solve some generalizations of our problem (see about them. Below).

## Related tasks

We give here a few problems that are closely connected with the problem of the longest increasing subsequence search.

### 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, simply by changing the sign of inequality comparisons, and it will be necessary to slightly change the binary search.

### Number of the longest increasing subsequence

To solve this problem, you can use the very first of the dynamics of  $O(n^2)$  any approach using data structures to solve for  $O(n \log n)$ . And in fact, in that case all changes are only in that, in addition dynamics values  $d[i]$  must also store the number of ways this value could be obtained.

Apparently, the way to solve through the dynamics of  $O(n \log n)$  a given problem can not be applied.

## The smallest number of non-increasing subsequences covering this sequence

**Conditions** such. Given an array of  $n$  integers  $a[0 \dots n - 1]$ . Want to color it in the least number of colors so that each color turns to a non-increasing subsequence.

**Solution**. It is alleged that the minimum number of colors required is the length of the longest increasing subsequence.

**Proof**. In fact, we need to prove **the duality** of this problem and the problem of search of the longest increasing subsequence.

Denoted by  $x$  the length of the longest increasing subsequence, and through  $y$ - the required minimum number of non-increasing subsequences. We need to prove that  $x = y$ .

On the one hand, it is understandable why can not be  $y < x$ , because if we have a  $x$  strictly increasing elements, no two of them could not get in a non-increasing subsequence, and therefore  $y \geq x$ .

We now show that, on the contrary,  $y$  can not be  $> x$ . We prove this by contradiction: suppose  $y > x$ . Then consider any optimal set of  $y$  non-increasing subsequences. We transform this set as follows: as long as there are two such subsequences that first begins before the second, but the first begins with a number greater than or equal to the start of the second - to unhook it from the first starting number subsequence and trailers at the beginning of the second. Thus, after a finite number of steps we will have  $y$  sub-sequences, and their starting numbers will form an increasing subsequence of length  $y$ . However  $y > x$ , i.e. we have a contradiction (in fact can not be increasing subsequence of length  $x$ ).

Thus, in fact,  $y = x$  that we wanted to prove.

**Restoring response**. It is argued that the very desired partition into subsequences can be searched eagerly, ie, going from left to right and attributing the current number in the subsequence, which now ends at the smallest number greater than or equal to the current.

## Tasks in the online judges

A list of tasks that can be solved on the subject:

- [MCCME # 1793 "Longest increasing subsequence of O \(n \\* log \(n\)\)"](#) [Difficulty: Low]
- [TopCoder SRM 278 "500 IntegerSequence"](#) [Difficulty: Low]
- [TopCoder SRM 233 "DIV2 1000 Automarket"](#) [Difficulty: Low]
- [Ukrainian School Olympiad in Informatics - task F "Tourist"](#) [Difficulty: Medium]
- [Codeforces Beta Round # 10 - task D "NCSV"](#) [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 - "BridgeArrangement 1000."](#) [Difficulty: High]

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:07  
EDIT: 11 Jun 2008 11:08

## K-th order statistic of O (N)

Suppose we are given an array A of length N and let the given number K. The challenge is to find in the array K-th largest number, ie K-th order statistic.

### Contents [hide]

- K-th order statistic of O (N)

The basic idea - the idea to use the quick sorting algorithm. Actually, the algorithm is simple, difficult to prove that it works on average for the O (N), as opposed to quick sort.

The 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], i.e. median among 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]);
    }

    // Insert the barrier
    a [l + 1] = a [j];
    a [j] = cur;

    // Continue to work in that part,
    // Which should contain the required element
    if (j >= k)
        r = j - 1;
    if (j <= k)
        l = i;

}

}
```

It should be noted that in the C ++ standard library, this algorithm has already been implemented - it is called `nth_element`.

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:08  
EDIT: 11 Jun 2008 11:08

## The dynamics of the profile. The problem of "parquet"

Typical problems of the dynamics of the profile are:

- Find the number of ways of tiling field some figures
- find a tiling with the fewest pieces
- find a paving with a minimum amount of unused cells
- find a paving with a minimum number of figures such that it is impossible to add one more piece

### Contents [hide]

- The dynamics of the profile. The problem of "parquet"
  - The problem of "Parquet"

## The problem of "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 be, the figures should not overlap).

We construct such dynamics: D [I] [Mask], where I = 1..N, Mask = 0..2 ^ M-1. I denotes the number of lines in the current field, and the 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 onto the "normal level", and if one - here "groove" depth 1. The response will obviously 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 make transitions ahead, ie add to it a new figure 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];

}
```

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:09  
EDIT: 24 Oct 2010 23:40

## Finding the most zero submatrix

Given matrix  $a$  size  $n \times m$ . You want to find in it a submatrix consisting only of zeros, and among all these - having the largest area (sub-matrix - the matrix is a rectangular area).

A trivial algorithm - fingering desired submatrix - even with the good implementation will work  $O(n^2m^2)$ . The following describes an algorithm that works for  $O(nm)$ , ie in linear time with respect to the size of the matrix.

### Contents [hide]

- Finding the most zero submatrix
  - Algorithm
    - Step 1: The auxiliary speaker
    - Step 2: Solution of the problem
  - Implementation

## Algorithm

To disambiguate immediately noticed that  $n$  equals the number of rows of the matrix  $a$ , respectively,  $m$ - the number of columns. Elements of the matrix will be numbered in 0-indeksatsii, ie in the designation  $a[i][j]$  codes  $i$  and  $j$  ranges run  $i = 0 \dots n - 1, j = 0 \dots m - 1$ .

### Step 1: The auxiliary speaker

First, calculate the following auxiliary dynamics:  $d[i][j]$  the closest unit to the top element  $a[i][j]$ . Formally speaking,  $d[i][j]$  equal to the largest line number (among lines range from  $-1$  to  $i$ ), which in  $j$ th column standing unit. In particular, if a line is not present,  $d[i][j]$  is assumed to be  $-1$  (it can be understood as that the whole matrix as limited outside units).

This momentum moving easily count the matrix from the top down: let us standing in  $i$ th row, and we know the value of the dynamics for the previous line. Then just copy these values into the dynamics of the current line, changing only those items which are in the matrix unit. It is clear that in this case do not even need to store all of the dynamics of a rectangular matrix, but rather 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;

    // вычислили d для i-ой строки, можем здесь использовать эти значения
}
```

### Step 2: Solution of the problem

Already, we can solve the problem for  $O(nm^2)$ - just to touch on the current line number of left and right columns of the required sub-matrix and using dynamics  $d[]$  to calculate for  $O(1)$  the upper limit of zero submatrix. However, you can go further and significantly improve the asymptotic behavior of the solutions.

It is clear that the desired zero submatrix bounded on all four sides by some Ones (or boundaries of the field) - which prevent it and increase in size and improve response. Therefore, it is argued, we will not miss the answer, if we act as follows: first brute over the number  $i$  of the bottom line of zero submatrices, then brute over which column  $j$  we will balk up zero submatrix. Using the value  $d[i][j]$ , we immediately obtain the number of the top line of zero submatrix. It remains now to determine the optimal left and right sides of zero submatrix - ie maximum push this submatrix left and right from  $j$ th column.

What does it mean to push the far left? Means to find an index  $k_1$  for which will  $d[i][k_1] > d[i][j]$ , while  $k_1$  the closest on the left to such an index  $j$ . It is clear that then  $k_1 + 1$  gives the number of the left column of

the desired zero submatrix. If such an index do not, then put  $k_1 = -1$ (this means that we can expand the current zero submatrix way to the left - to the boundary of the entire matrix  $a$ ).

Symmetrical index can be determined  $k_2$ for the right boundary: this is the closest to the right of  $j$ the index such that  $d[i][k_2] > d[i][j]$ (or  $m$ , if such an index is not present).

So, indexes  $k_1$ and  $k_2$ , if we learn to effectively look for them, give us all the necessary information about the current zero submatrix. In particular, it will be equal to the area  $(i - d[i][j]) \cdot (k_2 - k_1 - 1)$ .

How can find these codes  $k_1$ and  $k_2$ effective for fixed  $i$ and  $j$ ? We will satisfy only the asymptotic behavior  $O(1)$ , at least on average.

To achieve this, you can use the asymptotic behavior of the stack (stack) as follows. Learn how to search the index  $k_1$ , and store its value for each index  $j$ in the current row  $i$ in the dynamics  $d_1[i][j]$ . To do this, we will review all the columns  $j$ from left to right, and zavedëm a stack, which will always be based on only those columns in which the value of the dynamics  $d[]$ [more strictly  $d[i][j]$ ]. It is clear that the transition from the column  $j$ to the next column  $j + 1$ you want to update the contents of the stack. It is alleged that requires first put in a column stack  $j$ (a stack since "good"), and then, until the top of the stack is unsuited element (i.e. which value  $d \leq d[i][j + 1]$ ) - get the element. It is easy to understand that removed from the stack just enough from its top, and from any of his other places (because the stack will contain increases in  $d$ the sequence of columns).

The value  $d_1[i][j]$ for each  $j$ is the value that lies at this moment on top of the stack.

Clearly, since the addition of the stack on each line  $i$ going smoothly  $m$ pieces, then deletes also could not be more so in the asymptotic behavior of the sum will be linear.

Dynamics  $d_2[i][j]$ for finding indices  $k_2$ considered similar, but it is necessary to view columns from right to left.

Also note that this algorithm uses  $O(m)$ a memory (not counting the input data - the matrix  $a[]$ []).

## Implementation

This implementation of the above algorithm reads the size 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.

Easily improve this realization that it also outputs a zero submatrix itself: it is necessary at each change ans also remember the row and column of the submatrix (they will 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;
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:11  
EDIT: 2 May 2012 0:40

## Gauss solving a system of linear equations

Given a system  $n$  of linear algebraic equations (SLAE) with  $m$  unknowns. Is required to solve the system: to determine how it has solutions (none, one or infinite number of), and if it has at least one solution is found any of them.

**Formally, the problem is formulated as follows:**  
solve the system:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1m}x_m = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2m}x_m = b_2, \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nm}x_m = b_n, \end{cases}$$

where the coefficients

$a_{ij}$  ( $i = 1 \dots n, j = 1 \dots m$ ) and  $b_i$  ( $i = 1 \dots n$ ) are known as variables  $x_i$  ( $i = 1 \dots m$ ) - the unknown unknowns.

Convenient matrix representation of this problem:

$$Ax = b,$$

where  $A$  - the matrix  $n \times m$  of the coefficients  $a_{ij}$ ,  $x$  and  $b$  - the height of the column vectors  $m$

It is worth noting that UDUAL may be over the field of real numbers, and over the field **modulo** some number  $p$ , 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 is correct to call the method "Gauss-Jordan" (Gauss-Jordan elimination), because it is a variation of the Gauss method described surveyor William by Jordan in 1887 (it is worth noting that William Jordan is not the author of any of Jordan's theorem curves or Jordan algebra - all three different scientists namesake; moreover, appear to be more correct spelling is "Jordan", but writing "Jordan" is already fixed in Russian literature). It is also interesting to note that at the same time by Jordan (and

### Contents [hide]

- Gauss solving a system of linear equations
  - Gauss
    - The basic scheme
    - Search the support member (pivoting)
    - Degenerate cases
  - Implementation
  - Asymptotics
    - A more accurate estimate of the number of actions
  - Additions
    - Acceleration algorithm: its division into direct and reverse
    - Solving the linear modulo
    - A little bit about the different ways the choice of the support member
    - Improving found the answer
  - Literature

according to some sources even before him) came up with this algorithm class (B.-I. Clasen).

## The basic scheme

Briefly, the algorithm is a **sequential elimination** of the variables in each equation before until each equation will remain only one variable. If  $n = m$  it is possible to say that the Gauss-Jordan seeks to reduce the matrix  $A$  to the identity matrix system - because after the matrix has become a single solution of the system is obvious - the solution is unique and is set to receive coefficients  $b_i$ .

When this algorithm is based on two simple transformations equivalent system: First, the two equations can be exchanged, and secondly, any equation can be replaced by a linear combination of the line (with a nonzero coefficient), and the other rows (with arbitrary coefficients).

**In the first step** Gauss-Jordan algorithm divides the first line by a factor  $a_{11}$ . The algorithm then adds the first line to the other lines such factors to their coefficients in the first column appealed to zero - for that, obviously, the addition of the first line to  $i$ th need to multiply the it on  $-a_{i1}$ . Each matrix operations  $A$  (division by the number adding one line to the other) and the appropriate operations are carried out with the vector  $b$ ; in a sense, it behaves as if it were  $m + 1$  of th column of the matrix  $A$ .

As a result, after the first step, the first column of the matrix  $A$  becomes the unit (i.e., unit will contain in the first row and zeros in the other).

Similarly, the second step of the algorithm is performed, 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 zero the second column of the matrix  $A$ .

And so on, until we process all rows or all columns of the matrix  $A$ . If  $n = m$ , by the construction of the algorithm is obvious that the matrix  $A$  will turn the unit, which is what we needed.

## Search the support member (pivoting)

Of course, the above scheme is incomplete. It only works if at every  $i$ step th element  $a_{ii}$  is different from zero - otherwise, we simply can not achieve zero remaining coefficients in the current column by adding thereto  $i$ th row.

To make the algorithm works in such cases, there is just the process of **selection of the reference element** (in English it is called a word "pivoting"). It is made that the rows and / or columns of the matrix to the desired element  $a_{ii}$  turned nonzero number.

Note that a permutation of the rows is much easier implemented on a computer than a permutation of the columns: for the exchange of two places some columns have to remember that these two variables have exchanged places, and then, when you restore a response, right to recover what the answer to a variable is . When moving the lines, no such additional actions are necessary.

Fortunately for the correctness of the method alone is sufficient exchange lines (so-called "partial pivoting", as opposed to "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 of the support member 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 such **heuristics** : as a supporting member should be taken modulo largest element, with the support element to search and exchange with them must **always** and not just when it is needed ( i.e. not only when  $a_{ii} = 0$  ).

In other words, before executing  $i$ th phase Gauss-Jordan heuristics with partial pivoting to be

found in  $i$ th column among the elements with indices up to  $n$ the maximum modulus and the exchange line with this element with  $i$ th row.

Firstly, this heuristic will solve the linear algebraic equation, even if in the course of the decision will happen that element  $a_{ii} = 0$ . Second, which is very important, this heuristic improves the **numerical stability** of Gauss-Jordan.

Without this heuristic, even if the system is such that at each  $i$ phase of th  $a_{ii} \neq 0$ - Gauss-Jordan algorithm will work, but eventually accumulating error may be so great that even for matrices about 20the error will exceed the answer itself.

## Degenerate cases

So, if you stay on the Gauss-Jordan algorithm with partial pivoting, then, it is argued, if the  $m = n$ system is non-degenerate and (that has a nonzero determinant, which means that it has a unique solution), then the algorithm described above is fully mature and come to the unit matrix  $A$ (proof of this, ie the fact that a non-zero support element will always be, is not presented here).

We now consider the **general case** - where  $n$ and  $m$ are not necessarily equal. Suppose that the support element on the  $i$ th step is not found. This means that  $i$ all th column line, starting from the current to contain zeros. It is alleged that in this case the  $i$ Star variable can not be determined and is **an independent variable** (it may take an arbitrary value). To Gauss-Jordan continued his work for all subsequent variables in such a situation should just skip the current  $i$ th column, without increasing the number of the current line (we can say that we remove virtually  $i$ th column of the matrix).

So, some of the variables in the process of the algorithm may be provided independent. It is understood that when the number of  $m$ variables is greater than number  $n$ of equations, then at least the  $m - n$ independent variables are detected.

In general, if detected at least one independent variable, it can take an arbitrary value, while the other (dependent) variables will be expressed through it. This means that when we work in the field of real numbers, the system has a potentially **infinite number of solutions** (if we consider the linear algebraic equation modulo the number of solutions is equal to the power of this module is the number of independent variables). However, you should be careful: we must remember that even if the independent variables were found, however the linear algebraic equation **may have no solutions at all**. This happens when the remaining untreated equations (those to which the algorithm of Gauss-Jordan is not reached, that is equations in which there were only independent variables) there is at least one non-zero constant term.

However, it is easier to check the apparent substitution of the solution found: all independent variables assigned zero values, the dependent variables to assign values found, and substitute this solution into current Slough.

## Implementation

We give here the implementation of Gauss-Jordan heuristics with partial pivoting (choice of the support member as a maximum by column).

The input to the function `gauss()`is passed to the system matrix itself  $a$ . The last column of the matrix  $a$ - this is our old notation column  $b$ free coefficients (as done for the convenience of programming - as in the algorithm, all operations with the free coefficients  $b$ repeat the operation with the matrix  $A$ ).

The function returns the number of solutions of the system ( 0, 1, or  $\infty$ ) (infinity designated

special constant in the code **INF**, which you can set any great value). 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]][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 are supported by two pointers - for the current column **col** and the current line **row**.

Also, plant vector **where** in which each variable is written, in which line should it happen (in other words, for each column, write the number of the line where this column is nonzero). This vector is needed because some variables could not "decide" in the solution (ie, are independent variables, which can be assigned an arbitrary value - for example, in the reduced implementation is zeros).

The implementation uses the technique of partial pivoting, by searching the string with a maximum modulo element, and then rearranging the row position **row** (although obvious

permutation of rows can be replaced by the exchange of the two indexes in an array, in practice, it does not give a real win, because on exchanges spent  $O(n^2)$  operations).

In order to ease the implementation of the current line is not divisible by the support element - so that the end result 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 verification is successful solutions found, then the function returns 1, or  $\infty$  - depending on whether there is at least one independent variable, or not.

## Asymptotics

We estimate the asymptotic behavior of the resulting algorithm. The algorithm consists of  $m$  phases, in each of which there is:

- Search and rearrangement of the support element - a time  $O(n + m)$  when using heuristics "partial pivoting" (search for the maximum 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 item is less asymptotics than the second. Note also that the second item is executed no more than  $\min(n, m)$  once - as much as can be dependent variables in Slough.

Thus, the final asymptotic behavior of the algorithm takes the form  $O(\min(n, m) \cdot nm)$ .

With  $n = m$  this estimate turns into  $O(n^3)$ .

Note that when the linear algebraic equation is not seen in the field of real numbers, and in the modulo two, the system can be solved much faster - on this, see. Below in the section "for solving the linear modulo".

## A more accurate estimate of the number of actions

For simplicity, 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 this equation from the rest.

This can occur at each of the  $n$  steps, and the current equation is added to all  $n - 1$  the others. When adding work comes only with the columns from the current. Thus, the sum obtained in  $n^3/2$  operations.

## Additions

### Acceleration algorithm: its division into direct and reverse

To double its acceleration algorithm can consider another version of it, a classic, when the algorithm is divided into phases of 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 immediately

value of the last variable, then the obtained value is substituted in the penultimate equation and is the penultimate value of the variable, and so on. This process is called **reverse motion Gauss**.

**Direct flow of Gauss** - is an algorithm similar to the algorithm described above Gauss-Jordan, with one exception: the current variable is not excluded from all equations, and only after the current equations. As a result, it does not obtain the diagonal, and a triangular matrix.

The difference is that the direct flow of runs **faster** Gauss-Jordan - because on average it makes half the additions of one equation to another. Reverse works for  $O(nm)$ , in any case, asymptotically faster than the forward stroke.

Thus, if  $n = m$ , then the algorithm will be delivered by  $n^3/4$  operations - that is half Gauss-Jordan.

## Solving the linear modulo

For solving the linear module can be used for the algorithm described above, it retains its validity.

Of course, now it becomes unnecessary to use some clever technology of choice supporting member - it suffices to find any non-zero element in the current column.

If the module is a simple, no complications do not arise - occurring in the course of the algorithm of Gauss division does not create any problems.

Especially remarkable **unit equal to two** : for it all matrix operations can be performed very efficiently. For example, the subtraction of one line from another modulo two - it's actually their symmetric difference ("xor"). Thus, the whole algorithm can be greatly accelerated by squeezing the entire matrix in terms bitmaps and only them. We give here the main part of the implementation of a new algorithm, Gauss-Jordan, using a 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 can be seen, the implementation has become even a little shorter, despite the fact that it is much faster than the old implementation - namely, faster 32times at the expense of a bit of compression. It should also be noted that the solution is modulo two systems in practice is very fast, because the cases when from one line must take another, occur infrequently (for sparse matrices, this algorithm can work more time for the size of the order of the square than a cube).

If the module is **an arbitrary** (not necessarily simple), everything becomes more complicated. It is clear that using [the Chinese remainder theorem](#), we reduce the problem with an arbitrary module only to the modules of the form "prime power." [The following text has been hidden since this is anecdotal - perhaps the wrong way to solve]

Finally, we consider the question **of solving the linear modulo**. The answer is quite simple: the number of solutions is equal to  $p^k$ , where  $p$ - module,  $k$ - the number of independent variables.

## A little bit about the different ways the choice of the support member

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. Also, it turns out that it gives almost the same result as the "full pivoting" - when the support member is sought among the elements of the whole submatrix - starting with the current row with the current column.

But it is interesting to note that both of these heuristics with finding the maximum element, in fact, are very dependent on how much was scaled the original equations. For example, if one of the systems of equations multiplied by a million, then this equation is almost certain to be selected as the lead on the first step. It seems rather strange, so logical transition to a little more complex heuristics - the 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 (or each row of support so that it was at a maximum is equal to unity in modulus, but this may lead to an increase in the accumulated error).

## Improving found the answer

Because, despite the various heuristics, Gauss-Jordan algorithm can still lead to large errors in the matrix even special order sizes 50-100.

In this regard, the resulting algorithm Gauss-Jordan response can be improved by applying to it a simple numerical method - for example, the method of simple iteration.

Thus, the solution turns into a two-step: First, a Gauss-Jordan, and then - a numerical method that takes as initial solution obtained in the first step.

This technique allows multiple extend the set of problems solved by Gauss-Jordan with acceptable accuracy.

## 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]

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Posted: 5 Jul 2008 21:03  
 EDIT: 5 Jul 2008 23:54

## Finding the rank of the matrix

**Rank** - this is the largest number of linearly independent rows / columns. Rank is defined not only for square matrices; Suppose that the matrix is rectangular and has a size of NxM.

Also, the rank matrix can be defined as the highest order of the minors nonzero.

Note that if the square matrix and its determinant is nonzero, then the rank is equal to N (= M), otherwise it will be less. In general, the rank of the matrix is not greater than min (N, M).

## Algorithm

Search rank by using the modified [method of Gauss](#). Will perform exactly the same operations as in the solution of the system or find its determinant, but if at any step in the i-th column among unselected before that there are no nonzero rows, we skip this step, and the rank is decremented ( Rank initially set equal to max (N, M)). Otherwise, if we have found on the i-th step of the line with a non-zero element in the i-th column, then mark the row as selected, and perform common operations grabbing 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];
    }
}
```

### Contents [\[hide\]](#)

- Finding the rank of the matrix
  - Algorithm
  - Implementation

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: 5 Jul 2008 22:14  
EDIT: 6 Oct 2010 0:22

## Calculating the determinant of a matrix by Gauss

Given a square matrix A of size NxN.  
Required to compute its determinant.

