

ALGORITHMS

TRANSLATIONS AND COMMUNITY ADDITIONS

Maxim “e-maxx” Ivanov (e-maxx.ru)

Various contributors (e-maxx-eng)

Contents

I	Algebra	1
1	Fundamentals	2
1.1	Binary Exponentiation	2
1.2	Euclidean algorithm for computing the greatest common divisor	8
1.3	Extended Euclidean Algorithm	11
1.4	Linear Diophantine Equation	13
1.5	Fibonacci Numbers	18
2	Prime numbers	22
2.1	Sieve of Eratosthenes	22
2.2	Sieve of Eratosthenes Having Linear Time Complexity	28
3	Number-theoretic functions	31
3.1	Euler's totient function	31
3.2	Number of divisors / sum of divisors	35
4	Modular arithmetic	37
4.1	Modular Multiplicative Inverse	37
4.2	Linear Congruence Equation	40
4.3	Chinese Remainder Theorem	42
4.4	Factorial modulo p in $O(p \log n)$	46
4.5	Discrete Root	48
4.6	Primitive Root	51
4.7	Discrete Logarithm	54
5	Number systems	57
5.1	Balanced Ternary	57
5.2	Gray code	60
6	Miscellaneous	62
6.1	Submask Enumeration	62
6.2	Arbitrary-Precision Arithmetic	64
6.3	Fast Fourier transform	69

II	Data Structures	83
1	Fundamentals	84
1.1	Minimum stack / Minimum queue	84
1.2	Sparse Table	88
2	Trees	92
2.1	Disjoint Set Union	92
2.2	Fenwick Tree	105
2.3	Sqrt Decomposition	110
2.4	Segment Tree	117
2.5	Treap (Cartesian tree)	141
2.6	Sqrt Tree	149
2.7	Randomized Heap	158
3	Advanced	161
3.1	Deleting from a data structure in $O(T(n) \log n)$	161
III	Dynamic Programming	165
1	DP optimizations	166
1.1	Divide and Conquer DP	166
2	Tasks	168
2.1	Dynamic Programming on Broken Profile. Problem “Parquet” .	168
2.2	Finding the largest zero submatrix	170
IV	String Processing	173
1	Fundamentals	174
1.1	String Hashing	174
1.2	Rabin-Karp Algorithm for string matching	179
1.3	Prefix function. Knuth–Morris–Pratt algorithm	180
1.4	Z-function and its calculation	189
1.5	Suffix Array	195
1.6	Aho-Corasick algorithm	204
2	Advanced	210
2.1	Suffix Tree. Ukkonen’s Algorithm	210
2.2	Suffix Automaton	215
2.3	Lyndon factorization	234
3	Tasks	237
3.1	Expression parsing	237
3.2	Finding all sub-palindromes in $O(N)$	243
3.3	Finding repetitions	247

V	Linear Algebra	253
1	Matrices	254
1.1	Gauss method for solving system of linear equations	254
1.2	Calculating the determinant of a matrix by Gauss	260
1.3	Calculating the determinant using Kraut method in $O(N^3)$. . .	262
1.4	Finding the rank of a matrix	265
VI	Combinatorics	267
1	Fundamentals	268
1.1	Finding Power of Factorial Divisor	268
1.2	Binomial Coefficients	270
1.3	Catalan Numbers	276
2	Techniques	279
2.1	The Inclusion-Exclusion Principle	279
2.2	Burnside's lemma / Pólya enumeration theorem	292
2.3	Stars and bars	299
2.4	Generating all K -combinations	300
3	Tasks	303
3.1	Placing Bishops on a Chessboard	303
3.2	Balanced bracket sequences	305
3.3	Counting labeled graphs	311
VII	Numerical Methods	313
1	Search	314
1.1	Ternary Search	314
1.2	Newton's method for finding roots	317
2	Integration	320
2.1	Integration by Simpson's formula	320
VIII	Geometry	322
1	Elementary operations	323
1.1	Basic Geometry	323
1.2	Finding the equation of a line for a segment	332
1.3	Intersection Point of Lines	334
1.4	Check if two segments intersect	336
1.5	Finding Intersection of Two Segments	337
1.6	Circle-Line Intersection	340
1.7	Circle-Circle Intersection	342