### Contents [hide]

- Calculating the determinant of a matrix by Gauss
- Algorithm
- Implementation

## Algorithm

Use the ideas of Gauss method for solving systems of linear equations .

Will perform the same procedure as in solving systems of linear equations, excluding only the division of the current line on a [i] [i] (more precisely, the division itself can be carried out, but assuming that the number shall be made outside the determinant). Then all the operations that we will produce a matrix will not change the value of the determinant of the matrix, with the exception, perhaps, of the sign (we just interchanged the two lines, which changes the sign, or we add one line to another, which does not change the value determinant).

But the matrix to which we arrive after Gauss, is diagonal, and its determinant is equal to 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 must be reversed). Thus, we can use Gauss algorithm to compute the determinant of the matrix of O ( $N^3$  ).

It remains only to note that if at some point we do not find in the current column nonzero element, the algorithm should stop and return 0.

## Implementation

```
const double EPS = 1E-9;
int n;
vector <vector <double>> a (n, vector <double> (n));
..., And reading a ... n

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;
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:13  
EDIT: 15 Jul 2014 17:13

## Integration of the formula Simpson

Required to calculate the value of the definite integral:

$$\int_a^b f(x)dx$$

### Contents [hide]

- Integration of the formula Simpson
  - Simpson's formula
  - Error
  - Implementation

The solution described here, was published in one of theses **Thomas Simpson** (Thomas Simpson) in 1743

## Simpson's formula

Let  $n$ - be a natural number. We divide the interval of integration  $[a; b]$  on  $2n$  equal parts:

$$x_i = a + ih, \quad i = 0 \dots 2n,$$

$$h = \frac{b - a}{2n}.$$

Now calculate the integral separately for 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 three points  $(x_{2i-2}, x_{2i-1}, x_{2i})$ . This parabola always exists and is unique. It can be found analytically, then there will be only integrated 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 **formula of Simpson** :

$$\int_a^b f(x)dx = (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \dots + 4f(x_{2N-1}) + f(x_{2N})) \frac{h}{3}.$$

## Error

The error given by Eq Simpson, do not exceed the value of:

$$\frac{1}{180} h^4 (b - a) \max_{a \leq x \leq b} |f^{(4)}(x)|.$$

Thus, the error is of the order of reduction both  $O(n^4)$ .

## Implementation

Here  $f(x)$ - some user-defined function.

```
double a, b; // входные данные
const int N = 1000*1000; // количество шагов (уже умноженное на 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;
```

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 11 Jun 2008 11:14  
 Edited: 21 Sep 2010 0:54

## Newton's method (tangent) to search for the roots

This iterative method, invented by **Isaac Newton** (Isaak Newton) about 1664, however, this method is sometimes referred to by the Newton-Raphson (Raphson), as invented Raphson algorithm is the same 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, to find one of its roots (assuming the root exists). It is assumed that  $f(x)$  continuous and differentiable on the interval  $[a; b]$ .

### Contents [hide]

- Newton's method (tangent) to search for the roots
  - Algorithm
  - Application to calculate the square root

## Algorithm

The input parameters of the algorithm, besides the function  $f(x)$ , is also a **first approximation** - some  $x_0$  from which the algorithm starts to go.

Suppose we have calculated  $x_i$ , we calculate  $x_{i+1}$  as follows. Draw a tangent to the graph of the function  $f(x)$  at the point  $x = x_i$ , and find the point of intersection of this tangent with the x-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:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

It is intuitively clear that if the function  $f(x)$  is enough "good" (smooth), and  $x_i$  is close enough to the root, it  $x_{i+1}$  will be closer to the desired 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 of calculating the square root.

If we substitute  $f(x) = \sqrt{x}$ , after simplify the expression we 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 it is necessary 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 ("% .15lf", x);

```

Another common variant of the problem - when you want to find the root of an integer (for this  $n$  to find the greatest  $x$  such that  $x^2 \leq n$ ). Here we have a little change stop condition of the algorithm, because it can happen that  $x$  will start "jumping" next response. Therefore, we add a condition that if the value  $x$  has decreased from the previous step and the current step is trying to rise, the flow to stop.

```

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 present is a third option - in the case of long arithmetic. Since the number  $n$  may be quite large, it makes sense to pay attention to the initial guess. Obviously, the more it is closer to the root, the faster the results. Quite simple and effective to take as the initial approximation of  $2^{\text{bits}/2}$  where  $\text{bits}$  - the number of bits in the number  $n$ . Here is the code in the language of Java, showing this option:

```

BigInteger n; // входные данные

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 version of the code is executed for the number  $10^{1000}$  of 60 milliseconds, and if you remove an improved choice of the initial approximation (just to start with 1), it will run about 120 milliseconds.

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 23 Jul 2009 12:53

EDIT: 23 Jul 2009 12:53

## Ternary search

### Statement of the Problem

Given a function  $f(x)$ , **unimodal**

on some interval  $[l; r]$ . By unimodal is meant one of two options. First, strictly increasing function first, then reaches a maximum (at one point or the whole segment), 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 completely symmetrical him.

Required **to find the maximum of the function  $f(x)$  on the interval  $[l; r]$ .**

### Contents [hide]

- Ternary search
  - Statement of the Problem
  - Algorithm
    - The case of the integer argument
  - Implementation

### Algorithm

Take any two points  $m_1$ , and  $m_2$  in this segment:  $l < m_1 < m_2 < r$ . Calculate the values of the function  $f(m_1)$  and  $f(m_2)$ . Then we get three options:

- If you find that  $f(m_1) < f(m_2)$ , the desired maximum can not be in the left-hand side, ie, in part  $[l; m_1]$ . This is easily seen that if the left point of the function is smaller than the right, then either of these two points are in the area of "lifting" function, or only the left point is there. In any case, this means that the peak is meaningful to search on only the segment  $[m_1; r]$ .
- If, on the contrary,  $f(m_1) > f(m_2)$  then the situation is similar to the previous, up to symmetry. Now the desired maximum can not be on the right side, i.e. in part  $[m_2; r]$ , why go to the segment  $[l; m_2]$ .
- If  $f(m_1) = f(m_2)$ , then either both of these points are in the region of the maximum, or the left point is in the ascending and the right - in descending order (here essentially used that increase / decrease strict). Thus, after the search is meaningful in the segment  $[m_1; m_2]$ , but (in order to simplify the code), this case can be attributed to any of the previous two.

Thus, according to the comparison result of the function in two interior points instead of the current segment, we find  $[l; r]$  there is a new segment  $[l'; r']$ . We now repeat all the steps for this new segment, we again obtain a new, strictly smaller segment, etc.

Sooner or later the length of the segment will be a little lower than the pre-defined constants, accuracy, and the process can be stopped. This method is numerical, so after stopping the algorithm can approximately assume that all points of the interval  $[l; r]$  reaches its maximum; as a response can take, for example, a point  $l$ .

It remains to note that we do not impose any restrictions on the choice of points  $m_1$  and  $m_2$ . From this process, clearly, will depend on the rate of convergence (and error occurs). The most common way - choose a point so that the segment  $[l; r]$  was divided them into 3 equal parts:

$$m_1 = l + \frac{r - l}{3}$$

$$m_2 = r - \frac{r - l}{3}$$

However, a different choice of when  $m_1$  and  $m_2$  closer to each other, the convergence rate of increase slightly.

## The case of the integer argument

If the argument to  $f$  an integer, the segment  $[l; r]$  also becomes discrete, however, because we do not impose any restrictions on the choice of points  $m_1$  and  $m_2$  then on the correctness of the algorithm is not affected. Can still choose  $m_1$  and  $m_2$  so that they divide the segment  $[l; r]$  into 3 parts, but only approximately equal.

Wherein the second point - the stopping criterion of the algorithm. In this case, ternary search will have to stop when it will be  $r - l < 3$ , because in this case is no longer possible to select a point  $m_1$  and  $m_2$  so varied and different from  $l$ , and  $r$ , and this can lead to infinite loops. After ternary search algorithm will stop and  $r - l < 3$ , of the remaining number of candidate points  $(l, l + 1, \dots, r)$  is necessary to select a point with the maximum value of the function.

## Implementation

The implementation for the continuous case (ie, the function  $f$  has the form: double  $f$  (double  $x$ ):

```
double l = ..., r = ..., EPS = ...; // входные данные
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;
}
```

Here EPS- in fact, **the absolute error** response (not counting errors due to inaccurate calculation of the function).

Instead of the criterion "while ( $r - l > EPS$ )" can be selected and a stopping criterion:

```
for (int it=0; it<iterations; ++it)
```

On the one hand, it is necessary to choose a constant **iterations** to ensure the required accuracy (typically just a few hundred, to achieve maximum accuracy). But, on the other hand, the number of iterations stops depend on the absolute values  $|l|$  and  $r$ , i.e. we are actually using **iterations** Asking desired **relative error**

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:15  
Edited: 17 Sep 2010 22:19

## Binomial coefficients

Binomial coefficient  $C_n^k$  is the number of ways to select a set of  $k$  objects of  $n$  different subjects without regard to the order of these elements (ie, the number of unordered sets).

Also binomial coefficients - it koffsienty in the expansion  $(a + b)^n$  (the so-called 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 as a triangle, to effectively find the coefficients opened Blaise Pascal (Blaise Pascal), who lived in the 17th century. Nevertheless, she was known more Chinese mathematician Ian Hui (Yang Hui), who lived in the 13th century. Perhaps it opened a Persian scholar Omar Khayyam (Omar Khayyam). Moreover, the Indian mathematician Pingala (Pingala), who lived more in the 3rd. BC, got similar results. Newton same merit is that he generalized this formula for degrees that are not natural.

### Contents [hide]

- Binomial coefficients
  - Calculation
  - Properties
  - Calculations in the program
    - Direct calculations in analytical formula
    - Improved implementation
    - Pascal's triangle
    - The calculation of O (1)

## Calculation

An analytic formula for the calculation:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

This formula is easily derived from the problem of disordered sample (number of ways to randomly select  $k$  items from the  $n$  elements). First, count the number of ordered samples. Select the first element is a  $n$  way, the second -  $n - 1$  third -  $n - 2$  and so on. As a result, the number of ordered samples we obtain the formula:

$n(n - 1)(n - 2) \dots (n - k + 1) = \frac{n!}{(n-k)!}$ . To disordered samples easily go, if we observe that each disordered sample corresponds exactly  $k!$  ordered (as is the number of all possible permutations of  $k$  elements). As a result, sharing  $\frac{n!}{(n-k)!}$  on  $k!$ , we obtain the required 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 through the previous formula.

It is worth noting especially when  $n < k$  the value  $C_n^k$  is always assumed to be zero.

## Properties

Binomial coefficients have many different properties, we present the simplest of them:

- The 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  $k$ :

$$\sum_{k=0}^m C_{n+k}^k = C_{n+m+1}^m$$

- Summing the squares:

$$(C_n^0)^2 + (C_n^1)^2 + \dots + (C_n^n)^2 = C_{2n}^n$$

- Weighted summation:

$$1C_n^1 + 2C_n^2 + \dots + nC_n^n = n2^{n-1}$$

- Telecommunication with [Fibonacci numbers](#) :

$$C_n^0 + C_{n-1}^1 + \dots + C_{n-k}^k + \dots + C_0^n = F_{n+1}$$

## Calculations in the program

### Direct calculations in analytical formula

Calculating the first, immediate formula very easy to program, but this method is likely to overflow even at relatively small values  $n$ , and  $k$  (even if the answer is completely placed in any type of data, calculating factorials intermediates may lead to overflow). Therefore, very often this method can only be used with the [[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;
}
```

## Improved implementation

It can be noted that in the above implementation of the numerator and the denominator is the same number of factors ( $k$ ), each of which is less than unity. Therefore, we can replace our fraction on the product  $k$  fractions, each of which is a real-valued. However, you will notice that after multiplying the current response at each regular shot will still be given an integer (this, for example, follows from the properties of "making-rendering"). Thus, we obtain a 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 the fractional number carefully to the whole, taking into account that due to accumulated errors it may be slightly less than the true value (for example, 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 from it take the result. The advantage of this method is that intermediate results never exceed the response calculation for each new table element need only one addition. The downside is slow work for large  $N$ , and  $K$ , if in fact the table is not necessary, but you need a single value (because to calculate the  $C_n^k$  need to build a table for all  $C_i^j$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq n$ , or at least before  $1 \leq j \leq \min(i, 2k)$ ).

```
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 (current -  $n$ -th row and the previous -  $n - 1$ -s).

## The calculation of O (1)

Finally, in some situations is beneficial predposchitat advance the values of all the factorials in order to subsequently consider any necessary binomial coefficient, producing only two divisions. This can be beneficial when using [Long arithmetic](#) when the memory does not allow predposchitat all Pascal's triangle, or when you want to make some calculations on a simple module (if it is not simple, then there are difficulties in dividing the numerator by the denominator and can be overcome if factoring module and store all the numbers in the form of vectors of the degrees of these simple, see [the section "Long arithmetic in factored form"](#) ).

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 11 Jun 2008 11:16  
EDIT: 2 May 2009 17:24

## Catalan numbers

Catalan numbers - numerical sequence, which is found in a surprising number of combinatorial problems.

This sequence is named after the Belgian mathematician Catalana (Catalan), who lived in the 19th century, when in fact she was known to Euler (Euler), who lived a century before Catalana.

### Contents [hide]

- Catalan numbers
  - Sequence
  - Calculation
    - Recursion formula
    - An analytic formula

## Sequence

The first few Catalan numbers  $C_n$  (starting from zero):

1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

Catalan numbers are found in a large number of problems in combinatorics.  $n$  retracement number of Catalan - is:

- Number of correct bracket sequence consisting of  $n$  opening and  $n$  closing brackets.
- The number of root binary trees with  $n + 1$  leaves (vertices are not numbered).
- The number of ways to completely separate brackets  $n + 1$  factor.
- The number of triangulations of a convex  $n + 2$ -gon (ie the number of partitions of a polygon into triangles by nonintersecting diagonals).
- The number of ways to connect the  $2n$  points on a circle  $n$  disjoint chords.
- The number of nonisomorphic full binary trees with  $n$  internal nodes (ie, having at least one son).
- Number of monotone paths from point  $(0, 0)$  to point  $(n, n)$  in a square lattice of size  $n \times n$ , does not rise above the main diagonal.
- Length number of permutations  $n$  that can be sorted stack (it can be shown that the rearrangement is a sorted stack and then only if there is no such index  $i < j < k$ , that  $a_k < a_i < a_j$ ).
- The number of partitions of a set of continuous  $n$  elements (ie partitions on contiguous blocks).
- The number of ways to cover the ladder  $1 \dots n$  using  $n$  rectangles (referring to the figure, consisting of  $n$  columns  $i$ th of which has a height  $i$ ).

## Calculation

There are two formulas for the Catalan numbers: recurrent and analytical. Because we believe that all the above cited problems are equivalent, to prove the formulas we will choose the task with which to do it the easiest way.

### Recursion 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 sequences.

The leftmost opening parenthesis l corresponds to certain closing bracket r, that breaks the formula two parts, each of which in turn is the correct bracket sequence. Therefore, if we denote  $k = r - l - 1$ , for any fixed  $r$  will be exactly  $C_k C_{n-1-k}$  means. Summing this over all admissible  $k$ , we obtain a recurrence relation for  $C_n$ .

### An analytic formula

$$C_n = \frac{1}{n+1} C_{2n}^n$$

(Here  $C_n^k$  designated as usual binomial coefficient ).

This formula is the easiest way to withdraw from the problem of monotone paths. Total number of monotone paths in the lattice size  $n \times n$  is equal  $C_{2n}^n$ . Now count the number of monotone paths crossing diagonal. Consider some of these ways, and find the first edge, which is above the diagonal. Reflect about the diagonal all the way, going after this edge. As a result, we obtain a monotone path in the lattice  $(n-1) \times (n+1)$ . On the other hand, any monotonic path in the lattice  $(n-1) \times (n+1)$  diagonal crosses necessarily hence it just received in this way from any (and only) the monotonic path intersecting diagonal in the grid  $n \times n$ . Monotone paths in the lattice  $(n-1) \times (n+1)$  there  $C_{2n}^{n-1}$ . As a result, we obtain the formula:

$$C_n = C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$$

# MAXimal

home

algo

bookz

forum

about

Added: 11 Jun 2008 11:17  
edited 1 Jun 2009 11:36

## Necklaces

The problem of "necklaces" - is one of the classical combinatorial problems. Calculate the required amount of the various necklaces  $n$  beads, each of which can be colored in one  $k$  color. When comparing two necklaces can be rotated, but not to turn (ie, allowed to make a cyclic shift).

### Contents [hide]

- Necklaces
  - Solution

## Solution

You can solve this problem by using [Lemma Burnside and Polya theorem](#). [Here comes 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:

$$\pi_0 = 1 \ 2 \ 3 \ \dots \ n$$

$$\pi_1 = 2 \ 3 \ \dots \ n \ 1$$

$$\pi_2 = 3 \ \dots \ n \ 1 \ 2$$

$$\dots$$

$$\pi_{n-1} = n \ 1 \ 2 \ \dots \ (n-1)$$

Let us find an explicit formula for the calculation  $C(\pi_i)$ . First, we note that the permutation looks like that, that  $i$ th permutation on  $j$ th position is  $i + j$  (modulo  $n$ , if greater  $n$ ). If we consider the cyclic structure  $i$ th permutation, we can see that the unit enters  $1 + i, 1 + 2i, 1 + 3i$  and so on, until we arrive at the number  $1 + kn$ ; for the other elements are made similar statements. From this we can understand that all cycles have the same length, equal  $\text{lcm}(i, n)/i$ , ie  $n/\gcd(i, n)$  ("gcd" - the greatest common divisor, "lcm" - the least common multiple). Then the number of cycles  $i$  equal to the  $i$ th permutation will simply  $\gcd(i, n)$ .

Substituting these values in the Polya theorem, we obtain the **solution**:

$$\text{Ans} = \frac{1}{n} \sum_{i=1}^n k^{\gcd(i, n)}$$

You can leave a formula in this form, but you can minimize it even more. Let's move on from the sum of all  $i$  the sum of only divisors  $n$ . Indeed, in our sum will be much the same terms: if  $i$  not a divider  $n$ , such that there exists a divisor after

calculation  $\gcd(i, n)$ . Consequently, for each divider  $d|n$  term it  $k^{\gcd(d,n)} = k^d$  will take into account several times, ie, amount can be written in the form:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} C_d k^d$$

where  $C_d$ - is the number of such numbers  $i$  that  $\gcd(i, n) = d$ . Let us find an explicit expression for this quantity. Any such number  $i$  is given by:  $i = dj$  where  $\gcd(j, n/d) = 1$ (otherwise it would be  $\gcd(i, n) > d$ ). Remembering the Euler 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 the formula :

$$\text{Ans} = \frac{1}{n} \sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:18  
EDIT: 11 Jun 2008 11:18

## The alignment of elephants on a chessboard

Required to find the number of ways to place K elephants on the board size NxN.

### Contents [hide]

- The alignment of elephants on a chessboard
  - Algorithm
  - Implementation

## Algorithm

We 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, with only the diagonals that of the same color as the i-th diagonal. Then,  $i = 1..2N-1$ ,  $j = 0..K$ .

Diagonal enumerate as follows (example for board 5x5):

```
black: white:
1 _ 5 _ 9 _ 2 _ 6 _
 5 _ 9 _ 2 _ 6 _ 8
 5 _ 9 _ 7 _ 6 _ 8 _
 9 _ 7 _ 6 _ 8 _ 4
 9 _ 7 _ 3 _ 8 _ 4 _
```

Ie odd numbers correspond to black diagonals, the even - white; diagonal enumerate in order to increase the number of elements in them.

With such a numbering we can calculate each  $D[i][j]$ , based only on  $D[i-2][j]$  (deuce deducted that we view the diagonal of the same color).

So, let the current dynamic element -  $D[i][j]$ . We have two transitions. First -  $D[i-2][j]$ , ie put all j elephants on previous diagonal. The second transition - if we put a bishop on the current diagonal, and the remaining elephants  $j-1$  - previous; we note that the number of ways to put an elephant on the current diagonal equal to the number of cells in it minus  $j-1$ , because elephants standing in the previous diagonals will overlap part of the directions. Thus, 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) / 2 * 4 + 2;
}
```

It remains to determine the dynamics of the base, there is no difficulty:  $D[i][0] = 1$ ,  $D[1][1] = 1$ .

Finally, evaluating the dynamics, to find any **answer** to the problem is simple. Iterate number  $i = 0..K$  elephants standing on black diagonals (number of the last black diagonal -  $2N-1$ ), respectively  $K_i$  elephants put on white diagonal (number of the last white diagonal -  $2N-2$ ), ie, 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;
```

# MAXimal

home  
algorithms  
bookz  
forum  
about

Added: 11 Jul 2008 20:45  
EDIT: 18 Feb 2013 16:24

## Correct bracket sequence

Correct bracket sequence is a string consisting only of characters "brackets" (often considered only parentheses, but this will be considered the general case of several types of brackets), where each closing brace exists a corresponding opening (and of the same type).

Here we consider the classic problem parenthesis on the right sequence (for brevity, simply "sequence") to check the accuracy, the number of sequences, all sequences generated by finding the lexicographically next sequence determination  $k$ -th sequence in the sorted list of all sequences, and, conversely, determination sequence number. Each task is considered in two cases - when the brackets are allowed only one type, and when several types.

### Contents [hide]

- Correct bracket sequence
  - Check for correct
  - Number of sequences
    - Formula
    - Dynamic programming
  - Finding all sequences
  - Finding the next sequence
  - Sequence number
  - Finding  $k$ -th sequence

### Check for correct

Suppose first parenthesis are allowed only one type, then check the correctness of the sequence can be a very simple algorithm. Let  $\text{depth}$  - is the current number of open parentheses. Originally  $\text{depth} = 0$ . Will move from left to right on the line if the current opening bracket, the increase  $\text{depth}$  in the unit, otherwise reduced. If this when it turns negative, or at the end of the algorithm  $\text{depth}$  is different from zero, then this string is not correct bracket sequence is different.

If several types of brackets are valid, then the algorithm should be changed. Instead, the counter  $\text{depth}$  should create a stack, which will put the parentheses as they become available. If the current character string - opening parenthesis, then we place it on the stack, and if the closing - then check that the stack is not empty, and that on the top bracket is the same type as the current one, and then We reach this bracket from the stack. If any of the conditions are not fulfilled, or at the end of the algorithm the stack was not empty, then the sequence is not the correct bracket, otherwise is.

Thus, both of these problems, we learned to solve in time  $O(n)$ .

### Number of sequences

#### Formula

Number of correct bracket sequences with one type of brackets can be calculated as [the number of Catalan](#). Ie if there are  $n$  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  $k$  types of brackets. Then each pair of parentheses independently of the others can take one of the  $k$  types, so we get 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 point of view of [dynamic programming](#). Let  $d[n]$  - the number of correct bracket sequences of  $n$  pairs of brackets. Note that in the first position will always stand opening bracket. It is clear that within this pair of brackets is some kind of regular bracket sequence; Similarly, after the pair of brackets is also worth a correct bracket sequence. Now to calculate  $d[n]$ , brute over how many pairs of brackets  $j$  will be inside this first pair, then, respectively,  $n - 1 - j$  a pair of brackets will stand after the first pair. Therefore, the formula for  $d[n]$  the form:

$$d[n] = \sum_{i=0}^{n-1} d[i] \cdot d[n-1-i].$$

The initial value of the recursion formula - it  $d[0] = 1$ .

## Finding all sequences

Sometimes you need to find and display all the correct bracket sequence specified length  $n$  (in this case  $n$ - is the length of the string).

To do this, you can start with the lexicographically first sequence  $((\dots((\dots)))$ , and then find each time lexicographically following sequence using the algorithm described in the next section.

But if the restrictions are not very large ( $n$  up to  $10 - 12$ ), you can do much easier. Let us find all the possible permutations of these brackets (for this it is convenient to use the `next_permutation()`), they will  $C_{2n}^n$ , and every check on the correctness of the above algorithm, and if correct, shows the current sequence.

Also, the process of finding all sequences can be issued in the form of a recursive enumeration with clipping (which ideally can be brought to speed up the first algorithm).

## Finding the next sequence

Here we consider only the case of one type of brackets.

For a given proper bracketing sequence is required to find the correct sequence of the bracket, which is located next to the lexicographical order after the current (or issue "No solution", if it does not exist).

It is clear that the whole algorithm is as follows: find a rightmost open parenthesis that we have the right to replace the covering (so that at this point is correct not broken), and the rest of the right to replace a string lexicographically minimum: ie as a opening parenthesis, then all remaining closing brackets. In other words, we try to remain unchanged as the longest prefix of the original sequence, and this sequence in the suffix is replaced by the lexicographically minimum.

It remains to learn to look for that same position of the first changes. To do this, we will follow the line from right to left and to maintain a balance `depth` of open and closed brackets (at the meeting opening bracket will decrease `depth`, and at closing - increase). If at any point we are on the opening parenthesis, and the balance after treatment with this symbol is greater than zero, then we have found the right-most position from which we can begin to change the sequence (in fact,  $\text{depth} > 0$  means that the left has not yet closed parenthesis). Put the current position closing parenthesis, then the maximum possible number of opening brackets, and then all the remaining closing brackets - the answer is found.

If we watched an entire row and have not found a suitable position, the current sequence - the maximum, and there are no answers.

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 brackets in the sequence. Need to ask the right bracketing sequence to find its number in the list of ordered lexicographically correct bracket sequences.

Learn to consider supporting **the dynamics**  $d[i][j]$ , where  $i$ - the length of the bracket sequence (it is "semi-regular": any closing bracket features a steam room open, but not all the open brackets are closed),  $j$ - the balance (ie the difference between the number of opening and closing brackets)  $d[i][j]$ - the number of such sequences. When calculating these dynamics, we believe that the brackets are only one type.

Consider this dynamics can be as follows. Let  $d[i][j]$ - the value that we want to count. If  $i = 0$  the answer is clear right away:  $d[0][0] = 1$  everyone else  $d[0][j] = 0$ . Now suppose that  $i > 0$ , while mentally brute over what was equal to the last character of the sequence. If it is equal to '(', until this symbol we are able to  $(i - 1, j - 1)$ . If it is equal to ')', the previous state was  $(i - 1, j + 1)$ . Thus, we obtain:

$$d[i][j] = d[i - 1][j - 1] + d[i - 1][j + 1]$$

(Assuming that all values  $d[i][j]$  at negative  $j$ ). Thus, this dynamic, we can calculate for  $O(n^2)$ .

Let us now turn to the solution of the problem.

First let only allowed braces **one** type. Zavedäm counter **depthnesting** in parentheses, and will move in sequence from left to right. If the current symbol  $s[i]$  ( $i = 0 \dots 2n - 1$ ) is '(' we increase **depth** by 1 and go to the next character. If the current character is ')', then we must add to the response  $d[2n - i - 1][\text{depth} + 1]$ , thereby taking into account that this position could be the symbol ' ('(which would have led to the lexicographically less consistent than current), then we reduce **depth** by one.

Suppose now allowed parentheses **several**  $k$  types. Then when considering the current character  $s[i]$  to the conversion **depth** we need to go through all the brackets that are less than the current character, try to put this bracket at the current position (thereby obtaining a new balance **ndepth** = **depth** ± 1), and to add to the answer and the number corresponding to the "tails" - completions (which have a length  $2n - i - 1$ , balance **ndepth** and  $k$  types of brackets). It is alleged that the formula for this quantity has the form:

$$d[2n - i - 1][\text{ndepth}] \cdot k^{(2n - i - 1 - \text{ndepth})/2}.$$

This formula is derived from the following considerations. First, we "forget" about the fact that there are several types of braces, and just take the answer from  $d[2n - i - 1][\text{ndepth}]$ . Now calculate how to change the response of the presence of  $k$  types of brackets. We have an  $2n - i - 1$  undefined positions, of which **ndepth** are staples closing some of the previously discovered - hence, the type of braces, we can not be varied. But all the rest of the brackets (and they will  $(2n - i - 1 - \text{ndepth})/2$ steam) can be of any  $k$  type, so the answer is multiplied by the power of this  $k$ .

## Finding $k$ th sequence

Here let  $n$ - the number of pairs of brackets in the sequence. In this problem on the set  $k$  you want to search  $k$ -s correct bracket sequence in the list lexicographically ordered sequences.

As in the previous section, we can calculate **the dynamics**  $d[i][j]$ - the number of correct bracket sequences of length  $i$  with the balance  $j$ .

Suppose first that allowed only brackets **one** type.

We move on the characters of the string, with 0th at  $2n - 1$ th. As in the previous problem, let us have a counter **depth**- current depth of nesting in parentheses. Each current position will be possible to sort symbol - an open parenthesis or closing. Suppose we want to put the opening brace here, then we need to look at the value  $d[i + 1][\text{depth} + 1]$ . If it is  $\geq k$ , then we set the current position of the opening bracket, increasing **depth** by one and go to the next character. Otherwise, we will take away from  $k$  the value  $d[i + 1][\text{depth} + 1]$ , put the closing bracket and decrease the value **depth**. In the end, we obtain the desired sequence of the bracket.

The implementation of the Java language using long arithmetic:

```
int n;  BigInteger k; // входные данные

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  $k$  types of brackets. Then the algorithm solutions will be different from the previous case except that we have to multiply the value  $D[i + 1][ndepth]$  by an amount  $k^{(2n - i - 1 - ndepth)/2}$  to take into account that this residue could be different types of braces, and the pair of brackets in this balance will only be  $2n - i - 1 - ndepth$  because the  $ndepth$  brackets are closed for opening brackets that are is a residue of (and therefore their types we can not vary).

The implementation of the Java language for the case of two types of brackets - round and square:

```

int n;  BigInteger k; // входные данные

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;
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Posted: 8 Sep 2008 22:04  
edited 1 Jun 2009 15:46

## The number of labeled graphs

Given the number  $N$  of vertices. Required to count the number  $G_N$  of different labeled graphs with  $N$  vertices (ie vertices are labeled with different numbers from 1 before  $N$ , and graphs are compared in view of this painting vertices). Undirected graph edges, loops and multiple edges are prohibited.

Consider the set of all possible edges of the graph. For every edge  $(i, j)$ , we assume that  $i < j$  (based on the undirected graph and no loops). Then the set of all possible edges of the graph has the power  $C_N^2$ , ie  $\frac{N(N-1)}{2}$ .

Since any labeled graph is uniquely determined by its edges, the number of labeled graphs with  $N$  vertices is equal to:

$$G_N = 2^{\frac{N(N-1)}{2}}$$

### Contents [hide]

- The number of labeled graphs
- Number of connected labeled graphs
- The number of labeled graphs with  $K$  connected components

## Number of connected labeled graphs

Compared with the previous task, we additionally impose the restriction that the graph must be connected.

We denote the number sought after  $Conn_N$ .

Let us learn, on the contrary, count the number of **disconnected** graphs; then the number of connected graphs obtained as  $G_N$  minus number found. Moreover, learn to count the number of **the 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 disconnected, then there exists a connected component, within which lies the root, and the rest of the graph will be a few more (at least one) connected components.

Brute over the number  $K$  of vertices in this connected component containing the root (obviously  $K = 1 \dots N - 1$ ), and find the number of such graphs. Firstly, we need to select  $K$  vertices  $N$ , ie, multiplied by the response  $C_N^K$ . Second, the connected component of the root gives a factor  $Conn_K$ . Third, the remaining graph of  $N - K$  vertices is arbitrary graph, and because it gives a multiplier  $G_{N-K}$ . Finally, the number of ways to allocate the root in a connected component of the  $K$  vertices is  $K$ . Total, with a fixed  $K$  number of **root disconnected** graph is:

$$K C_N^K Conn_K G_{N-K}$$

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 is:

$$Conn_N = G_N - \frac{1}{N} \sum_{K=1}^{N-1} K C_N^K Conn_K G_{N-K}$$

## The number of labeled graphs with $K$ connected components

Based on the previous formula, learn to count the number of labeled graphs with  $N$ vertices and  $K$  connected components.

This can be done by using dynamic programming. Learn to count  $D[N][K]$ - the number of labeled graphs with  $N$ vertices and  $K$ connected components.

Learn how to calculate the next element  $D[N][K]$ , knowing previous values. We use standard trick for solving such problems: take the top number 1, it belongs to some component, that this component we will sort out. Brute over the size  $S$ of this component, then the number of ways to choose a set of vertices is equal to  $C_{N-1}^{S-1}$ one (the top - the top one - to sort out is not necessary).

The quantity of ways to build a connected component of the  $S$ peaks we already know how to count - it  $Conn_S$ . After removal of this component of the graph we still have a graph with  $N - S$ vertices and  $K - 1$ connected components, ie we obtain a recurrence relation, on which you can calculate the values of  $D[]$ :

$$D[N][K] = \sum_{S=1}^N C_{N-1}^{S-1} Conn_S D[N - S][K - 1]$$

Total obtain code like this:

```
int d[n+1][k+1]; // изначально заполнен нулями
d[0][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 it is likely to need a long arithmetic .

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 14 Sep 2008 22:02  
EDIT: 2 May 2012 0:54

## Generation of combinations of N elements

### Combinations of N elements on K lexicographically

Statement of the problem. Are positive integers N and K. Consider the set of numbers from 1 to N. You want to display all the different subsets of power K, moreover in lexicographical order.

The algorithm is very 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 this end, this combination will find the rightmost element not yet reached its maximum value; then increase it by one and all subsequent 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 (average), if K is not close to N (i.e., if not satisfied, that  $K = N - o(N)$ ). It suffices to prove that the comparison " $a[i] < n-k + i + 1$ " are performed in the amount of  $C_{n+1}^k$  times, ie in the  $(N+1) / (N-K+1)$  times more than there are combinations of all N elements in K.

### Combinations of N elements on K with change 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.

#### Contents [[hide](#)]

- Generation of combinations of N elements
  - Combinations of N elements on K lexicographically
  - Combinations of N elements on K with change of 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 that order, when two adjacent subsets differ by exactly one element. This problem is solved directly by using [the Gray code](#) : if we each subset assign the bit mask is generating using Gray codes, these bitmaps, 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 numbers from 0 to  $2^N - 1$ , and leave only those codes that contain exactly K units. Surprising is the fact that the obtained sequence, any two adjacent masks (as well as the first and last mask) will differ in exactly two bits, we just need.

**Let us prove it.**

To prove recall the fact that the sequence G (N) Gray codes can be obtained as follows:

$$G(N) = \theta G(N-1) \cup 1G(N-1)^R$$

ie take a sequence of Gray codes for the N-1, is added to the beginning of each mask 0, we add to the answer; then take the sequence of Gray codes for the N-1, invert it, append to the beginning of each mask and add 1 to the answer.

Now we can produce the proof.

We first prove that the first and last masks 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 the NK-1 zeros, then K-1 unit. It is easy to prove by induction on N, using the above formula for the reduced sequence of Gray codes.

We now prove that any two adjacent codes will differ in exactly two bits. To do this, we turn again to the formula for the sequence of Gray codes. Let the inside of each half (formed from G (N-1)) statement is true, prove that it is true for the entire sequence. It suffices to prove that it is true in a place "gluing" the 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 give now a naive implementation that runs in  $2^N$  :

```
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 is worth noting that it is possible and in some ways more effective implementation, which will build all kinds of combinations on the go, and thus work for the  $O(C_n^k n)$ . On the other hand, this is a recursive implementation of the function and hence for small  $n$ , it is probably has a large latent constant than the previous solution.