1.8	Length of the union of segments	343
2	Polygons	344
2.1	Oriented area of a triangle	344
2.2	Finding area of simple polygon in $O(N)$	346
2.3	Check if point belongs to the convex polygon in $O(\log N)$	347
2.4	Pick's Theorem	350
2.5	Lattice points inside non-lattice polygon	352
3	Convex hull	355
3.1	Convex Hull construction using Graham's Scan	355
3.2	Convex hull trick and Li Chao tree	358
4	Sweep-line	363
4.1	Search for a pair of intersecting segments	363
4.2	Point location in $O(\log n)$	369
5	Miscellaneous	376
5.1	Finding the nearest pair of points	376
5.2	Delaunay triangulation and Voronoi diagram	382
IX	Graphs	389
1	Graph traversal	390
1.1	Breadth-first search	390
1.2	Depth First Search	394
2	Connected components, bridges, articulations points	398
2.1	Search for connected components in a graph	398
2.2	Finding bridges in a graph in $O(N + M)$	400
2.3	Finding Bridges Online	403
2.4	Finding articulation points in a graph in $O(N + M)$	410
2.5	Finding strongly connected components – Building condensation graph	413
3	Single-source shortest paths	417
3.1	Dijkstra Algorithm	417
3.2	Dijkstra on sparse graphs	423
3.3	Bellman-Ford Algorithm	427
3.4	0-1 BFS	434
3.5	D'Esopo-Pape algorithm	436
4	All-pairs shortest paths	438
4.1	Floyd-Warshall Algorithm	438
4.2	Number of paths of fixed length / Shortest paths of fixed length	443

5	Spanning trees	446
5.1	Minimum spanning tree - Prim's algorithm	446
5.2	Minimum spanning tree - Kruskal's algorithm	452
5.3	Minimum spanning tree - Kruskal with Disjoint Set Union	456
5.4	Kirchhoff's theorem. Finding the number of spanning trees . . .	458
5.5	Prüfer code	459
6	Cycles	466
6.1	Checking a graph for acyclicity and finding a cycle in $O(M)$. . .	466
6.2	Finding a negative cycle in the graph	468
6.3	Finding the Eulerian path in $O(M)$	471
7	Lowest common ancestor	475
7.1	Lowest Common Ancestor - $O(\sqrt{N})$ and $O(\log N)$ with $O(N)$ preprocessing	475
7.2	Lowest Common Ancestor - Binary Lifting	479
7.3	Lowest Common Ancestor - Farach-Colton and Bender Algorithm	481
7.4	Solve RMQ (Range Minimum Query) by finding LCA (Lowest Common Ancestor)	486
7.5	Lowest Common Ancestor - Tarjan's off-line algorithm	490
8	Flows and related problems	492
8.1	Maximum flow - Ford-Fulkerson and Edmonds-Karp	492
8.2	Maximum flow - Push-relabel algorithm	501
8.3	Maximum flow - Push-relabel method improved	505
8.4	Maximum flow - Dinic's algorithm	507
8.5	Maximum flow - MPM algorithm	511
8.6	Flows with demands	516
8.7	Minimum-cost flow - Successive shortest path algorithm	518
8.8	Solving assignment problem using min-cost-flow	522
9	Matchings and related problems	525
9.1	Check whether a graph is bipartite	525
10	Miscellaneous	527
10.1	Topological Sorting	527
10.2	Edge connectivity / Vertex connectivity	529
10.3	Paint the edges of the tree	532
10.4	2 - SAT	537
10.5	Heavy-light decomposition	541
X	Miscellaneous	546
1	Sequences	547
1.1	Range Minimum Query	547
1.2	Longest increasing subsequence	549