Proper implementation itself - it is a direct follow formula:

$$G(N, K) = \emptyset G(N-1, K) \cup 1G(N-1, K-1)^R$$

This formula is 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, int old_n) {
    if (k > n || k < 0) return;
    if (!n) {
        for (int i = 0; i < old_n; ++ i)
            printf ("% d", (int) ans [i]);
        puts ("");
        return;
    }
    ans [rev? r: 1] = false;
    gen (n-1, k, ! rev? l + 1: l, ! rev? r: r-1, rev, old_n);
    ans [rev? r: 1] = true;
    gen (n-1, k-1, ! rev? l + 1: l, ! rev? r: r-1, ! rev, old_n);
}

void all_combinations (int n, int k) {
    gen (n, k, 0, n-1, false, n);
}

```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 4 Nov 2008 11:48  
EDIT: 2 Jun 2009 10:54

## Burnside lemma. Polya theorem

### Lemma Burnside

This lemma was formulated and proved **Burnside** (Burnside) in 1897, but it was found that this formula was previously opened by **Frobenius** (Frobenius) in 1887, and even earlier - **Cauchy** (Cauchy) in 1845. Therefore, this formula is sometimes called Burnside lemma, and sometimes - Cauchy-Frobenius theorem.

Burnside lemma allows you to count the number of equivalence classes in a set based on some internal symmetry.

### Objects and representations

We draw a clear line between the number of objects and the number of views.

One and the same object may correspond to different views, but, of course, any representation corresponds to exactly one object. Consequently, the set of all representations is divided into equivalence classes. Our task - in counting the number of objects it is, or what is the same, the number of equivalence classes.

### Example problem: Coloring binary trees

Suppose we consider the following problem. Required to count the number of ways to paint the root binary trees with  $n$  nodes 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 sense trees.

We now define the set of representations. Each coloring assign the defining its function  $f(v)$ , where  $v = 1 \dots n$ , and  $f(v) = 0 \dots 1$ . Then the set of ideas - a lot of different functions of this type, and size it is obviously equal  $2^n$ . At the same time, this set of representations, we introduced the partition into equivalence classes.

For example, suppose that  $n = 3$ , as the tree is as follows: root - the top 1 and the vertices 2 and 3 - her sons. Then the following functions  $f_1$  and  $f_2$  are considered equivalent:

$$\begin{array}{ll} f_1(1) = 0 & f_2(1) = 0 \\ f_1(2) = 1 & f_2(2) = 0 \\ f_1(3) = 0 & f_2(3) = 1 \end{array}$$

### Contents [hide]

- Burnside lemma. Polya theorem
  - Lemma Burnside
    - Objects and representations
    - Example problem: Coloring binary trees
    - Permutation invariant
    - The formulation of Lemma
    - The proof of Burnside
  - Polya theorem. The simplest option
    - Proof
  - Example problem: Necklaces
  - Applying Lemma Burnside together with software calculations

## Permutation invariant

Why are these two functions  $f_1$  and  $f_2$  belong to the same equivalence class? Intuitively, this is understandable - because we can to interchange sons peaks 1, ie, 2 and 3, top, and after such a conversion function  $f_1$  and  $f_2$  coincide. But technically, this means that there exists such a **permutation invariant  $\pi$**  (ie, under the terms of 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 the invariant permutation, ie applying that we do not pass from one equivalence class to another. Then, to check whether two functions  $f_1$  and  $f_2$  equivalent (ie, whether they are actually the same object), it is necessary for each permutation invariant  $\pi$  check will not be executed if the condition  $f_2\pi \equiv f_1$  (or, equivalently,  $f_1\pi \equiv f_2$ ). If at least one permutation found this equation, then  $f_1$  and  $f_2$  are equivalent, otherwise they are not equivalent.

Finding all of these permutations are invariant with respect to which our task is invariant - a key step for the application as Burnside lemma and Polya theorem. It is understood that these invariant permutation dependent on the specific tasks and to find - purely heuristic process based on intuitive considerations. However, in most cases it is sufficient to manually find a few "core" of permutations, of which all the other permutations can be obtained them by all products (and this, only the mechanical part of the work can be shifted to the computer; further discussed below for an example of the problem).

It is easy to understand that the invariant permutations form **a group** - as the product of any permutation invariant is also invariant permutation. We denote the **group of permutations of invariant** through  $G$ .

## The formulation of Lemma

To formulate the left to remind one concept of algebra. **The fixed point  $f$**  for the rearrangement  $\pi$  is an element, which is invariant under this permutation:  $f \equiv f\pi$ . For example, in our example, the fixed points will be those functions  $f$  that match the coloring, do not change when they apply the permutation  $\pi$  (not changing it in the formal sense of the equality of two functions). Denoted by the **number of fixed points** for adjustment  $I(\pi)$

Then **Lemma Burnside** is as follows: the number of classes ekvivaletnosti amounts equal to the sum of fixed points over all permutations of the group  $G$  divided by the size of this 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 is unclear how quickly to seek value  $I(\pi)$ ), it is most clearly reveals the essence of mathematics, which is based on the idea of calculating the equivalence classes.

## The proof of Burnside

Described here is the proof of Lemma Burnside is not so important for her understanding and application in practice, so it can be skipped on a first reading.

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 not nothing but a number of "invariant pairs"  $(f, \pi)$ , ie such pairs that  $f\pi \equiv f$ . It is obvious that in the formula, we have the right to change the order of summation - make foreign sum over the elements  $f$ , while inside it put the value  $J(f)$ - the number of permutations with respect to which  $f$  is invariant:

$$\text{ClassesCount}|G| = \sum_f J(f)$$

To prove this formula up a table whose columns are to be signed by all the values  $f_i$ , the line - all permutations  $\pi_j$ , and in the cells of the table will be the product  $f_i \pi_j$ . Then, if we consider the columns of this table as a set, some of them may coincide, and this will mean as that corresponding to each column  $f$  is also equivalent. Thus, as the number of different set of columns is equal to the desired value **ClassesCount**. By the way, from the point of view of the theory of groups column of the table, signed by some element  $f_i$  is the orbit of the element; for equivalent elements, obviously the same orbit, and the number it gives different orbits **ClassesCount**.

Thus, the table columns themselves fall into equivalence classes; We now fix a class and look at the columns in it. First, we note that in these columns may be only elements  $f_i$  of one equivalence class (otherwise it would turn out that some equivalent transformation  $\pi_j$  we moved to another class of equivalence, which is impossible). Second, each element  $f_i$  will meet the same number of times in all columns (this also follows from the fact that the columns correspond to equivalent elements). It can be concluded that all the columns in the same equivalence class coincide with each other as a multiset.

We now fix an arbitrary element  $f$ . On the one hand, it is found in its column exactly  $J(f)$  once (by definition  $J(f)$ ). On the other hand, all the columns in the same equivalence class are the same as the multiset. Hence, within each column of each element of the equivalence class  $g$  occurs exactly  $J(g)$  once.

Thus, if we take an arbitrary manner from each equivalence class by one column and summing the number of elements in them, we obtain, on the one hand **ClassesCount** $|G|$  (it is obtained by simply multiplying the number of columns on their size), and on the other hand - the sum of the quantities  $J(f)$  by all  $f$  (this follows from all the preceding discussion):

$$\text{ClassesCount}|G| = \sum_f J(f)$$

QED.

## Polya theorem. The simplest option

**Theorem of Polya** (Polya) is a generalization of Lemma Burnside, besides providing a convenient tool for finding the 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 went 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 the theorem of Polya, and that is very convenient to use for calculations in practice. General Polya theorem in this article will not be considered.

Denoted by  $C(\pi)$  the number of cycles in the permutation  $\pi$ . Then the following formula (**a special case of the theorem of Polya**):

$$\text{ClassesCount} = \frac{1}{|G|} \sum_{\pi \in G} k^{C(\pi)}$$

where  $k$  - the number of values that can take each item in the view  $f(v)$ . For example, in our problem-example (coloring the root of a binary tree in 2 colors)  $k = 2$ .

## Proof

This formula is a direct consequence of Lemma Burnside. To get it, we just need to find an explicit expression for the value  $I(\pi)$  in Lemma (remember, this is the number of fixed points of permutations  $\pi$ ).

Thus, we consider a permutation  $\pi$  and a certain element  $f$ . Under the action of the permutation  $\pi$  elements  $f$  move as known batcher permutation. Note that as a result to be obtained  $f \equiv f\pi$ , the permutation within each cycle must be identical elements  $f$ . At the same time, for the different loops is no relationship between the values of the elements does not occur. Thus, for each cycle permutations  $\pi$  we select a single value (including  $k$  options), and thus we get all the representations  $f$  that are invariant under this permutation, ie.:

$$I(\pi) = k^{C(\pi)}$$

where  $C(\pi)$  - the number of cycles of permutations.

## Example problem: Necklaces

The problem of "necklaces" - is one of the classical combinatorial problems. Calculate the required amount of the various necklaces  $n$  beads, each of which can be colored in one  $k$  color. When comparing two necklaces can be rotated, 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:

$$\begin{aligned}\pi_0 &= 1 \ 2 \ 3 \ \dots \ n \\ \pi_1 &= 2 \ 3 \ \dots \ n \ 1 \\ \pi_2 &= 3 \ \dots \ n \ 1 \ 2 \\ &\vdots \\ \pi_{n-1} &= n \ 1 \ 2 \ \dots \ (n-1)\end{aligned}$$

Let us find an explicit formula for the calculation  $C(\pi_i)$ . First, we note that the permutation looks like that, that  $i$ th permutation on  $j$ th position is  $i + j$  (modulo  $n$ , if greater  $n$ ). If we consider the cyclic structure  $i$ th permutation, we can see that the unit enters  $1 + i, 1 + i$  moves in  $1 + 2i, 1 + 2i - a 1 + 3i$ , and so on, until we arrive at the number  $1 + kn$ ; for the other elements are made similar statements. From this we can understand that all cycles have the same length, equal  $\text{lcm}(i, n)/i$ , ie  $n/\text{gcd}(i, n)$  ("gcd" - the greatest common divisor, "lcm" - the least common multiple). Then the number of cycles  $i$  equal to the th

permutation will simply  $\gcd(i, n)$ .

Substituting these values in the Polya theorem, we obtain the **solution** :

$$\text{Ans} = \frac{1}{n} \sum_{i=1}^n k^{\gcd(i, n)}$$

You can leave a formula in this form, but you can minimize it even more. Let's move on from the sum of all  $i$  the sum of only divisors  $n$ . Indeed, in our sum will be much the same terms: if  $i$  not a divider  $n$ , such that there exists a divisor after calculation  $\gcd(i, n)$ . Consequently, for each divider  $d|n$  term it  $k^{\gcd(d, n)} = k^d$  will take into account several times, ie, amount can be written in the form:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} C_d k^d$$

where  $C_d$ - is the number of such numbers  $i$  that  $\gcd(i, n) = d$ . Let us find an explicit expression for this quantity. Any such number  $i$  is given by:  $i = dj$  where  $\gcd(j, n/d) = 1$  (otherwise it would be  $\gcd(i, n) > d$ ). Remembering the Euler 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 the formula :

$$\text{Ans} = \frac{1}{n} \sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

## Applying Lemma Burnside together with software calculations

Not always succeed purely analytical way to obtain an explicit formula for the number of equivalence classes. In many problems, the number of permutations within the group, it may be too large for manual calculations and analytically calculate 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  $G$ . Then you can write a program that will generate all permutations of the group  $G$ , counted in each of them the number of cycles and substituting them into the formula.

Consider for example the problem of the number of colorings of the torus . There is a rectangular checkered sheet of paper , some of the cells are colored black. Then get out of this sheet cylinder gluing the two sides of lengths . Then get out of the cylinder torus by gluing two circles (the base of the cylinder) without twisting. Required to count the number of different tori (sheet was originally painted at random), assuming that the contact lines are indistinguishable, and the torus can rotate and flip. $n \times m (n < m)m$

In this problem, the representation can be considered a piece of paper  $n \times m$ , some of the cells of which are painted in black. It is easy to understand that the following types of transformations preserve the equivalence class: cyclic shift rows sheet cyclic shift column sheet, turning sheet at 180 degrees; also can be understood intuitively that these three kinds of transformations enough to generate the whole group of invariant transformations. If we are in any way enumerate the cells of the field, we can write the three permutations  $p_1, p_2,$

$p_3$  corresponding to these types of transformations. Then we can only generate all the permutations obtained as a product of this. Obviously, all such permutations are of the form  $p_1^{i_1} p_2^{i_2} p_3^{i_3}$  where  $i_1 = 0 \dots m - 1$ ,  $i_2 = 0 \dots n - 1$ ,  $i_3 = 0 \dots 1$ .

Thus, we can write the implementation to solve 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;  
}
```

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 25 Aug 2011 13:55  
EDIT: 22 Oct 2011 19:29

## The principle of inclusion-exclusion

The principle of inclusion-exclusion - this is an important combinatorial technique that allows to count the size of any set, 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 the sizes of all **pairwise** intersections of these sets, we add back the size of the intersection of all possible **triples** of sets, subtract the size of the intersection **of fours**, and so on, up to the intersection **of all** sets.

#### Formulation in terms of sets

In mathematical form The above wording is as follows:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{i=1}^n |A_i| - \sum_{\substack{i,j: \\ 1 \leq i < j \leq n}} |A_i \cap A_j| + \sum_{\substack{i,j,k: \\ 1 \leq i < j < k \leq n}} |A_i \cap A_j \cap A_k| - \dots + (-1)^{n-1} |A_1 \cap \dots \cap A_n|.$$

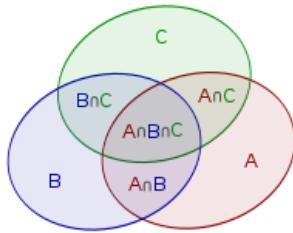
It can be written more compactly, the sum over subsets. We denote by  $B$  the set whose elements are  $A_i$ . Then the principle of inclusion-exclusion takes the form:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{C \subseteq B} (-1)^{\text{size}(C)-1} \left| \bigcap_{e \in C} e \right|.$$

This formula is credited Moivre (Abraham de Moivre).

#### The wording of using Venn diagrams

Let the diagram marked by three figures  $A$ ,  $B$  and  $C$ :



Then the area of association  $A \cup B \cup C$  is the sum of squares  $A$ ,  $B$  and  $C$  less double-covered areas  $A \cap B$ ,  $A \cap C$ ,  $B \cap C$  but with the addition of three areas covered  $A \cap B \cap C$ :

$$S(A \cup B \cup C) = S(A) + S(B) + S(C) - S(A \cap B) - S(A \cap C) - S(B \cap C) + S(A \cap B \cap C).$$

Likewise, it can be generalized to union  $n$  figures.

#### Formulation in terms of probability theory

#### Contents [hide]

- The principle of inclusion-exclusion
  - Formulation of the principle of inclusion-exclusion
    - The wording
    - Formulation in terms of sets
    - The wording of using Venn diagrams
    - Formulation in terms of probability theory
  - The proof of the principle of inclusion-exclusion
  - Application in solving problems
    - The simple task of permutations
    - The simple task of  $(0,1,2)$  is a sequence
    - The number of integer solutions of the equation
    - The number of relatively prime numbers in a given interval
    - Amount of numbers in the given interval, multiple at least one of the given numbers
    - The number of rows that satisfy a number of patterns
    - Number of tracks
    - The number of relatively prime fours
    - Number of harmonic triples
    - The number of permutations with no fixed points
  - Tasks in the online judges
  - Literature

If - this event - their probability, the probability of their association (ie, what will happen at least one of these events) is equal to:  $A_i (i = 1 \dots n) \mathcal{P}(A_i)$

$$\begin{aligned} \mathcal{P}\left(\bigcup_{i=1}^n A_i\right) &= \sum_{i=1}^n \mathcal{P}(A_i) - \sum_{\substack{i,j: \\ 1 \leq i < j \leq n}} \mathcal{P}(A_i \cap A_j) + \\ &+ \sum_{\substack{i,j,k: \\ 1 \leq i < j < k \leq n}} \mathcal{P}(A_i \cap A_j \cap A_k) - \dots + (-1)^{n-1} \mathcal{P}(A_1 \cap \dots \cap A_n). \end{aligned}$$

This amount can also be written as a sum over subsets of  $B$  whose elements are the events  $A_i$ :

$$\mathcal{P}\left(\bigcup_{i=1}^n A_i\right) = \sum_{C \subseteq B} (-1)^{\text{size}(C)-1} \cdot \mathcal{P}\left(\bigcap_{e \in C} e\right).$$

## The proof of the principle of inclusion-exclusion

To prove convenient to use 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 - a set consisting of  $A_i$ -s.

We need to prove that any element contained in at least one of the sets  $A_i$ , the formula will take into account only once. (Note that other elements not contained in either of  $A_i$ , in no way can be considered as missing in the right side of the formula).

Consider an arbitrary element  $x$  contained in exactly  $k \geq 1$  sets  $A_i$ . We will show that he thought the formula exactly once.

Note that:

- under the terms in which  $\text{size}(C) = 1$  the element  $x$  will take into account exactly  $k$  once, with a plus sign;
- in those terms, in which  $\text{size}(C) = 2$  the element  $x$  will take into account (with a minus sign) exactly  $C_k^2$  once - because  $x$  to count only those terms that correspond to the two sets of  $k$  sets containing  $x$ ;
- under the terms in which  $\text{size}(C) = 3$  the element  $x$  will take into account exactly  $C_k^3$  once, with a plus sign;
- ...
- under the terms in which  $\text{size}(C) = k$  the element  $x$  will take into account exactly  $C_k^k$  once, with the sign  $(-1)^{k-1}$ ;
- under the terms in which  $\text{size}(C) > k$  the element  $x$  will take into account zero times.

Thus, we need to calculate a sum of 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 in the binomial expression  $(1 - x)^k$ :

$$(1 - x)^k = C_k^0 - C_k^1 \cdot x + C_k^2 \cdot x^2 - C_k^3 \cdot x^3 + \dots + (-1)^k \cdot C_k^k \cdot x^k.$$

It can be seen that the  $x = 1$  expression  $(1 - x)^k$  is nothing else but  $1 - T$ . Therefore,  $T = 1 - (1 - 1)^k = 1$  what we wanted to prove.

## Application in solving problems

The principle of inclusion-exclusion is difficult to understand without good study examples of its application.

First, we look at three simple tasks "on paper", illustrating the application of the principle, and then consider the more practical tasks that are difficult to solve without the use of the principle of inclusion-exclusion.

Of particular note is the problem of "search of ways" because it demonstrates that the principle of inclusion-exclusion can sometimes lead to polynomial solutions, and not necessarily exponential.

## The simple task of permutations

How many permutations of the numbers from 0 to 9 such that the first element is greater than 1, and the last - less than 8?

Count the number of "bad" permutations, ie those in which the first element  $\leq 1$  and / or last  $\geq 8$ .

Denote  $X$  the set of permutations in which the first element  $\leq 1$ , and through  $Y$  - in which the last element  $\geq 8$ . Then the number of "bad" for the permutation inclusion-exclusion formula is:

$$|X| + |Y| - |X \cap Y|.$$

After spending some simple combinatorial calculations, we find that it is:

$$2 \cdot 9! + 2 \cdot 9! - 2 \cdot 2 \cdot 8!$$

Subtract this number from the total number of permutations  $10!$ , we get a response.

## The simple task of (0,1,2) is a sequence

How many sequences of length  $n$ , consisting only of numbers 0, 1, 2, with each number occurs at least once?

Again, let's move to the inverse problem, ie, we assume that the number of sequences which are not present in at least one of the numbers.

We denote by  $A_i$  ( $i = 0 \dots 2$ ) a plurality of sequences in which the number is not found  $i$ . Then, by the inclusion-exclusion formula the number of "bad" sequence is:

$$|A_0| + |A_1| + |A_2| - |A_0 \cap A_1| - |A_0 \cap A_2| - |A_1 \cap A_2| + |A_0 \cap A_1 \cap A_2|.$$

The dimensions of each  $A_i$  are obviously  $2^n$  (since such sequences can meet only two numbers). The power of each pairwise intersection  $A_i \cap A_j$  are 1 (as is still available, only one digit). Finally, the cardinality of the intersection of all three sets is equal to 0 (as available figures do not remain).

Remembering that we solve the inverse problem, we obtain the final answer :

$$3^n - 3 \cdot 2^n + 3 \cdot 1 - 0.$$

## The number of integer solutions of the equation

Given the equation:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 20,$$

where all  $0 \leq x_i \leq 8$  (where  $i = 1 \dots 6$ ).

Required to count the number of solutions of this equation.

Forget about the first restriction  $x_i \leq 8$ , and simply count the number of non-negative solutions of this equation. This is easily done through [the binomial coefficients](#) - we want to break the 20 elements on the 6 groups, ie distribute the 5 "wall" separating groups on 25 sites:

$$N_0 = C_{25}^5$$

Now count on inclusion-exclusion formula the number of "bad" decisions, ie, solutions of these in which one or more  $x_i$  larger 9.

We denote by  $A_k$  (where  $k = 1 \dots 6$ ) the set of solutions of an equation in which  $x_k \geq 9$ , as everyone else  $x_i \geq 0$  (for all  $i \neq k$ ). To calculate the size of the set  $A_k$ , we note that we have essentially the same combinatorial problem that was solved by the two paragraphs above, only now 9 the elements excluded from consideration and belong to the first group exactly. In this way:

$$|A_k| = C_{16}^5$$

Similarly, the capacity of the intersection of two sets  $A_k$  and  $A_p$  equal to the number:

$$|A_k \cap A_p| = C_7^5$$

The capacity of each intersection of three or more sets is zero, since 20 the elements are not enough for three or more variables is greater than or equal 9.

Combining all of this in the inclusion-exclusion formula and given that we solve the inverse problem, we finally get the answer :

$$C_{25}^5 - C_6^1 \cdot C_{16}^5 + C_6^2 \cdot C_7^5.$$

## The number of relatively prime numbers in a given interval

Suppose we are given 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 numbers.

Consider all the prime divisors of  $n$ ; 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 take advantage of inclusion-exclusion formula.

For example, it is possible for the  $2^k$ -subset of all  $p_i$ -s, find them work, and add or subtract to the inclusion-exclusion formula next term.

Final **implementation** for counting 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  $O(\sqrt{n})$ .

## Amount of numbers in the given interval, multiple at least one of the given numbers

Are given  $n$  numbers  $a_i$  and the number  $r$ . Required to count the number of numbers in the interval  $[1; r]$  that is a multiple of at least one  $a_i$ .

Algorithm for solving virtually identical to the previous task - make inclusion-exclusion formula on numbers  $a_i$ , ie each term in this formula - the number of numbers divisible by a given subset of numbers  $a_i$  (in other words, dividing their **least common multiple**).

Thus, the decision boils down to is that for  $2^n$  a subset enumerate numbers of  $O(n \log r)$  operations to find them 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 a number  $k$ . Required to count the number of rows that satisfy exactly  $k$  patterns or, in another setting, at least  $k$  patterns.

Note first that we can **easily count the number of rows** that satisfy all of the right patterns. To do this, you just have to "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 defined unambiguously), the second symbol, etc.

Now learn how to solve **the problem the first option** when the search string must meet exactly  $k$  patterns.

For this brute over and zafksiruem specific subset of  $X$  patterns size  $k$ - 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 summarize all supersets 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  $f(Y)$  is the number of rows that correspond to a set of patterns  $Y$ .

If we sum up  $ans(X)$  all  $X$ , we get the answer:

$$ans = \sum_{X : |X|=k} ans(X).$$

However, by doing so we got the solution for the time of the order  $O(3^k \cdot k)$ .

The decision can be accelerated, noting that in different  $ans(X)$  summation is often conducted on the same sets  $Y$ .

Perevernëm inclusion-exclusion formula and will lead to the summation  $Y$ . Then it is easy to see that the set  $Y$  will take into account in the  $C_{|Y|}^k$  inclusion-exclusion formula, always with the same sign  $(-1)^{|Y|-k}$ :

$$ans = \sum_{Y : |Y| \geq k} (-1)^{|Y|-k} \cdot C_{|Y|}^k \cdot f(Y).$$

The solution is obtained with the asymptotic behavior  $O(2^k \cdot k)$ .

We turn now to the **second embodiment of the problem** : when the search string must meet at least  $k$  patterns.

Clearly, we can simply use the solution of the first version of the problem and summarize the answers to  $k$  before  $n$ . However, you will notice that all the arguments will continue to be true, but in this version of the problem by the sum  $X$  is not just for those sets whose size is equal to  $k$ , and in all the sets with the size  $\geq k$ .

Thus, in the final formula to  $f(Y)$  be another factor is not a binomial coefficient with some familiar, and their sum:

$$(-1)^{|Y|-k} \cdot C_{|Y|}^k + (-1)^{|Y|-k-1} \cdot C_{|Y|}^{k+1} + (-1)^{|Y|-k-2} \cdot C_{|Y|}^{k+2} + \dots + (-1)^{|Y|-|Y|} \cdot C_{|Y|}^{|Y|}.$$

Looking into Graham ( [Graham, Knuth, Patashnik. "Concrete Mathematics" \[1998\]](#) ), we see a well-known formula for the **binomial coefficients** :

$$\sum_{k=0}^m (-1)^k \cdot C_n^k = (-1)^m \cdot C_{n-1}^m.$$

Applying it here, we find that the entire amount of the binomial coefficients is minimized in:

$$(-1)^{|Y|-k} \cdot C_{|Y|-1}^{|Y|-k}.$$

Thus, for this version of the problem, we also got a solution with the asymptotic behavior  $O(2^k \cdot k)$ :

$$ans = \sum_{Y : |Y| \geq k} (-1)^{|Y|-k} \cdot C_{|Y|-1}^{|Y|-k} \cdot f(Y).$$

## Number of tracks

There is a field  $n \times m$ , some  $k$  cells which - impenetrable wall. On the field into the cage  $(1, 1)$  (lower left cell) is initially robot. The robot can only move left or up, and in the end he has to get into the cage  $(n, m)$ , avoiding all obstacles. Required to count the number of ways in which he can do it.

We assume that the size  $n$  and  $m$  very large (say, up  $10^9$ ), and the number  $k$  small (of the order 100).

To solve at once for convenience **sort the obstacles** in the order in which we can avoid them: ie, for example, coordinate  $x$ , and at equality - coordinate  $y$ .

Also, learn how to solve the problem immediately without obstacles: ie learn how to count the number of ways to walk from one cell to another. If one coordinate we need to go through  $x$  cells, and on the other -  $y$  cells, from simple combinatorics, we obtain a formula through [the binomial coefficients](#):

$$C_{x+y}^x$$

Now count the number of ways to walk from one cell to another, avoiding all the obstacles, you can use the **inclusion-exclusion formula**: count the number of ways to walk, stepping on at least one obstacle.

This can be, for example, to sort the subset of those obstacles, which we do come, count the number of ways to do it (just multiplying the number of ways to reach the starting cell to the first of the selected obstacles, from the first to the second obstacle, and so on), and then add or subtract the number on the answer, according to the standard inclusion-exclusion formula.

However, it will again be non-polynomial solution - for the asymptotic behavior  $O(2^k k)$ . We show how to obtain a **polynomial solution**.

Will solve the **dynamic programming**: how to calculate the number  $d[i][j]$ - the number of ways to walk from  $i$  point to the  $j$ th without stepping while neither one obstacle (except themselves  $i$  and  $j$ , of course). All in all we would  $k + 2$  point to the obstacles are added as the start and end of the cell.

If we for a moment forget about all the obstacles and simply count the number of paths from the cell  $i$  into the cell  $j$ , we thereby will take into account some of the "bad" path passing through the obstacles. Learn to count the number of these "bad" ways. Brute over the first obstacle  $i < t < j$ , to which we come, then the number of paths is equal to  $d[i][t]$  times the number of arbitrary paths  $t$  in  $j$ . Summing it all  $t$ , we count the number of "bad" ways.

Thus, the value  $d[i][j]$  we have learned to count in time  $O(k)$ . Hence, the solution of the whole problem has the asymptotic behavior  $O(k^3)$ .

## The number of relatively prime fours

Given  $n$  the numbers:  $a_1, a_2, \dots, a_n$ . Required to count the number of ways to choose the four numbers so that their combined greatest common divisor is equal to one.

We will solve the inverse problem - count the number of "bad" fours, ie such quadruples in which all numbers are divided by the number  $d > 1$ .

We use the inclusion-exclusion formula, summing the number of fours divisible by the divisor  $d$  (but perhaps dividing and greater divisor):

$$ans = \sum_{d \geq 2} (-1)^{\deg(d)-1} \cdot f(d),$$

where  $\deg(d)$ - is the number of prime factorization of a number  $d$ .  $f(d)$ - the number of fours divisible by  $d$ .

To calculate the function  $f(d)$ , you simply count the number of multiples  $d$ , and [binomial coefficients](#) to calculate the number of ways to choose from these four.

Thus, using the inclusion-exclusion formula, we summarize the number of fours divisible by primes, then subtract the number of fours divisible by the product of two primes, add the Quartet divisible by three simple, etc.

## Number of harmonic triples

Given the number  $n \leq 10^6$ . Required to count the number of triples  $2 \leq a < b < c \leq n$  that they are harmonic triples, ie.:

- or  $\gcd(a, b) = \gcd(a, c) = \gcd(b, c) = 1$ ,
- or  $\gcd(a, b) > 1, \gcd(a, c) > 1, \gcd(b, c) > 1$ .

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 prime to one number threes and not relatively prime to the other number three.

Thus, the number of non-harmonic triples equal to the sum of all the numbers from 2 before the  $n$  works of the number relatively prime to the current number of numbers on the number of non-mutually prime numbers.

Now all that remains for us to solve the problem - it is to learn to count numbers for each segment in  $[2; n]$  the amount of numbers relatively prime (or coprime) with him. Although this problem has already been highlighted above, the solution described above is not appropriate here - it requires the factorization of each number from 2 before  $n$ , and then iterate through all possible products of primes in the factorization.

Therefore, we need a faster solution that calculates the answers to all numbers from the interval  $[2; n]$  immediately.

To do this, you can implement a **modification of the sieve of Eratosthenes** :

- Firstly, we need to find all the numbers in the interval  $[2; n]$  in which no simple factorization is not included twice. In addition, the formula for inclusion-exclusion, we need to know how many simple comprises the factorization of the number of each.

To do this, we need to have arrays  $deg[]$  that store for each of the number of primes in its factorization, and  $good[]$  - containing for each number *true* or *false* - all prime enter it in the degree  $\leq 1$  or not.

After that, during the sieve of Eratosthenes in the processing of the next prime number, we'll go over all the numbers that are multiples of the current number, and increase  $deg[]$  them, and all the multiples of the square of the current simple - deliver  $good = false$ .

- Secondly, we need to find an answer to all the numbers from  $2$  before  $n$ , ie, array  $cnt[]$  - the number of numbers that are not relatively prime to the data.

To do this, let us recall how the inclusion-exclusion formula - here we actually implement it the same, but with inverted logic: if we iterate term and see in what inclusion-exclusion formula for what numbers this term included.

Thus, suppose we have a number  $i$  for which  $good[i] = true$ , ie is the number of participating in the inclusion-exclusion formula. Brute over all multiples  $i$ , and to answer  $cnt[i]$  each of these numbers, we need to add or subtract value  $\lfloor N/i \rfloor$ . Sign - the addition or subtraction - depends on  $deg[i]$  if  $deg[i]$  is odd, it is necessary to add, or subtract.

### 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;
}

```

Asymptotic behavior of such solutions is  $O(n \log n)$  because almost every number  $i$  it makes about  $n/$  nested loop iterations.

## The number of permutations with no fixed points

We prove that the number of permutations of length  $n$  without fixed points is the next 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 rounded to the nearest whole - you get exactly the number of permutations with no fixed points)

Denote  $A_k$  the set of permutations length  $n$  from a fixed point at the position  $k$  ( $1 \leq k \leq n$ ).

We now use the inclusion-exclusion formula to count the number of permutations with at least one fixed point. To do this, we need to learn to count the size of the sets-intersections  $A_i$ , they are as follows:

$$\begin{aligned} |A_p| &= (n-1)! , \\ |A_p \cap A_q| &= (n-2)! , \\ |A_p \cap A_q \cap A_r| &= (n-3)! , \\ &\dots , \end{aligned}$$

because if we know that the number of fixed points is  $x$ , by the same token, we know the position of  $x$  elements of the permutation, 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 the size  $x$  of the  $n$ -element set equal  $C_n^x$ , we 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 with no 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 **the 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 - are 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 (and not to all, as we have just solved). Will result in the formula as given above exact formula, but it will go up to the sum  $k$ , not before  $n$ .

## Tasks in the online judges

A list of tasks that can be solved using the principle of inclusion-exclusion:

- [UVA # 10325 "The Lottery"](#) [Difficulty: Low]
- [UVA # 11806 "Cheerleaders"](#) [Difficulty: Low]
- [TopCoder SRM 477 "CarelessSecretary"](#) [Difficulty: Low]
- [TopCoder TCHS 16 "Divisibility"](#) [Difficulty: Low]
- [SPOJ # 6285 NGM2 "Another Game With Numbers"](#) [Difficulty: Low]
- [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 "Sky Code"](#) [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"](#)

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:19  
EDIT: 11 Jun 2008 11:20

## Games on arbitrary graphs

Let the game is played by two players on a graph  $G$ . That is, the current state of the game - this is some vertex, and each vertex of the edges are in the vertices, in which you can go to the next move.

We consider the most general case - the case of arbitrary directed graph with cycles. Required for a given starting position to determine who will benefit from the best game both players (or determine that the result is a draw).

We solve this problem very effectively - find the answers to all vertices of the graph in linear time with respect to the number of edges -  $O(M)$ .

### Contents [hide]

- Games on arbitrary graphs
  - Description of the algorithm
  - Implementation
  - An example of the problem. "Police and Thief"

### 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 a vertex does not have outgoing edges.

We have the following facts :

- if from some vertex has an edge in losing vertex, then this vertex is winning;
- if from some vertex all edges come to winning the top, then this vertex losing;
- If at any point yet remain uncertain summit, but none of them do not fit the first nor the second rule under, all these peaks - no man.

Thus, the algorithm is already clear, working for the asymptotic behavior of  $O(NM)$  - we iterate through all the vertices, each trying to apply the first or the second rule, and if we have made any changes, then repeat all over again.

However, the process of finding and updating can be greatly accelerated, bringing to the asymptotic behavior of linear.

Brute over all vertices, about which was originally known to be winning or losing. From each of these empty the next search depth. This dfs will move in reverse edges. First of all, it will not go into the vertices that have already been defined as winning or losing. Further, if the depth-first search is trying to go from losing the top of a vertex, then it marks it as winning, and goes to her. If dfs trying to go from winning the top in a vertex, then it should check whether all edges are from this vertex in winning. This test is easy to implement in  $O(1)$  if at each vertex will keep count of edges that lead to winning the top. So if dfs tries to go from winning the top in a vertex, then it increases its count, and if the counter is equal to the number of edges emanating from that vertex, then this vertex is marked as losing, and depth-first search goes to the vertex . Otherwise, if the target vertex and is not defined as winning or losing, the depth-first search does not come into it.

Total, we see that each and every winning losing the top of our algorithm is visited exactly once, and no man tops and not visited. Consequently, the asymptotic behavior is really  $O(M)$  .

### Implementation

Consider the implementation of depth-first search, on the assumption that the graph of the game is built in memory, the degree of outcome are counted and recorded in degree (this is just a counter, it will be reduced if there is an edge in winning the top), as well as winning or losing initially have tops marked.

```
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);
        }
}
```

### An example of the problem. "Police and Thief"

The algorithm has become more clear, consider it a concrete example.

**Condition of the problem** . There is a field of size  $M \times N$  cells, 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 a thief, then won a cop. If the thief will be in the cage with the release (and in this cell should not be a police officer) will win a thief. A police officer can walk in 8 directions, the thief - only 4 (along the coordinate axes). And a policeman and a thief can skip your turn. The first move is made by a police officer.

**Graphing** . We construct a graph of the game. We need to formalize the rules of the game. Current state of play is determined by the coordinates of the police  $P$ , thief  $T$ , as well as boolean  $P_{step}$ , which determines who will make the next move. Therefore, the vertex of the

graph is defined by the triple (P, T, Pstep). Count to build easy, just matching condition.

Next you need to determine which vertices are winning or as lost initially. There is a **subtle point**. Winning / losing vertex coordinates in addition depends on Pstep - whose turn. If now the course of the police, then the top of winning if the coordinates of the police and the thief are the same; Losing vertex, if it is not successful and a thief is output. If the course is now a thief, then the top of winning, if the thief is on the way out and losing, if it is not successful, and coordinates police and thief are the same.

The only thing that you need to decide - to build the graph explicitly or do it "on the fly", right in the DFS. On one hand, if we construct a graph beforehand, it will be less likely to be mistaken. On the other hand, it will increase the amount of code, and the work will be several times slower than if you build the Count "on the go".

**The implementation** of the 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 main () {
    int n, m;
    cin >> n >> m;
    vector <string> a (n);
    for (int i = 0; i <n; ++ i)
        cin >> a [i];

    for (int p = 0; p <n * m; ++ p)
        for (int t = 0; t <n * m; ++ t)
            for (char pstep = 0; pstep <= 1; ++ pstep) {
                int px = p / m, py = p% m, tx = t / m, ty = t% m;
                if (a [px] [py] == '*' || a [tx] [ty] == '*') continue;

                bool & wwin = win [p] [t] [pstep];
                bool & llose = loose [p] [t] [pstep];
                if (pstep)
                    wwin = px == tx && py == ty, llose = ! wwin && a [tx] [ty] == 'E';
                else
                    wwin = a [tx] [ty] == 'E', llose = ! wwin && px == tx && py == ty;
                if (wwin || llose) continue;

                state st = {p, t, ! pstep};
                g [p] [t] [pstep] .push_back (st);
                st.pstep = pstep! = 0;
                degree [p] [t] [pstep] = 1;

                const int dx [] = {-1, 0, 1, 0, -1, -1, -1, 1};
                const int dy [] = {0, 1, 0, -1, -1, 1, -1, 1};
                for (int d = 0; d <(pstep? 8: 4); ++ d) {
                    int ppx = px, ppy = py, ttx = tx, tty = ty;
                    if (pstep)
                        ppx += dx [d], ppy += dy [d];
                    else
                        ttx += dx [d], tty += dy [d];
                    if (ppx >= 0 && ppx <n && ppy >= 0 && ppy <m && a [ppx] [ppy]! = '*' &&
                        ttx >= 0 && ttx <n && tty >= 0 && tty <m && a [ttx] [tty]! = '*')
                    {
                        g [ppx * m + ppy] [ttx * m + tty] [! pstep] .push_back (st);
                    }
                }
            }
        }
    }
}

```

```

        ++ Degree [p] [t] [pstep];
    }
}
}

for (int p = 0; p <n * m; ++ p)
    for (int t = 0; t <n * m; ++ t)
        for (char pstep = 0; pstep <= 1; ++ pstep)
            if ((win [p] [t] [pstep] || loose [p] [t] [pstep]) &&! used [p] [t] [pstep])
                dfs (p, t, pstep! = 0);

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");
}

```

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 17 Jul 2009 23:00  
 EDIT: 15 Jul 2014 18:05

## Theory Shpraga Grande. Them

### Introduction

Theory Shpraga Grande - a theory describing the so-called **peer** (eng. "impartial") play two players, ie games in which the allowed moves and winning / losing depend only on the state of the game. On which of the two players go, does not depend on anything: that is, 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 opponent's position).

It is assumed that the game **is finite**, i.e. for any strategy players will sooner or later come to a **losing** position from which there is no transition to other positions. This position is a loser for a player who has to make a move from this position. Accordingly, it is advantageous for the player who came into this position. Clearly, a tie in a game like this does not happen.

In other words, this game can be completely described **directed acyclic graph**: vertices there are state of the game, and the edges - transitions from one state to another game as a result of the progress of the current player (again, in the first and second players are equal). One or more vertices have outgoing edges, they is as lost vertices (for the player who must perform the course of such a vertex).

Since a tie does not happen, then all the states of the game are divided into two classes: **winning and losing**. Winning - a state such that there exists a course of the current player that will lead to inevitable defeat another player even with his best game. Accordingly, the losing state - a state from which all transitions are in the state, leading to the victory of the second player, despite the "resistance" of the first player. In other words, a winner will be the state from which there is at least one transition in a losing condition, and losing a state from which all transitions lead to winning state (or from which there are no transitions).

Our task - for any given game to classify the state of the game, ie, for each state to determine winning or losing it.

Theory of games developed independently Shprag Roland (Roland Sprague) in 1935 and Patrick Michael Grundy (Patrick Michael Grundy) in 1939

### Game "Him"

This game is one example of the above described games. Moreover, as we shall see later, **any** of the games of two players of equal actually equivalent to the game "them" (eng. "nim"), so learning this

### Contents [hide]

- Theory Shpraga Grande. Them
  - Introduction
  - Game "Him"
    - Game Description
    - Solution of neem
  - Equivalence of any game nimu. Theorem Shpraga Grande
    - Lemma of Nimes with increases
    - Theorem Shpraga Grande on the equivalence of any game nimu
  - Application of Theorem Shpraga Grande
  - Patterns in the values of Shpraga Grande
  - Examples of games
    - "Tic-Tac"
    - "Noughts and crosses - 2"
    - "Pawns"
    - "Lasker's nim"
    - "The game of Kayles"
    - Grundy's game
    - "Stair him"
    - "Nimble" and "Nimble-2"
    - "Turning turtles" and "Twins"
    - Northcott's game
    - Triomino
    - Chips on a graph
  - Implementation
  - Generalization of neem: These Moore ( $\text{K}^{\text{a}}\text{-nim}$ )
  - "Him giveaway"
  - Tasks in the online judges
  - Literature

game will automatically allow us to solve all the rest of the game (but more on that later).

Historically, this game was popular back 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 belong to the XVI century. The name "it" came up with the mathematician Charles Bud (Charles Bouton), who in 1901 published a full analysis of this game. Origin of the name "it" is not known.

## Game Description

The game "it" is a next game.

There are several piles in each of which several stones. In one move, the player can take from 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.