1.3	Search the subsegment with the maximum/minimum sum	556
1.4	K-th order statistic in $O(N)$	561
2	Game Theory	563
2.1	Games on arbitrary graphs	563
3	Schedules	568
3.1	Scheduling jobs on one machine	568
3.2	Scheduling jobs on two machines	570
3.3	Optimal schedule of jobs given their deadlines and durations . .	573
4	Miscellaneous	575
4.1	Josephus Problem	575
4.2	15 Puzzle Game: Existence Of The Solution	579
4.3	The Stern-Brocot tree and Farey sequences	581

Part I

Algebra

Chapter 1

Fundamentals

1.1 Binary Exponentiation

Binary exponentiation (also known as exponentiation by squaring) is a trick which allows to calculate a^n using only $O(\log n)$ multiplications (instead of $O(n)$ multiplications required by naive approach).

It also has important applications in many tasks unrelated to arithmetic, since it can be used with any operations that have the property of **associativity**:

$$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$$

Most obviously this applies to modular multiplication, to multiplication of matrices and to other problems which we will discuss below.

1.1.1 Algorithm

Raising a to the power of n is expressed naively as multiplication by a done $n - 1$ times: $a^n = a \cdot a \cdot \dots \cdot a$. However, this approach is not practical for large a or n .

$$a^{b+c} = a^b \cdot a^c \text{ and } a^{2b} = a^b \cdot a^b = (a^b)^2.$$

The idea of binary exponentiation is, that we split the work using the binary representation of the exponent.

Let's write n in base 2, for example:

$$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$$

Since the number n has exactly $\lfloor \log_2 n \rfloor + 1$ digits in base 2, we only need to perform $O(\log n)$ multiplications, if we know the powers $a^1, a^2, a^4, a^8, \dots, a^{\lfloor \log n \rfloor}$.

So we only need to know a fast way to compute those. Luckily this is very easy, since an element in the sequence is just the square of the previous element.

$$3^1 = 3$$

$$3^2 = (3^1)^2 = 3^2 = 9$$

$$3^4 = (3^2)^2 = 9^2 = 81$$

$$3^8 = (3^4)^2 = 81^2 = 6561$$

So to get the final answer for 3^{13} , we only need to multiply three of them (skipping 3^2 because the corresponding bit in n is not set): $3^{13} = 6561 \cdot 81 \cdot 3 = 1594323$

The final complexity of this algorithm is $O(\log n)$: we have to compute $\log n$ powers of a , and then have to do at most $\log n$ multiplications to get the final answer from them.

The following recursive approach expresses the same idea:

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

1.1.2 Implementation

First the recursive approach, which is a direct translation of the recursive formula:

```
long long binpow(long long a, long long b) {
    if (b == 0)
        return 1;
    long long res = binpow(a, b / 2);
    if (b % 2)
        return res * res * a;
    else
        return res * res;
}
```

The second approach accomplishes the same task without recursion. It computes all the powers in a loop, and multiplies the ones with the corresponding set bit in n . Although the complexity of both approaches is identical, this approach will be faster in practice since we have the overhead of the recursive calls.

```
long long binpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
}
```

```

    }
    return res;
}

```

1.1.3 Applications

Effective computation of large exponents modulo a number

Problem: Compute $x^n \bmod m$. This is a very common operation. For instance it is used in computing the [modular multiplicative inverse](#).

Solution: Since we know that the module operator doesn't interfere with multiplications ($a \cdot b \equiv (a \bmod m) \cdot (b \bmod m) \pmod m$), we can directly use the same code, and just replace every multiplication with a modular multiplication:

```

long long binpow(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}

```

Note: If m is a prime number we can speed up a bit this algorithm by calculating $x^{n \bmod (m-1)}$ instead of x^n . This follows directly from Fermat's little theorem.

Effective computation of Fibonacci numbers

Problem: Compute n -th Fibonacci number F_n .

Solution: For more details, see the [Fibonacci Number article](#). We will only go through an overview of the algorithm. To compute the next Fibonacci number, only the two previous ones are needed, as $F_n = F_{n-1} + F_{n-2}$. We can build a 2×2 matrix that describes this transformation: the transition from F_i and F_{i+1} to F_{i+1} and F_{i+2} . For example, applying this transformation to the pair F_0 and F_1 would change it into F_1 and F_2 . Therefore, we can raise this transformation matrix to the n -th power to find F_n in time complexity $O(\log n)$.