Thus, the state of the game "it" uniquely describes an unordered set of natural numbers. In one move is allowed strictly 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 the XOR-sum of the sizes of heaps nonzero. Otherwise, the current player is on the losing state. (XOR-sum of the numbers  $a_i$  is an expression  $a_1 \oplus a_2 \oplus \dots \oplus a_n$ , where  $\oplus$  - bitwise exclusive or)

### Proof .

The main essence of the proof below - there **symmetrical strategy for the enemy** . We show that, being in a state of zero XOR-sum, the player will not be able to get out of this state - in any of its transition to a state with a nonzero XOR-sum of the enemy exists a counter move, returning XOR-sum back to zero.

We now turn to the formal proof (it will be constructive, ie, we show how it looks symmetrical strategy of the enemy - what kind of progress will need to make it).

Will prove the theorem by induction.

To empty the neem (when the sizes of all piles are zero) XOR-sum is equal to zero, and the theorem is true.

Suppose now that we want to prove the theorem for a certain state of the game, of which there is at least one transition. Using the induction hypothesis (and acyclic games), we believe that the theorem is proved for all the states in which we can get from this.

Then the proof is divided into two parts: if the XOR-sum  $s$  in its current state  $= 0$ , it is necessary to prove that the current state is losing, ie, all transitions from it are in a state with XOR-sum  $t \neq 0$ . If  $s \neq 0$  it is necessary to prove that there exists a transition that leads us to a state of  $t = 0$ .

- Suppose  $s = 0$ , then, we want to prove that the current state - disadvantageous. Consider any transition from the current state of neem: denote the  $p$  number of variable handful through  $x_i$  ( $i = 1 \dots n$ ) - the size of piles to travel through  $y_i$  ( $i = 1 \dots n$ ) - after the move. 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  $y_p < x_p$  means that  $t \neq 0$ . Hence, the new state will have a non-zero XOR-sum, ie, according to the base of induction is advantageous, as required.

- Let  $s \neq 0$ . Then our task - to prove that the current state - winning, ie of course it exists in a losing state (zero XOR-sum).

Consider a bit of a record  $s$ . Take a senior non-zero bit, let his number is  $d$ . Let  $k$ - the number of the piles, in the size of  $x_k$  which  $d$ 'th bit different from zero (so  $k$  there will otherwise be in the

XOR-sum of  $s$ this bit would not have been different from zero).

Then, it is argued, required course - a change  $k$ of th pile, making its size  $y_k = x_k \oplus s$ .

Sure.

First, you need to verify that this course correct, ie that  $y_k < x_k$ . However, this is true because all the bits, senior  $d$ th, at  $x_k$ and  $y_k$ the same, and in  $d$ th bit from  $y_k$ zero to and from  $x_k$ the unit will be.

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, given our course - really move in a losing state, and this proves that the current state advantageous.

The theorem is proved.

**Corollary** . Any condition it games can be replaced by an equivalent condition, consisting of only a handful of size equal to the XOR-sum of the sizes of heaps in the old state.

In other words, the analysis of neem with several piles can be calculated XOR-sum  $s$ of their sizes, and proceeding to the analysis of only a handful of neem size  $s$ - as the theorem just proved, winning / losing will not change.

## Equivalence of any game nimu. Theorem Shpraga Grande

Here we show how any game (an equal game two players) to put them in line. In other words, any state of any game we will learn how to put them in line, a handful of fully describing the state of the original game.

### Lemma of Nimes with increases

We first prove an important lemma - **lemma Nimes with increases** .

Namely, consider the following modified them: all the same as in the conventional Nima, but now form an additional stroke permitted: instead of decreasing, conversely, **increase the size of some piles** . To be more precise, the player's turn now lies in the fact that it takes a non-zero number of stones from some piles, or increase the size of a handful (in accordance with certain rules. See the next paragraph).

It is important to understand that the rules of how the player can make to increase, **we are not interested** - but such rules still have to be to our game was still **acyclic** . Below under "Examples of games" are considered examples of such games: "stair him", "nimble-2", "turning turtles".

Again, we will prove the lemma in general for all games of this type - the type of games "with them increases"; specific rules increases in the proof can not be used.

**The formulation of the lemma** . Him with increases equivalent to the usual nimu, in the sense that the winning / losing state is determined by Theorem Bouton according XOR-sum of the sizes of heaps. (Or, in other words, the essence of the lemma is that the increase is useless, they do not make sense to apply the optimal strategy, and they do not change a winning / losing compared to conventional Nimes.)

### Proof .

The idea of the proof as in Theorem Bouton - there **symmetrical strategy** . We show that the increase does not change anything, because after one of the players will resort to increase, the other will be able to answer it symmetrically.

In fact, assume that the current player makes progress, any increase in piles. Then his opponent can just answer it, reducing it back to a pile of old values - because usually moves neem we still may be used freely.

Thus, a symmetrical response to the course, the course will increase, decrease back to the old size of the piles. Therefore, after this answer game returns back to the same size piles, ie a player who has made an increase in anything not benefit from it. Because game acyclic, then sooner or later run out of moves, increase, and the current player must make a move-reduction, and this means that the presence of increasing the stroke does not change anything.

## Theorem Shpraga Grande on the equivalence of any game nimu

Let us now turn to the most important fact in this article - the theorem on the equivalence nimu any equitable game two players.

**Theorem Shpraga Grande**. Consider any state of  $v$  a game of two equal players. Let him out there in some state transitions (where). It is alleged that the state of the game can be associated with a handful of neem certain size (which will be fully describe the state of our game - that these two states of two different games are equivalent). This number - called **value Shpraga Grande** state  $x_i$  ( $i = 1 \dots k$ )  $k \geq 0$

Moreover, this number  $x$  can be found in 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 of 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 vertices without outgoing edges, slowly **count values Shpraga Grande for all the states of our game**. If the value Shpraga Grande any state is zero, then this state is losing, anyway - winning.

**Proof**. Will prove the theorem by induction.

For the vertices of which there is no transition, the value  $x$  according to the theorem will be obtained as mex the empty set, ie  $x = 0$ . But, in fact, no state transitions - is losing condition, and he really should correspond to them, a bunch of size 0.

Consider now any state  $v$ , from which there transitions. By induction, we may assume that for all the states  $v_i$  in which we can move from its current state, the values  $x_i$  already calculated.

Count value  $p = \text{mex}\{x_1, \dots, x_k\}$ . Then, according to the definition of the function mex, we obtain that for any number  $i$  in between  $[0; p]$  there is at least one corresponding transition in some of the  $v_i$ -s state. Furthermore, there may be additional transitions - in states with values Grandi large  $p$ .

This means that the current state **equivalent to the state nimu with increases with the size of the heap**: in fact, we have a transition from the current state to a state with heaps of smaller size, and can be transitions to states of large dimensions.

Consequently, the value  $\text{mex}\{x_1, \dots, x_k\}$  really 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 equitable game two players to determine the winning / losing the current state  $v$ .

The function that each of the game puts him in line-number, called the **Sprague-Grundy theorem**.

So, to calculate the Sprague-Grundy theorem for the current state of a game, you need to:

- Write down all the possible transitions from the current state.
- Each transition may lead either to a single game or a **sum of independent games**.

In the first case - just count the Grundy function recursively to this new state.

In the second case, when the transition from the current state leads to the sum of several

independent games - recursively count for each of these games Grundy function, then we say that the function of Grundy sum games is XOR-sum of the values of these games.

- Once we find Grundy function for each possible transition - think **mex** of the values, and found the number - is the desired value for the current state of Grundy.
- If the value is zero Grande, the current state is losing, anyway - winning.

Thus, in comparison with Theorem Shpraga Grande here we take into account the fact that the game can be transitions from the individual states in **the amount of several games**. To work with the sums of games, we first replace every game its value Grundy, ie a bunch of them, a certain size. After that, we come to the sum of several piles of them - ie nimu to normal, the answer to which, according to Theorem Bouton - XOR-sum of the sizes of heaps.

## Patterns in the values of Shpraga Grande

Very often for specific tasks when required to learn to count Sprague-Grundy theorem for a given game, helping **the study of tables of values** of this function in the search for patterns.

In many games that seem very difficult for theoretical analysis, Sprague-Grundy theorem in practice is periodic or as having a very simple form, which is easily seen "by eye". In the majority of cases seen patterns are correct, and if you want to prove by mathematical induction.

However, not always Sprague-Grundy theorem contains simple patterns, and for some, even very simple in formulation, games whether such laws are still open (eg, "Grundy's game" below).

## Examples of games

To illustrate the theory Shpraga Grande, we shall understand a few problems.

Particular attention should be paid to the problem of "stair him", "nimble-2", "turning turtles", which demonstrated a nontrivial reducing the original problem to nimu with increases.

### "Tic-Tac"

**Condition**. Consider plaid stripes size  $1 \times n$  cells. In one move, the player needs to put a cross, but it is forbidden to put two crosses next (neighboring cells). The player who can not make a move. Say who will win the game at the optimum.

**Solution**. When a player puts the X in any cell, it can be assumed that the whole band splits into two independent halves: the left of the cross and to his right. In this case, the cell itself with a cross and its left and right neighbor destroyed - because they can not be anything else to put.

Therefore, if we enumerate the cells of the strip  $1$  up  $n$ , then put a cross in position  $1 < i < n$ , the strip will disintegrate into two strips of length  $i - 2$  and  $n - i - 1$ , that is, we move into the sum of two games  $i - 2$  and  $n - i - 1$ . If the cross is placed in position  $1$ , or  $n$ , it is a special occasion - we just move on to the state  $n - 2$ .

Thus, the function Grande  $g[n]$  is (for  $n \geq 3$ ):

$$g[n] = \text{mex} \left\{ g[n - 2], \bigcup_{i=2}^{n-1} (g[i - 2] \oplus g[n - i - 1]) \right\}.$$

That is  $g[n]$  obtained as **mex** from the set consisting of numbers  $g[n - 2]$ , as well as all possible values of expression  $g[i - 2] \oplus g[n - i - 1]$ .

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  $n$ , we can see that, since  $n = 52$  the sequence  $g[n]$  becomes periodic with a period 34. This pattern persists beyond (that

probably can be proved by induction).

## "Noughts and crosses - 2"

**Condition**. Again, the game is conducted on a strip of  $1 \times n$  cells, and players take turns putting one cross. The winner is the one who will put three crosses in a row.

**Solution**. Note that if  $n > 2$  we left after his move two crosses near or through one space, the opponent wins the next move. Consequently, if one player put somewhere cross, the other player is not profitable to put a cross in its neighboring cells, as well as in the neighboring adjacent (ie, at a distance 1and 2put unprofitable, it will lead to the defeat).

Then the solution is obtained almost similar to the previous problem, but now removes cross in each half by not one, but two cells.

## "Pawns"

**Condition**. There is a field  $3 \times n$  in which the first and the third row are for  $n$ 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 mandatory.

**Solution**. Let us see what happens when one pawn makes a move forward. Next move opponent will be required to eat it, then we will have to eat pawn of the opponent, then he will eat, and finally, our enemy pawn eat and stay, "upērshis" a pawn of the opponent. Thus, if at the beginning we walked on in the column  $1 < i < n$ , resulting in a three-column  $[i - 1; i + 1]$  board virtually annihilated, and we proceed to the sum of the size of the game  $i - 2$  and  $n - i - 1$ . Borderline cases  $i = 1$  and  $i = n$  lead us simply to the board size  $n - 2$ .

Thus, we obtain an expression for the function Grundy, similar to the above problem "Noughts and crosses."

## "Lasker's nim"

**Condition**. There are  $n$ heaps of stones set sizes. In one move, the player can take any non-zero number of stones from a handful, or else to share any pile into two non-empty piles. The player who can not make a move.

**Solution**. Writing both types of transitions, easy to get Sprague-Grundy theorem as:

$$g[n] = \text{mex} \left\{ \bigcup_{i=0}^{n-1} g[i], \bigcup_{i=1}^{n-1} \left( g[i] \oplus g[n-i] \right) \right\}.$$

However, you can build a table of values for small  $n$  and see the simple pattern:

$n$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$g[n]$	0	1	2	4	3	5	6	8	7	9	10	12	11	13	14	16	15	17	18	20

Here it is seen that  $g[n] = n$  for values that equal 1or 2modulo 4, and  $g[n] = n \pm 1$  to numbers equal 3and 0modulus 4. You can prove this by induction.

## "The game of Kayles"

**Condition**. There are  $n$ pins on display in a row. For one player can knock out punch or a skittle or two standing next to the pins. The winner is the one who knocked last pins.

**Solution**. And when a player kicks a kingpin, and when he knocks two - game splits into the sum of two independent games.

It is easy to obtain an expression for the Sprague-Grundy theorem:

$$g[n] = \text{mex} \left\{ \bigcup_{i=0}^{n-1} (g[i] \oplus g[n-1-i]), \bigcup_{i=0}^{n-2} (g[i] \oplus g[n-2-i]) \right\}.$$

Count for a table 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, after a certain time, the sequence becomes periodic with a period 12. In the future, this periodicity is not violated.

## Grundy's game

**Condition**. There are  $n$ heaps of stones, the size of which will be denoted by  $a_i$ . In one move, the player can take any size at least a handful 3and divide it into two non-empty heaps of unequal sizes. The player who can not make a move (ie, when the sizes of all the remaining piles of less than or equal to two).

**Solution**. If  $n > 1$ , all these several piles obviously - independent games. Therefore, our task - to learn to look Sprague-Grundy theorem for a handful, and the answer to a few piles will be obtained as their XOR-sum.

For a handful of this function is built as easy enough to see all the possible transitions:

$$g[n] = \text{mex} \left\{ \bigcup_{\substack{i=[1 \dots n-1], \\ i \neq n-i}} (g[i] \oplus g[n-i]) \right\}.$$

What is this game is interesting - the fact that so far it has been found to the general law. Despite the assumption that the sequence  $g[n]$ to be periodic, it has been calculated up to 2<sup>35</sup>and periods in this region were detected.

## "Stair him"

**Condition**. There is a staircase with  $n$ steps (numbered from 1to  $n$ ), on  $i$ th step is  $a_i$ coins. In one move is permitted to move some non-zero number of coins with  $i$ th at  $i - 1$ of th step. The player who can not make progress.

**Solution**. If you try to reduce this problem to nimu "head", it turns out that the course we have - is to reduce how much one of a handful of something, and a bunch of other simultaneous increase by the same amount. As a result, we obtain a modification of the neem, which is very difficult to solve.

Proceed in another way: consider only the even-numbered steps:  $a_2, a_4, a_6, \dots$ . Let's see how this will change the set of numbers when making a turn.

If the move is done with an even  $i$ , then this move means reducing the number  $a_i$ . If the move is made with odd  $i$  ( $i > 1$ ), this means an increase  $a_{i-1}$ .

It turns out that our problem - it is common to increases with the size of piles  $a_2, a_4, a_6, \dots$ .

Consequently, the function of Grande it - is XOR-sum of the numbers of the form  $a_{2i}$ .

## "Nimble" and "Nimble-2"

**Condition**. There checkered strip  $1 \times n$  on which the  $k$  coins:  $i$  Star coin is in  $a_i$ th cell. In one move, the player may take some coin and move it to the left to an arbitrary number of cells, but so that it did not come out beyond the strip. In the game "Nimble-2" further prohibited to jump 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 a set of numbers , it is clear that one player may take any of these numbers and reduce it, but a loss is when all the numbers are equal to zero. Consequently, the game "Nimble" - is **it normal** , and the answer to the problem is the XOR-sum of the numbers  $a_i - 1 (i = 1 \dots k) a_i - 1$

**Decision "Nimble-2"**. We enumerate the coins in the order they appear from left to right. Then we denote  $d_i$ the distance from  $i$ th to  $i - 1$ th 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  $d_p$ a number  $q$ and add this number  $q$ to  $d_{p+1}$ . Thus, this game - it's actually "**ladder him**" on numbers  $d_i$ (it is only necessary to change the order of the numbers on the opposite).

## "Turning turtles" and "Twins"

**Condition**. Dana checkered stripe size  $1 \times n$ . In each cell should either cross or crosses. In one move, you can take some toe and turn it into a cross.

In this case, **further** permitted to choose one of the cells to the left of a variable and change it to the opposite value (ie, replace toe on the cross, and the cross - at the toe). In the game "turning turtles" this is not required (ie, move the player may be limited by converting toe in a cross) and "twins" - sure.

**Decision "turning Turtles"**. It is argued that this game - it is a regular on numbers  $b_i$ , where  $b_i$ - the position of  $i$ th toe (1-indexed). We verify this assertion.

- If a player is simply reversed toe on the cross without using additional course - it can be understood as the fact that he just took the whole pile corresponding to that toe. In other words, if the player changed his toe on the cross in a position , by the same token, he took a handful of size and made her size zero. $x (1 \leq x \leq n)x$
- If a player took an additional course, that in addition to what has changed in the toe position  $x$ on the cross, he still changed the cell in position , then we can assume that it reduced pile to size . Indeed, if the position was formerly a cross - that, in fact, after the player's turn there will be a toe, ie will be a bunch of size . And if the position was formerly toe, after the player's turn this bunch disappears - or what is the same, there was a second bunch of exactly the same size (as in Nîmes two piles of equal size actually "kill" each other). $y (y < x)y y y y y y$

Thus, the answer to the problem - it is XOR-sum of the numbers - coordinates all zeros in one-indexing.

**The decision of "twins"**. All the arguments given above remain true, except that the course "reset pile" is now a player does not. Ie if we take away from all the coordinates unit - then again, the game turns into an ordinary him.

Thus, the answer to the problem - it is XOR-sum of the numbers - the coordinates of all zeros in the 0-indexed.

## Northcott's game

**Condition**. There are board size  $n \times m$ :  $n$ rows and  $m$ columns. Each row consists of 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 not jumping over another chip (and not getting up on it). The player who can not make progress.

**Solution**. Firstly, it is clear that each of the  $n$  rows forms an independent board game. Therefore, the problem reduces to the analysis of the game on a single line, and the answer to the problem will be XOR-sum of Shpraga Grande for each of the rows.

Solving the problem for one line, we denote  $x$  the distance between the black and white chip (which can vary from zero to  $m - 2$ ). In one move, each player can either reduce  $x$  to some arbitrary value, or may increase it up to a certain value (the increase is not always available). Thus, this game - it's "them with increases", and, as we have seen, the increase in this game are useless. Therefore, the function Grande for one line - this is the distance  $x$ .

(It should be noted that such an argument is formally incomplete - as in "Nimes with increases" is assumed that the game is finite, and here are the rules of the game allow players to play indefinitely. However, an endless game can not take place at the optimum game - t. a. one player stands to increase the distance  $x$  (the price closer to the edge of the field), the other player closer to him, reducing the  $x$  back. Consequently, for optimal game player will not be able to make the moves increase indefinitely, so still described solution of the problem 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 form of the letter "G" (ie, a coherent figure of three cells that do not lie on a straight line). It is forbidden to put a figure so that it is crossed by at least one cell with some of the figures that have been set. The player who can not make a move.

**Solution**. Note that the formulation of a figure divides the entire 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, we can understand that whatever the configuration of the field, the main thing - just how many cells in this field. In fact, if the current field is  $x$  free of cells and we want to divide the field into two fields size  $y$  and  $z$  (where  $y + z + 3 = x$ ), it is always possible, i.e. you can always find an appropriate place for figurines.

Thus, our task becomes such: initially we have a handful of stones the size  $2n$ , and in one move we can throw out a handful of 3 stones and then break this pile on 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**. Dan directed acyclic graph. In some vertices of the graph are the chips. In one move, the player may take some piece and move it along any edge in the new vertex. The player who can not make a move.

Also takes place and a 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.

**Solution of the problem of the first embodiment**. Firstly, all the chips - independent of each other, so our task - to learn to look for Grundy function for one chip in the graph.

Given that the graph is acyclic, we can do it recursively: suppose we felt Grundy function for all descendants of the current node. Then the function Grande in the current top - it is mex from this set of numbers.

Thus, the solution of the problem is the following: for each vertex recursively count Grundy function if it was a feature in this top. After that, the answer to the problem will be XOR-sum of the Grande from those vertices of the graph, which by hypothesis are the chips.

**Solution of the problem of the second embodiment**. In fact, the second variant of the problem does not differ from the first one. In fact, if two chips are in one and the same vertex of the graph, the resulting XOR-sum values Grundy mutually destroy each other. Therefore, it is actually the same task.