Applying a permutation k times

Problem: You are given a sequence of length n . Apply to it a given permutation k times.

Solution: Simply raise the permutation to k -th power using binary exponentiation, and then apply it to the sequence. This will give you a time complexity of $O(n \log k)$.

Note: This task can be solved more efficiently in linear time by building the permutation graph and considering each cycle independently. You could

then compute k modulo the size of the cycle and find the final position for each number which is part of this cycle.

Fast application of a set of geometric operations to a set of points

Problem: Given n points p_i , apply m transformations to each of these points. Each transformation can be a shift, a scaling or a rotation around a given axis by a given angle. There is also a “loop” operation which applies a given list of transformations k times (“loop” operations can be nested). You should apply all transformations faster than $O(n \cdot \text{length})$, where length is the total number of transformations to be applied (after unrolling “loop” operations).

Solution: Let’s look at how the different types of transformations change the coordinates:

- Shift operation: adds a different constant to each of the coordinates.
- Scaling operation: multiplies each of the coordinates by a different constant.
- Rotation operation: the transformation is more complicated (we won’t go in details here), but each of the new coordinates still can be represented as a linear combination of the old ones.

As you can see, each of the transformations can be represented as a linear operation on the coordinates. Thus, a transformation can be written as a 4×4 matrix of the form:

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

that, when multiplied by a vector with the old coordinates and an unit gives a new vector with the new coordinates and an unit:

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

(Why introduce a fictitious fourth coordinate, you ask? Without this, it would not be possible to implement the shift operation, as it requires us to add a constant to the coordinates. Without the fictitious coordinates, we would only be able to apply a linear combination to the coordinates, not being able to add a constant.)

Here are some examples of how transformations are represented in matrix form:

- Shift operation: shift x coordinate by 5, y coordinate by 7 and z coordinate

by 9.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 7 & 9 & 1 \end{pmatrix}$$

- Scaling operation: scale the x coordinate by 10 and the other two by 5.

$$\begin{pmatrix} 10 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotation operation: rotate θ degrees around the x axis following the right-hand rule (counter-clockwise direction).

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now, once every transformation is described as a matrix, the sequence of transformations can be described as a product of these matrices, and a “loop” of k repetitions can be described as the matrix raised to the power of k (which can be calculated using binary exponentiation in $O(\log k)$). This way, the matrix which represents all transformations can be calculated first in $O(m \log k)$, and then it can be applied to each of the n points in $O(n)$ for a total complexity of $O(n + m \log k)$.

Number of paths of length k in a graph

Problem: Given a directed unweighted graph of n vertices, find the number of paths of length k from any vertex u to any other vertex v .

Solution: This problem is considered in more detail in [a separate article](#). The algorithm consists of raising the adjacency matrix M of the graph (a matrix where $m_{ij} = 1$ if there is an edge from i to j , or 0 otherwise) to the k -th power. Now m_{ij} will be the number of paths of length k from i to j . The time complexity of this solution is $O(n^3 \log k)$.

Note: In that same article, another variation of this problem is considered: when the edges are weighted and it is required to find the minimum weight path containing exactly k edges. As shown in that article, this problem is also solved by exponentiation of the adjacency matrix. The matrix would have the weight of the edge from i to j , or ∞ if there is no such edge. Instead of the usual operation of multiplying two matrices, a modified one should be used: instead of multiplication, both values are added, and instead of a summation, a minimum is taken. That is: $result_{ij} = \min_{1 \leq k \leq n} (a_{ik} + b_{kj})$.

Variation of binary exponentiation: multiplying two numbers modulo m

Problem: Multiply two numbers a and b modulo m . a and b fit in the built-in data types, but their product is too big to fit in a 64-bit integer. The idea is to compute $a \cdot b \pmod{m}$ without using bignum arithmetics.