## Implementation

From the standpoint of the 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  $O(c \log c)$  (where  $c$  - the number of arguments):

```
int mex(vector<int> a) {
    set<int> b(a.begin(), a.end());
    for (int i=0; ; ++i)
        if (!b.count(i))
            return i;
}
```

However, it is so difficult is an option **in linear time**, ie for  $O(c)$  where  $c$  - number of function arguments `mex`. Denoted by  $D$  a constant equal to the maximum possible value  $c$  (ie, the maximum degree of vertices in a graph of the game). In this case, the result of the function `mex` will not exceed  $D$ .

Consequently, when implementing enough to have an array of size  $D + 1$  (array global or static - the main thing that it was not created for each function call). When you call the function `mex`, we first note in all this array  $c$  of arguments (skipping those that are more  $D$  - those values are obviously not affect the result). Then passes through the array for we  $O(c)$  find the first element unmarked. Finally, at the end you can again go through all the arguments and reset the array back to them. Thus, we perform all actions for  $O(c)$ , which in practice may be considerably less than the maximum degree  $D$ .

```
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;
            break;
        }

    for (int i=0; i<c; ++i)
        if (a[i] <= D)
            used[a[i]] = false;

    return result;
}
```

Another option - use the technique "**numerical USED**". Ie do `used` not array of boolean variables and numbers ("Version"), and make a global variable indicating the number of the current version. When you enter the function `mex` we increase the current version number in the first cycle, we shall appear in the array `used` is not `true`, and the current version number. Finally, in the second cycle we simply compare the `used[i]` number with the current version - if they do not match, it means that the current number is not found in the array `a`. The third cycle (which vanishes before the array `used`) in such a decision is not necessary.

## Generalization of neem: These Moore ( $k$ -nim)

One of the interesting generalization of the usual neem was given by Moore (Moore) in 1910

**Condition**. There are  $n$ heaps of stones size  $a_i$ . Also contains a positive integer  $k$ . In one move, the player can reduce the size of one to  $k$ piles (ie now allows simultaneous moves in several piles at once). The player who can not make progress.

Obviously, when  $k = 1$  it becomes an ordinary Moore them.

**Solution**. The solution of this problem is amazingly simple. Record the size of each pile in the binary system. Then sum these numbers in  $k + 1$ -ary notation without hyphenation discharges. If you get a number zero, the current position is losing, anyway - winning (and of course it has to position a zero-valued).

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  $k + 1$ . If as a result of this sum for each bit turned zero, then the current position - losing, otherwise - winning.

**Proof**. As for Neem, the proof is to describe the strategies of the players: on the one hand, we show that out of the game with zero, we can only go to the game with a non-zero value, on the other hand - that out of the game with a nonzero value is the course of the game with a zero value.

Firstly, we show that out of the game with a zero value can only go into the game with a non-zero value. It is quite clear that if the sum modulo  $k + 1$  is zero, then after changing from one to  $k$ the bit we could not get over the amount of zero.

Second, we show how a zero-sum game go to a zero sum game. Let's take the bits in which a nonzero sum, in order from oldest to youngest.

Denoted by  $u$ the number of piles, which we have already started to change; originally  $u = 0$ . Note that in these  $u$ heaps, we can already put all the bits we want (because any of a handful that fall into the number of these  $u$ piles have decreased from the previous one, older, bits).

So, let us consider the current bit in which the sum modulo  $k + 1$ a nonzero. Denoted by  $s$ this amount, but that does not take into account those  $u$ piles, which we have already begun to change. Denoted by  $q$ the amount that can be obtained by putting these  $u$ piles current bit equal to one:

$$q = (s + u) \pmod{(k + 1)}$$

We have two options:

- If  $q \leq u$ .

Then we can manage only already selected  $u$ in groups: it is enough to  $k + 1 - s = u - q$ put them out of the current bit equal to one, and all the others - zero.

- If  $q > u$ .

In this case, contrary, we have chosen to deliver  $u$ piles current bit equal to zero. Then the sum of the current bit is equal to  $s > 0$ , and, therefore, among the unselected  $n - u$ heaps in the current bit has at least  $s$ units. Select some  $s$ piles of them, and reduce them to the current bit with one to zero.

As a result, the number of  $u$ unchanged heaps grow by  $s$ , and will be  $q \leq k$ .

Thus, we have shown how to select multiple variable piles and which bits should they change to the total number  $u$ never exceeded  $k$ .

Hence, we have proved that the desired transition from the non-zero sum in the state, there is a zero-sum, as required.

## "Him giveaway"

That it, we looked at all of this article - also called "normal Nimes" ("normal nim"). In contrast, there is also a "giveaway him" ("misère nim") - a player who has made the last move loses (and not win).

(By the way, seems to him like the board game - it is a more popular version of "giveaway" and not a

"normal" version)

**The decision** of the neem surprisingly simple: we will act in the same way as in the ordinary Nimes (ie calculate the XOR-sum of all sizes piles, 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 the size of all the piles are equal to one, then the winning / losing are reversed compared to the conventional Nimes.

Thus, the winning / losing neem "giveaway" is defined by the number of:

$$a_1 \oplus a_2 \oplus \dots \oplus a_n \oplus z,$$

where through the  $z$  designated Boolean variable equal to one if  $a_1 = a_2 = \dots = a_n = 1$ .

In view of this exception, the **optimal strategy** for the player in a winning position is determined as follows. Let us find a move which would have made the player, if he played it to normal. Now, if the move leads to a position in which the sizes of all the piles are equal to one (and thus before the course 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**. Note that the general theory Shpraga Grande belongs to the "normal" games, rather than to the games in the giveaway. However, it - one of those games for which the solution of the game "giveaway" is not much different from the decision a "normal" game. (By the way, the solution of neem "giveaway" was given the same Charles Bouton, who described the decision to "normal" neem.)

How can we explain such a strange pattern - that of winning / losing neem "giveaway" coincides with a winning / losing the "normal" neem almost always?

Consider some **things around** : ie, choose an arbitrary starting position and write down the moves of players until the end of the game. It is not difficult to understand that provided optimal play rivals - the game will end that will remain one bunch size 1, and the player will be forced to go there and play.

Therefore, in any game of two best players sooner or later there comes a **time** when the size of all non-empty piles are equal to one. Denoted by  $k$  the number of non-empty heaps at this point - then the current player this a win-win position if and only if  $k$  is even. Ie We have seen that in these cases the winning / losing neem "giveaway" **opposite** "normal" nimu.

Again, go back to the time when the first time in the game all the piles of steel size 1, and rolled one move ago - right before the situation turned out. We were in a situation that one has a bunch of size  $> 1$ , and all other handful (maybe there were zero pieces) - size 1. This position is obviously a win-win (since we really can always make such a move that left an odd number of piles of size 1, ie we give 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** with Nîmes "giveaway".

Further, if we continue to roll back on the game, we will come to the point where the game was two piles size  $> 1$ , three piles, etc. For all those of the winning / losing will also coincide with the "normal" Nimes - simply because we've got more than a handful of sizes  $> 1$ , all transitions are in a state with one or more heap size  $> 1$ - and for all of them, as we have showed nothing compared to the "normal" Nîmes **not changed** .

Thus, changes in Nimes "giveaway" affect only state where all the piles have a size equal to one - was to be proved.

## Tasks in the online judges

List of tasks in online judges, which can be solved using the Grande:

- [TIMUS # 1465 "Game of pawns"](#) [Difficulty: Low]
- [UVA # 11534 "Say Goodbye to Tic Tac-Toe-](#) [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**
-

# MAXimal

[home](#)

[algo](#)

[bookz](#)

[forum](#)

[about](#)

Added: 11 Jun 2008 11:20  
EDIT: 2 Dec 2010 10:29

## Johnson problem with one machine

It is the task of optimal treatment schedules  $n$  parts on a single machine, if  $i$  the item is processed Star on it for the time  $t_i$ , and for  $t$  seconds to wait before processing this part to pay the fine  $f_i(t)$ .

Thus, the problem is to find such a reordering parts, the following quantity (amount of the fine) is minimal. If we denote the  $\pi$  permutation details ( $\pi_1$ - number of the first workpiece,  $\pi_2$ - the second, etc.), the amount of the penalty  $f(\pi)$  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 the problem uniprocessor serve many applications.

### Contents [hide]

- Johnson problem with one machine
  - Solution of the problem in some special cases
    - The first special case: linear penalty function
    - The second special case: the exponential penalty function
    - The third special case: the same monotone functions fine
  - Theorem Livshits-Kladova

## Solution of the problem in some special cases

### The first special case: 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 numbers. Note that in these linear functions of the constant term is zero because otherwise to respond immediately, you can add the constant term, and solve the problem with zero constant term.

Fix some schedule - permutation  $\pi$ . Let us fix some number  $i = 1 \dots n - 1$ , and let the permutation  $\pi'$  is a permutation  $\pi$ , which traded  $i$ th and  $i + 1$ th elements. Let's see how much has changed in this fine:

$$F(\pi') - F(\pi) =$$

easy to understand that changes have occurred only with the  $i$ -th and  $i + 1$ -th terms:

$$\begin{aligned}
 &= c_{\pi'_i} \cdot \sum_{k=1}^{i-1} t_{\pi'_k} + c_{\pi'_{i+1}} \cdot \sum_{k=i}^i t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=i}^i t_{\pi_k} = \\
 &= c_{\pi_{i+1}} \cdot \sum_{k=1}^i t_{\pi'_k} + c_{\pi_i} \cdot \sum_{k=1}^i t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=1}^{i-1} t_{\pi_k} = \\
 &= c_{\pi_i} \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i}.
 \end{aligned}$$

Clearly, if the schedule  $\pi$  is optimal, then any change results in an increase in the fine (or maintaining the same value), so the optimal plan can write the condition:

$$\forall i = 1 \dots n-1 : c_{\pi_i} \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i} \geq 0.$$

Transforming, 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, **an optimum schedule** can be obtained by simply **sorting** all the details in relation  $c_i$  to  $t_i$  the reverse order.

It should be noted that we have received this algorithm so-called **commutation reception**: we tried to exchange the position of two adjacent elements schedules, calculate how much has changed in this fine, and hence derived an algorithm for finding the optimal schedule.

## The second special case: the exponential penalty function

Suppose now that the penalty function are of the form:

$$f_i(t) = c_i \cdot e^{\alpha \cdot t},$$

where all the numbers  $c_i$  are non-negative, the constant  $\alpha$  is positive.

Then, applying the same way commutes are welcome, it is easy to get the details necessary to sort in descending order of value:

$$v_i = \frac{1 - e^{\alpha \cdot t_i}}{c_i}.$$

## The third special case: the same monotone functions fine

In this case, it is believed that all  $f_i(t)$  coincide with some function  $\phi(t)$  that 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 technique is only applicable to the above three special cases, and only them, ie.:

- Linear case:  $f_i(t) = c_i \cdot t + d_i$  where  $c_i$ - non-negative constants
- The exponential case:  $f_i(t) = c_i \cdot e^{\alpha \cdot t} + d_i$  where  $c_i$  and  $\alpha$ - positive constants,
- Identical case:  $f_i(t) = \phi(t)$  where  $\phi$ - increasing function.

This theorem is proved under the assumption that the penalty function are sufficiently smooth (there are third derivatives).

In all three cases, the applicable commutes technique by which the desired optimal schedule can be found by simple sorting, therefore, for the time  $O(n \log n)$ .

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:21  
Edited: 15 Sep 2011 1:36

## Johnson problem with two machines

There are  $n$ two parts and the machine. Each item must first be processed on the first machine, and then - on the second. When this  $i$  item is processed Star machine for the first  $a_i$ time, and the second - after  $b_i$ time. Each machine at each time point can only operate in one piece.

Required to make such an order to supply details of machines to the final processing time of all the details would be minimal.

This problem is sometimes called the dual-processor maintenance tasks, or task Johnson (named SM Johnson, who in 1954 proposed an algorithm for its solution).

It is worth noting that when the number of tools greater than two, the task is NP-complete (as demonstrated by Gary (Garey) 1976).

### Contents [hide]

- Johnson problem with two machines
  - Construction of an algorithm
  - Implementation
  - Literature

## 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, since Details for the second machine are available only after the first treatment, and if there are several available for the second machine parts processing time will be equal to the sum of their  $b_i$  independent of their order - that is most advantageous to send to the second machine from parts, which previously has been treated on the other first machine.

Consider the procedure for submission of details on the machines, which coincides with their input order:  $1, 2, \dots, n$ .

Denoted by **the downtime** of the machine just before the second treatment th part (after processing th details). Our goal - **to minimize the total simple**  $x_i$   $i = 1$

$$F(x) = \sum x_i \rightarrow \min .$$

For the first part, we have:

$$x_1 = a_1 .$$

For the second - because it becomes ready to send the second time in the machine  $a_1 + a_2$ , and the machine is released in the second time  $x_1 + b_1$ , we have:

$$x_2 = \max((a_1 + a_2) - (x_1 + b_1), 0).$$

The third item is made available for the second time in the machine  $a_1 + a_2 + a_3$  and in the machine is released  $x_1 + b_1 + x_2 + b_2$ , so:

$$x_3 = \max((a_1 + a_2 + a_3) - (x_1 + b_1 + x_2 + b_2), 0).$$

Thus, for the general form  $x_i$  looks like this:

$$x_k = \max\left(\sum_{i=1}^k a_i - \sum_{i=1}^{k-1} b_i - \sum_{i=1}^{k-1} x_i, 0\right).$$

Now calculate **the total simple**  $F(x)$ . It is argued that it is of the form:

$$F(x) = \max_{k=1 \dots n} K_i,$$

where

$$K_i = \sum_{i=1}^k a_i - \sum_{i=1}^{k-1} b_i.$$

(At this can be seen by induction or consistently find an expression for the sum of the first two, three, and so on  $x_i$ .)

We now use the **commutation reception** : try to exchange any two adjacent elements  $j$  and  $j + 1$  and see how this will change when the total simple.

By type of function expressions for  $K_i$  it is clear that change only  $K_j$ , and  $K_{j+1}$ ; denote their new values through  $K'_j$  and  $K'_{j+1}$ .

Thus, the item to  $j$  the items discussed before  $j + 1$ , it is sufficient (but not necessary) to:

$$\max(K_j, K_{j+1}) \leq \max(K'_j, K'_{j+1}).$$

(Ie, we have ignored the others, has not changed, the arguments in the expression for the maximum  $F(x)$ , thereby obtaining a sufficient but not a necessary condition that the old  $F(x)$  is less than or equal to the new value)

Subtracting  $\sum_{i=1}^{j+1} a_i - \sum_{i=1}^{j-1} b_i$  from both sides of this inequality, we obtain:

$$\max(-a_{j+1}, -b_j) \leq \max(-b_{j+1}, -a_j),$$

or getting rid of negative numbers, we get:

$$\min(a_j, b_{j+1}) \leq \min(b_j, a_{j+1}).$$

Thus, we have received **the comparator**: sorting out the details on it, we have, according to the calculations given above, we arrive at the optimal order of parts in which it is impossible to interchange any two parts, improving the final time.

However, you can further **simplify the** sorting, if you look at the comparator with the other hand. In fact, he tells us that if a minimum of four numbers  $(a_j, a_{j+1}, b_j, b_{j+1})$  is achieved by an element of the array  $a$ , the corresponding item should go ahead, and if an element of the array  $b$ - that later. Thus, we obtain another form of algorithm: sort the details of the minimum  $(a_i, b_i)$  and if the current minimum is equal parts  $a_i$ , then this item should be processed first of the remaining, otherwise - the last remaining.

Anyway, it turns out that the problem of Johnson with two machines reduced to sorting items with a particular comparison function. Thus, the asymptotic behavior of the solution is  $O(n \log n)$ .

## Implementation

Implement a second embodiment of the algorithm described above when the parts are sorted out on a minimum  $(a_i, b_i)$ , and then sent at 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 items are stored in the form of patterns **item**, each of which contains the value  $a$  and  $b$  number of the original parts.

Details are sorted, then distributed to the list  $a$ (it's the details that were sent to the queue) and  $b$ (the ones that were sent to the end). After that, the two lists are merged (and the second list is taken in reverse order), and then found the order calculated the required minimum time supported by two variables  $t_1$ , and  $t_2$ - the liberation of the first and second machine, respectively.

## Literature

- SM Johnson. **Optimal two- and Three-stage Production Schedules with Setup Times included** [1954]
  - MR Garey. **The Complexity of flowshop and JobShop Scheduling** [1976]
-

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:22  
EDIT: 6 Dec 2010 14:43

## The optimal choice of job completion times for known and long-running

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 assignment. The process of performing any task can not be interrupted before it is completed. Required to make a schedule to carry out the largest number of jobs.

### Contents [hide]

- The optimal choice of job completion times for known and long-running
  - Solution
  - Implementation

### Solution

Algorithm for solving - **greedy** (greedy). Sort all tasks on their deadline, and consider them one by one in descending order deadline. Also create a place  $q$  in which we will gradually put jobs and benefit from the job queue with the lowest execution time (for example, you can use the set or priority\_queue). Initially  $q$  empty.

Let us consider  $i$  retracement job. First put it in  $q$ . Consider the length of time between the deadline for completion of  $i$  the  $i$ th task and deadline for completion of  $i - 1$  the  $i$ th job - a segment of a certain length  $T$ . Will be removed from the  $q$  job (in increasing order of the remaining time of their performance) and put on the performance of this segment until it fills the entire segment  $T$ . An important point - if at some point in time the next extracted from the structure of the task you can manage to partially fulfill a segment  $T$ , then we carry out this task in part - that as far as possible, ie, for  $T$  units of time, and the remaining part of the job is put back in  $q$ .

At the end of this algorithm, we choose the optimal solution (or at least one of several solutions). Asymptotics of the solution -  $O(n \log n)$ .

### Implementation

```

int n;
vector < pair<int,int> > a; // задания в виде пар (крайний срок, длительность)
... чтение n и 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] << ' ';
```

# MAXimal

[home](#)
[algo](#)
[bookz](#)
[forum](#)
[about](#)

 Added: 11 Jun 2008 11:25  
 EDIT: 14 Oct 2011 0:05

## Joseph task

Condition of the problem. Are natural  $n$  and  $k$ . The circle prescribed all the natural numbers from 1 to  $n$ . First, count the  $k$  number of retracement from the first, and remove it. Then, from a count  $k$  number and  $k$  removed -th, etc. The process stops when one number remains. Required to find this number.

The problem was posed by **Josephus** (Flavius Josephus) back in the 1st century (albeit in a slightly more narrow formulation: at  $k = 2$ ).

To solve this problem can be simulated. The simplest simulation will work  $O(n^2)$ . Using the [segment tree](#), can be made for modeling  $O(n \log n)$ .

### Contents [hide]

- Joseph task
  - Solution for  $O(n)$
  - Solution for  $O(k \log n)$
  - Analytical solution for  $k = 2$
  - Analytical solution for  $k > 2$

## Solution for $O(n)$

Try to find a pattern that expresses the answer to the problem  $J_{n,k}$  by solving previous problems.

By 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 following pattern :

$$\begin{aligned} J_{n,k} &= (J_{(n-1),k} + k - 1) \% n + 1 \\ J_{1,k} &= 1 \end{aligned}$$

Here are 1-indexed spoils elegance formula if numbered positions from scratch, you get a very visual 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, works for  $O(n)$  operations.

A simple **recursive implementation** (1-indexed):

```
int joseph (int n, int k) {
    return n>1 ? (joseph (n-1, k) + k - 1) % n + 1 : 1;
}
```

**Non-recursive 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  $k$ , you can come up with a better solution than that discussed above for the recursive solution  $O(n)$ . If  $k$  small, even intuitively clear that the algorithm makes a lot of unnecessary actions: major changes occur only when there is a modulo  $n$ , and up to this point the algorithm simply adds a few times to account number  $k$ . Accordingly, we can get rid of these unnecessary steps,

Partly arising from this difficulty lies in the fact that after the removal of these numbers, we obtain the problem with a smaller  $n$ , but the starting position is not in the first days, but somewhere else. Therefore, recursively calling itself the task with the new  $n$ , we must then carefully transfer the result of our numbering system of its own.

Also, it is necessary to separately examine the case when  $n$  will be less  $k$ - in this case, the above optimization degenerate into an infinite loop.