Solution: We simply apply the binary construction algorithm described above, only performing additions instead of multiplications. In other words, we have “expanded” the multiplication of two numbers to $O(\log m)$ operations of addition and multiplication by two (which, in essence, is an addition).

$$a \cdot b = \begin{cases} 0 & \text{if } a = 0 \\ 2 \cdot \frac{a}{2} \cdot b & \text{if } a > 0 \text{ and } a \text{ even} \\ 2 \cdot \frac{a-1}{2} \cdot b + b & \text{if } a > 0 \text{ and } a \text{ odd} \end{cases}$$

Note: You can solve this task in a different way by using floating-point operations. First compute the expression $\frac{a \cdot b}{m}$ using floating-point numbers and cast it to an unsigned integer q . Subtract $q \cdot m$ from $a \cdot b$ using unsigned integer arithmetics and take it modulo m to find the answer. This solution looks rather unreliable, but it is very fast, and very easy to implement. See [here](#) for more information.

1.1.4 Practice Problems

- [UVa 1230 - MODEX](#)
- [UVa 374 - Big Mod](#)
- [UVa 11029 - Leading and Trailing](#)
- [Codeforces - Parking Lot](#)
- [SPOJ - The last digit](#)
- [SPOJ - Locker](#)
- [LA - 3722 Jewel-eating Monsters](#)
- [SPOJ - Just add it](#)

1.2 Euclidean algorithm for computing the greatest common divisor

Given two non-negative integers a and b , we have to find their **GCD** (greatest common divisor), i.e. the largest number which is a divisor of both a and b . It's commonly denoted by $\gcd(a, b)$. Mathematically it is defined as:

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

(here the symbol “ $|$ ” denotes divisibility, i.e. “ $k | a$ ” means “ k divides a ”)

When one of the numbers is zero, while the other is non-zero, their greatest common divisor, by definition, is the second number. When both numbers are zero, their greatest common divisor is undefined (it can be any arbitrarily large number), but we can define it to be zero. Which gives us a simple rule: if one of the numbers is zero, the greatest common divisor is the other number.

The Euclidean algorithm, discussed below, allows to find the greatest common divisor of two numbers a and b in $O(\log \min(a, b))$.

The algorithm was first described in Euclid's “Elements” (circa 300 BC), but it is possible that the algorithm has even earlier origins.

1.2.1 Algorithm

The algorithm is extremely simple:

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

1.2.2 Implementation

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

Using the ternary operator in C++, we can write it as a one-liner.

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

And finally, here is a non-recursive implementation:

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

1.2.3 Correctness Proof

First, notice that in each iteration of the Euclidean algorithm the second argument strictly decreases, therefore (since the arguments are always non-negative) the algorithm will always terminate.

For the proof of correctness, we need to show that $\gcd(a, b) = \gcd(b, a \bmod b)$ for all $a \geq 0, b > 0$.

We will show that the value on the left side of the equation divides the value on the right side and vice versa. Obviously, this would mean that the left and right sides are equal, which will prove Euclid's algorithm.

Let $d = \gcd(a, b)$. Then by definition $d \mid a$ and $d \mid b$.

Now let's represent the remainder of the division of a by b as follows:

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

From this it follows that $d \mid (a \bmod b)$, which means we have the system of divisibilities:

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

Now we use the fact that for any three numbers p, q, r , if $p \mid q$ and $p \mid r$ then $p \mid \gcd(q, r)$. In our case, we get:

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

Thus we have shown that the left side of the original equation divides the right. The second half of the proof is similar.

1.2.4 Time Complexity

The running time of the algorithm is estimated by Lamé's theorem, which establishes a surprising connection between the Euclidean algorithm and the Fibonacci sequence:

If $a > b \geq 1$ and $b < F_n$ for some n , the Euclidean algorithm performs at most $n - 2$ recursive calls.

Moreover, it is possible to show that the upper bound of this theorem is optimal. When $a = F_n$ and $b = F_{n-1}$, $\gcd(a, b)$ will perform exactly $n - 2$ recursive calls. In other words, consecutive Fibonacci numbers are the worst case input for Euclid's algorithm.

Given that Fibonacci numbers grow exponentially, we get that the Euclidean algorithm works in $O(\log \min(a, b))$.

1.2.5 Least common multiple

Calculating the least common multiple (commonly denoted **LCM**) can be reduced to calculating the GCD with the following simple formula:

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

Thus, LCM can be calculated using the Euclidean algorithm with the same time complexity:

A possible implementation, that cleverly avoids integer overflows by first dividing a with the GCD, is given here:

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

1.2.6 Practice Problems

- [Codechef - GCD and LCM](#)

1.3 Extended Euclidean Algorithm

While the [Euclidean algorithm](#) calculates only the greatest common divisor (GCD) of two integers a and b , the extended version also finds a way to represent GCD in terms of a and b , i.e. coefficients x and y for which:

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

1.3.1 Algorithm

The changes to the original algorithm are very simple. All we need to do is to figure out how the coefficients x and y change during the transition from (a, b) to $(b \bmod a, a)$.

Let us assume we found the coefficients (x_1, y_1) for $(b \bmod a, a)$:

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

and we want to find the pair (x, y) for (a, b) :

$$a \cdot x + b \cdot y = g$$

We can represent $b \bmod a$ is:

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

Substituting this expression in the coefficient equation of (x_1, y_1) gives:

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

and after rearranging the terms:

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

We found the values of x and y :

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

1.3.2 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;  
}
```

The recursive function above returns the GCD and the values of coefficients to x and y (which are passed by reference to the function).

Base case for the recursion is $a = 0$, when the GCD equals b , so the coefficients x and y are 0 and 1, respectively. In all other cases the above formulas are used to re-calculate the coefficients on each iteration.

This implementation of extended Euclidean algorithm produces correct results for negative integers as well.

1.3.3 Practice Problems

- [10104 - Euclid Problem](#)
- [GYM - \(J\) Once Upon A Time](#)
- [UVA - 12775 - Gift Dilemma](#)

1.4 Linear Diophantine Equation

A Linear Diophantine Equation (in two variables) is an equation of the general form:

$$ax + by = c$$

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

In this article, we consider several classical problems on these equations:

- finding one solution
- finding all solutions
- finding the number of solutions and the solutions themselves in a given interval
- finding a solution with minimum value of $x + y$

1.4.1 The degenerate case

A degenerate case that need to be taken care of is when $a = b = 0$. It is easy to see that we either have no solutions or infinitely many solutions, depending on whether $c = 0$ or not. In the rest of this article, we will ignore this case.

1.4.2 Finding a solution

To find one solution of the Diophantine equation with 2 unknowns, you can use the extended Euclidean algorithm. First, assume that a and b are non-negative. When we apply extended Euclidean algorithm for a and b , we can find their greatest common divisor g and 2 numbers x_g and y_g such that:

$$ax_g + by_g = g$$

If c is divisible by $g = \gcd(a, b)$, then the given Diophantine equation has a solution, otherwise it does not have any solution. The proof is straight-forward: a linear combination of two numbers is divisible by their common divisor.

Now supposed that c is divisible by g , then we have:

$$a \cdot x_g \cdot \frac{c}{g} + b \cdot y_g \cdot \frac{c}{g} = c$$

Therefore one of the solutions of the Diophantine equation is:

$$x_0 = x_g \cdot \frac{c}{g},$$

$$y_0 = y_g \cdot \frac{c}{g}.$$

The above idea still works when a or b or both of them are negative. We only need to change the sign of x_0 and y_0 when necessary.

Finally, we can implement this idea as follows (note that this code does not consider the case $a = b = 0$):

```

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) {
        return false;
    }