**Implementation** (for convenience in the 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;
        else res += res / (k - 1);
        return res;
    }
}

```

We estimate **the asymptotic behavior** of the algorithm. Immediately, we note that the case  $n < k$  understands our old solution that will work in this case over  $O(k)$ . Now consider the algorithm itself. In fact, at each iteration instead of its  $n$  numbers, we get about  $n \left(1 - \frac{1}{k}\right)$  numbers, so the total number of  $x$  iterations of the algorithm can be found approximately from the equation:

$$n \left(1 - \frac{1}{k}\right)^x = 1,$$

logarithm of it, we get:

$$\ln n + x \ln \left(1 - \frac{1}{k}\right) = 0,$$

$$x = -\frac{\ln n}{\ln \left(1 - \frac{1}{k}\right)},$$

using the expansion of the logarithm of the 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 it was placed and Josephus problem), the problem is solved much easier.

In the case of even  $n$  get that will delete all even numbers, and then left for the task  $\frac{n}{2}$ , then the answer is to  $n$  be obtained from the answer to  $\frac{n}{2}$  the multiplication by two and subtracting units (due to shift positions):

$$J_{2n,2} = 2J_{n,2} - 1$$

Similarly, in the case of odd  $n$  will delete all even numbers, then the first number, and will remain a task for  $\frac{n-1}{2}$ , and taking into account the shift positions get the second formula:

$$J_{2n+1,2} = 2J_{n,2} + 1$$

When implementing this can be used directly recursive dependency. This pattern can be converted into another form:  $J_{n,2}$  represent the sequence of all odd numbers, "Restarts" from the unit whenever  $n$  is a power of two. This can be written in a single 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 small  $k$  derive 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").

---

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 11 Jun 2008 11:27  
EDIT: 27 Dec 2008 16:39

## Game Fifteen: the existence of solutions

Recall that the game is a field 4on 4, on which there are 15chips, numbered from 1to 15, and one field is left blank. Required at each step of moving any piece on a free position, to come in the end to the next position:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	○

Fifteen game ("15 puzzle") was invented in 1880 by Noyes Chapman (Noyes Chapman).

### Contents [hide]

- Game Fifteen: the existence of solutions
  - The existence of solutions
  - Implementation
  - Proof

## The 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 there be given a 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 zero, and an empty cell indicates  $a_z = 0$ .

Consider the permutation:

$$a_1 a_2 \dots a_{z-1} a_{z+1} \dots a_{15} a_{16}$$

(ie, a permutation of numbers corresponding to the position on the board, without the zero element)

Denoted by  $N$ the number of inversions in this permutation (ie, the number of such elements  $a_i$ and  $a_j$ that  $i < j$ , but  $a_i > a_j$ ).

Next, let  $K$ - line number in which there is an empty element (ie, in our notation  $K = (z - 1) \text{ div } 4 + 1$ ).

Then, **a solution exists if and only if there  $N + K$  is even .**

## Implementation

Let us illustrate the above algorithm by using the 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])
        for (int j=0; j<i; ++j)
            if (a[j] > a[i])
                ++inv;
for (int i=0; i<16; ++i)
    if (a[i] == 0)
        inv += 1 + i / 4;

puts ((inv & 1) ? "No Solution" : "Solution Exists");

```

## Proof

Johnson (Johnson) in 1879 proved that if  $N + K$  is odd, then there is no solution, and Story (Story) in the same year showed that all items for which  $N + K$  even, have a solution.

However, both of these proofs were quite complex.

In 1999, Archer (Archer) proposed a much simpler proof (article can download it [here](#) ).

# MAXimal

[home](#)  
[algo](#)  
[bookz](#)  
[forum](#)  
[about](#)

Added: 17 Jul 2009 23:00  
 EDIT: 16 Aug 2009 1:59

## Wood Stern-Brock. Farey sequence

### Stern-tree Brokaw

Stern-tree Brokaw - is an elegant design that allows you to build the set of all non-negative fractions. It was independently discovered by the German mathematician Moritz Stern (Moritz Stern) in 1858 and the French watchmaker Achilles Brokaw (Achille Brocot) in 1861. However, according to some sources, this design was discovered by the ancient Greek scholar Eratosthenes More (Eratosthenes).

At **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, on each **subsequent** iteration taken this list of fractions and between each two adjacent fractions  $\frac{a}{b}$  and  $\frac{c}{d}$  inserted them **mediant**, ie fraction  $\frac{a+c}{b+d}$ .

Thus, the first set of the current iteration is:

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

The second:

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

In the third:

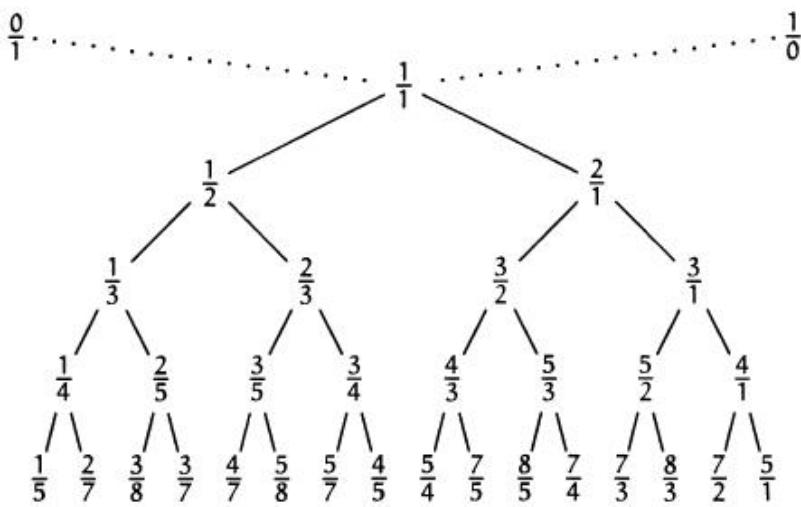
$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

Continuing this process to **infinity**, it is argued, can get a lot of non-negative fractions. Moreover, all the fractions obtained will be **different** (ie, the current set of each fraction does not occur more than once), **irreducible** (numerators and denominators will be obtained relatively prime). Finally, all fractions will be automatically **ordered** in ascending order. 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 a changing set). At the root of this tree is infinite fraction  $\frac{1}{1}$ , and the left and right of the tree are shot  $\frac{0}{1}$  and  $\frac{1}{0}$ . Any tree node has two sons, each of which is obtained as the mediant of its left and right ancestor:

### Contents [hide]

- Wood Stern-Brock. Farey sequence
  - Stern-tree Brokaw
    - Proof
    - An algorithm for constructing the tree
    - Search algorithm fraction
  - Farey sequence
  - Literature



## Proof

**Orderliness**. It is easy to prove: note that the mediant of two fractions is always between them, ie.:

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$$

provided that

$$\frac{a}{b} \leq \frac{c}{d}$$

Proved it's just bringing three fractions to a common denominator.

Since the zero-order iteration occurred, it will remain on each iteration.

**Uncancellable**. To do this, we show that for every iteration of any two adjacent fractions in the list  $\frac{a}{b}$  and  $\frac{c}{d}$  executed:

$$bc - ad = 1$$

Indeed, recalling the [Diophantine equation with two unknowns](#) ( $ax + by = c$ ), we obtain from this statement that  $\gcd(a, b) = \gcd(c, d) = 1$  we required.

So, we need to prove the truth of statements  $bc - ad = 1$  on any iteration. We prove it by induction also. At zero iteration this property holds (what is easy to see). Now let it was performed at the previous iteration, we show that it holds at the current iteration. To do this, we need to consider three fractions neighbors in the new list:

$$\frac{a}{b}, \frac{a+c}{b+d}, \frac{c}{d}$$

For these conditions take the form:

$$\begin{aligned} b(a+c) - a(b+d) &= 1, \\ c(b+d) - d(a+c) &= 1 \end{aligned}$$

However, the truth of these conditions is obvious, when the condition is true  $bc - ad = 1$ . Thus, indeed, this property holds on the current iteration, as required.

**The presence of all fractions**. The proof of this property is closely linked to the algorithm for finding the fractions in the tree Stern-Brocot. Given that the Stern-tree Brokaw all fractions in order, we find that for any tree node in its left subtree are a fraction smaller than her, and the right - most of it. Hence we obtain the obvious search algorithm and a shot in the tree-Stern Brokaw: first, we are at the root; compare our shot to shot, recorded in the current node: if our fraction less, then go to the

left subtree if our shot more - go to the right, and if it is the same - the fraction found, the search is completed.

To prove that the infinite tree Stern-Brokaw contains all the fractions, it suffices to show that the search algorithm fractions completed in a finite number of steps for any given shot. This algorithm can be understood as follows: we have the current segment  $\left[ \frac{a}{b}; \frac{c}{d} \right]$  in which we are looking for our shot  $\frac{x}{y}$ . Initially  $\frac{a}{b} = \frac{0}{1}$ ,  $\frac{c}{d} = \frac{1}{0}$ . At each step, a fraction  $\frac{x}{y}$  compared to the mediant endpoints, ie, with  $\frac{a+c}{b+d}$ , and depending on that we either stop the search, or go to the left or right side of the line. If the search algorithm fraction worked indefinitely, then 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:

$$\begin{aligned} bx - ay &\geq 1, \\ cy - dx &\geq 1 \end{aligned}$$

(We have used the fact that they are integers, so of  $> 0$  should  $\geq 1$ )

Then, multiplying the first on  $c + d$ , and the second - on  $a + b$ , and adding them, we obtain:

$$(c + d)(bx - ay) + (a + b)(cy - dx) \geq a + b + c + d$$

The brackets on the left and given that  $bc - ad = 1$  (see. the proof of the previous properties), we finally obtain:

$$x + y \geq a + b + c + d$$

And since at each iteration at least one of the variables  $a, b, c, d$  is strictly increasing, then the process of finding the fraction  $\frac{x}{y}$  will contain no more  $x + y$  iterations, as required.

## An algorithm for constructing the tree

To build any subtree of the Stern-Brock, is sufficient to know only the left and right of their ancestors. Initially, in the first level, the left is the ancestor  $\frac{0}{1}$ , and right -  $\frac{1}{0}$ . They can be used to calculate the fraction in the current node, and then start from the left and right sons (left to his son pass itself as an ancestor of the right, and right child - as an ancestor of the left).

The pseudo-code of the procedure, trying to build all the 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;
    ... вывод текущей дроби x/y на уровне дерева 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 tree Stern-Brokaw contains all fractions repeat it here. This algorithm - in fact, binary search, or search algorithm specified 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 top. If they match, then the process stops - we found shot in the tree. Otherwise, if the fraction is less than our fraction in the current node, then move to the left child, otherwise - to the right.

As was shown in the property that tree Stern-Brokaw contains all non-negative fractions, fractions

when searching  $\frac{x}{y}$  algorithm will make no more  $x + y$  iterations.

We give an implementation that returns the path to the top, containing a predetermined fraction  $\frac{x}{y}$ , returning it 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 and a fraction  $\frac{1}{1}$ ). In fact, such a sequence of characters, creatures, and are uniquely determined by any non-negative fraction, called the **Stern-number 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 numbers in base Stern-Brokaw will meet endless sequence of characters; if you know some accuracy given in advance, we can restrict some prefix of the infinite sequence. In the course of this endless search irrational fraction in the Stern-tree algorithm will Brokaw every time there is a simple fraction (with gradually increasing denominators), which provides a better approximation of the irrational number (it is important to use just to watch the technique, and in this regard, and Achilles Brokaw opened a tree).

## Farey sequence

Farey sequence of order  $n$  is the set of all irreducible fractions between 0 and 1, the denominators do not exceed  $n$ , with fractions sorted in ascending order.

This sequence is named after the English geologist John Farey (John Farey), who in 1816 tried to prove that in some Farey fraction is MEDIANT any two adjacent. As far as we know, his proof was wrong, and correct proof offered later Cauchy (Cauchy). However, even in 1802 the mathematician Haros (Haros) in one of his works came to almost the same results.

Farey sequence have their own and plenty of interesting properties, but the most obvious their **relationship with tree Stern-Brokaw**: in fact, Farey sequence is obtained by removing some branches from a tree. Or we can say that for Farey sequence have to take the set of fractions obtained in the construction of wood-Stern Brokaw on the infinite iteration, and leave this set only fractions with denominators not exceeding  $n$  numerators and not exceeding denominators.

Of the algorithm for constructing the tree-Stern Brokaw follows a similar **algorithm** for sequences of Farey. At zero iteration will include a set of only fractions  $\frac{0}{1}$  and  $\frac{1}{1}$ . At each next iteration we between every two adjacent fractions insert their mediant, if it does not exceed the denominator  $n$ . Sooner or later in the set will no longer be any changes, and the process can be stopped - we found the required sequence Farey.

We calculate the **length of the** sequence Farey. Farey sequence of order  $n$  contains all the elements of a sequence Farey order  $n - 1$ , as well as all irreducible fractions with denominators equal  $n$ , but this number is known as well  $\phi(n)$ . Thus, the length of  $L_n$  the sequence Farey order  $n$  given 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, Oren Patashnik. **Concrete Mathematics. The foundation of computer science** [1998]
-

# MAXimal

home  
algo  
bookz  
forum  
about

Added: 23 Aug 2011 12:40  
EDIT: 15 Jul 2014 22:21

## Search subsegment array with maximum / minimum amount

Here we consider the problem of finding subsegment array with a maximum amount ("maximum subarray problem" in English), as well as some of its variations (including an algorithm for solving this problem in version online -- described algorithm - KADR (Yaroslav Tverdohleb)).

### Contents [hide]

- Search subsegment array with maximum / minimum amount
  - Statement of the Problem
  - Algorithm 1
    - Description of the algorithm
    - Implementation
  - Algorithm 2
    - Description of the algorithm
    - Implementation
  - Related tasks
    - Search maximum / minimum subsegment with restrictions
    - The two-dimensional case of the problem: the search for the maximum / minimum submatrix
    - Search subsegment with maximum / minimum average amount
    - Solution of the problem online

## Statement of the Problem

Given an array of numbers  $a[1 \dots n]$ . It is necessary to find a subsegment  $a[l \dots r]$  that amount on it is maximal :

$$\max_{1 \leq l \leq r \leq n} \sum_{i=l}^r a[i].$$

For example, if all of the array  $a[]$  would be non-negative, then the response would be to take the whole array. The decision is non-trivial when the array may contain both positive and negative numbers.

It is clear that the problem of finding a **minimum** subsegment - essentially the same, you just change the signs of all the numbers are reversed.

## Algorithm 1

Here we consider the almost obvious algorithm. (Next we will look at 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 very simple.

We introduce for convenience **the notation** :  $s[i] = \sum_{j=1}^i a[j]$ . I.e array  $s[i]$ - an array of partial sums of the array  $a[]$ . Also set the value  $s[0] = 0$ .

We now **iterate** index  $r = 1 \dots n$ , and learn for each current value  $r$  quickly find the optimum  $l$ , at which the maximum amount for subsegments  $[l; r]$ .

Formally, this means that we need for the current  $r$  to find a  $l$  (not exceeding  $r$ ) that the value  $s[r] - s[l - 1]$  was the highest. After the trivial transformation we find that we need to find in the array  $s[]$  at least in the interval  $[0; r - 1]$ .

Hence we immediately obtain an algorithm for solving: we're just going to keep, where the array  $s[]$  is the current minimum. Using this minimum, we are for  $O(1)$  the current find the optimal index  $l$ , and in the transition from the current index  $r$  to the next, we simply update this minimum.

Obviously, this algorithm works for  $O(n)$  and is asymptotically optimal.

## Implementation

For implementation, we do not even need to explicitly store an array of partial sums  $s[]$  - from him, we will be required only to the current element.

The implementation is given in 0-indexed arrays rather than one numbering as described above.

We give first a solution that is simple numerical answer, not finding the desired index segment:

```
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);
}
```

Now we present the full version of the decision, which in parallel with the numerical solution of the desired length of the border is:

```
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 look at a different algorithm. It's a little difficult to understand, but it is more elegant than the above should and realized a little shorter. This algorithm has been proposed Kadan Jay (Jay Kadane) in 1984

### Description of the algorithm

Himself **algorithm** is as follows. Let's go through the array and store in a variable  $s$  current partial sum. If at any point  $s$  will be negative, 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 be the answer to the problem.

We prove this algorithm.

In fact, consider the first point in time, when the amount  $s$  was negative. This means that, starting from a zero partial sum, we have eventually come 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, it is not enough to prove the algorithm. In the algorithm we actually restrict to answering only those segments that start immediately after the places where happened  $s < 0$ .

But, in fact, consider an arbitrary interval  $[l; r]$ , with  $l$  is not in a "critical" positions (ie  $l > p + 1$ , where  $p$  - the last such position in which  $s < 0$ ). Since the last critical position is strictly earlier than  $l - 1$  it turns out that the sum of  $a[p + 1 \dots l - 1]$  non-negative. This means that by moving  $l$  into position  $p + 1$ , we will increase the response or, in extreme cases, do not change it.

One way or another, but it turns out that actually finding the answer can be limited only by segments, starting immediately after the position in which one

finds  $s < 0$ . This proves the correctness of the algorithm.

## Implementation

As in Algorithm 1, we give first a simplified implementation that searches only numerical answer, not finding the boundaries of the desired segment:

```
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 decision to the maintenance of indexes, the boundaries of the desired 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 maximum / minimum subsegment with restrictions

If in the problem to the desired length of  $[l; r]$  additional restrictions (for example, the length  $r - l + 1$  of the interval should be within the prescribed limits), then the algorithm is likely to be easily generalized to these cases - one way or

another, the problem will continue to be to find a minimum in the array  $s[]$  at No additional restrictions.

## The two-dimensional case of the problem: the search for the maximum / minimum submatrix

Described in this article the problem naturally generalizes to higher dimensions. For example, in the two-dimensional case, it turns into a search submatrix  $[l_1 \dots r_1; l_2 \dots r_2]$  given matrix, which has the maximum sum of the numbers in it.

From the above solutions for one-dimensional case **is easy to get** a solution for  $O(n^3)$ : brute over  $l_1$  and  $r_1$ , and count with an array of sums  $l_1$  for  $r_1$  each row of the matrix; we came to the one-dimensional problem of finding the index  $l_2$  and  $r_2$  in this array, which can already be solved in linear time.

**Faster** algorithms for solving this problem though known, but they are not much faster  $O(n^3)$ , and at the same time very complicated (so complex that the hidden constant, many of them are inferior to the trivial algorithm for any reasonable restrictions). Apparently, the best known algorithms for work  $O\left(n^3 \frac{\log^3 \log n}{\log^2 n}\right)$  (T. Chan 2007 "More algorithms for all-pairs shortest paths in weighted graphs").

This algorithm Chan, as well as many other results in this area actually describe the **rapid multiplication of** matrices (where the matrix multiplication is meant modified multiplication instead of addition using a minimum, and instead of multiplication - 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 this task, in turn - is reduced to such a matrix multiplication.

## Search subsegment with maximum / minimum average amount

This problem lies in the fact that it is necessary to find a segment  $[l; r]$  that the average value on it maximized:

$$\max_{l \leq r} \frac{1}{r - l + 1} \sum_{i=l}^r a[i].$$

Of course, if the desired interval  $[l; r]$  for the condition is not imposed on other conditions, the solution will always be a segment of length 1 at the maximum-array. The task is meaningful only if there are **additional constraints** (for example, the length of the desired interval is bounded below).

In this case, we apply **the standard technique** when dealing with the problems of the average value: will select the required maximum average value of **a binary search**.

To do this, we need to learn how to solve a subproblem: given a number  $x$ , and it is necessary to check whether there is a subsegment of the array  $a[]$  (of course,

satisfy all additional constraints of the problem), on which the mean value is greater  $x$ .

To solve this subproblem, subtract  $x$  from each element of the array  $a[]$ . Then our subtask actually turns into this: whether or not a given array subsegment positive amount. And this task we already know how to solve.

Thus, we have obtained for the asymptotic behavior of the solution  $O(T(n) \log W)$ , where  $W$  - the required accuracy  $T(n)$ - time solutions for subproblems array length  $n$ (which may vary depending on the specific additional restrictions).

## Solution of the problem online

Condition of the problem is this: given an array of  $n$  integers, and are given a number  $L$ . Receives requests species  $(l, r)$ , and in response to the request you want to find subsegment segment  $[l; r]$  of length not less than  $L$  the maximum possible arithmetic mean.

Algorithm for solving this problem is quite complicated. Author of this algorithm - KADR (Yaroslav Tverdohleb) - [the algorithm described in his forum post](#) .

---