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

```

1.4.3 Getting all solutions

From one solution (x_0, y_0) , we can obtain all the solutions of the given equation.

Let $g = \gcd(a, b)$ and let x_0, y_0 be integers which satisfy the following:

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

Now, we should see that adding b/g to x_0 , and, at the same time subtracting a/g from y_0 will not break the equality:

$$a \cdot \left(x_0 + \frac{b}{g}\right) + b \cdot \left(y_0 - \frac{a}{g}\right) = a \cdot x_0 + b \cdot y_0 + a \cdot \frac{b}{g} - b \cdot \frac{a}{g} = c$$

Obviously, this process can be repeated again, so all the numbers of the form:

$$\begin{aligned}
 x &= x_0 + k \cdot \frac{b}{g} \\
 y &= y_0 - k \cdot \frac{a}{g}
 \end{aligned}$$

are solutions of the given Diophantine equation.

Moreover, this is the set of all possible solutions of the given Diophantine equation.

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

From previous section, it should be clear that if we don't impose any restrictions on the solutions, there would be infinite number of them. So in this section, we add some restrictions on the interval of x and y , and we will try to count and enumerate all the solutions.

Let there be two intervals: $[min_x; max_x]$ and $[min_y; max_y]$ and let's say we only want to find the solutions in these two intervals.

Note that if a or b is 0, then the problem only has one solution. We don't consider this case here.

First, we can find a solution which have minimum value of x , such that $x \geq min_x$. To do this, we first find any solution of the Diophantine equation. Then, we shift this solution to get $x \geq min_x$ (using what we know about the set of all solutions in previous section). This can be done in $O(1)$. Denote this minimum value of x by lx_1 .

Similarly, we can find the maximum value of x which satisfy $x \leq max_x$. Denote this maximum value of x by rx_1 .

Similarly, we can find the minimum value of y ($y \geq min_y$) and maximum values of y ($y \leq max_y$). Denote the corresponding values of x by lx_2 and rx_2 .

The final solution is all solutions with x in intersection of $[lx_1, rx_1]$ and $[lx_2, rx_2]$. Let denote this intersection by $[lx, rx]$.

Following is the code implementing this idea:

```
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);

    if (lx > rx) return 0;
    return (rx - lx) / abs(b) + 1;
}

```

When we have lx and rx , it is simple to enumerate through all the solutions. Just need to iterate through $x = lx + k \cdot b$ and find the corresponding y using the equation $ax + by = c$.

1.4.5 Find the solution with minimum value of $x + y$

Here, x and y also need to be given some restriction, otherwise, the answer may become negative infinity.

The idea is similar to previous section: We find any solution of the Diophantine equation, and then shift the solution to satisfy some conditions.

Finally, use the knowledge of the set of all solutions to find the minimum:

$$\begin{aligned}
 x' &= x + k \cdot \frac{b}{g}, \\
 y' &= y - k \cdot \frac{a}{g}.
 \end{aligned}$$

Note that $x + y$ change as follows:

$$x' + y' = x + y + k \cdot \left(\frac{b}{g} - \frac{a}{g} \right) = x + y + k \cdot \frac{b - a}{g}$$

If $a < b$, we need to select smallest possible value of k . If $a > b$, we need to select the largest possible value of k . If $a = b$, all solution will have the same sum $x + y$.

1.4.6 Practice Problems

- [Spoj - Crucial Equation](#)
- [SGU 106](#)

-
- [Codeforces - Ebony and Ivory](#)
 - [Codechef - Get AC in one go](#)
 - [LightOj - Solutions to an equation](#)

1.5 Fibonacci Numbers

The Fibonacci sequence is defined as follows:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

The first elements of the sequence ([OEIS A000045](#)) are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$$

1.5.1 Properties

Fibonacci numbers possess a lot of interesting properties. Here are a few of them:

- Cassini's identity:

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

- The “addition” rule:

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

- Applying the previous identity to the case $k = n$, we get:

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

- From this we can prove by induction that for any positive integer k , F_{nk} is multiple of F_n .
- The inverse is also true: if F_m is multiple of F_n , then m is multiple of n .
- GCD identity:

$$\text{GCD}(F_m, F_n) = F_{\text{GCD}(m, n)}$$

- Fibonacci numbers are the worst possible inputs for Euclidean algorithm (see [Lame's theorem](#) in [Euclidean algorithm](#))

1.5.2 Fibonacci Coding

We can use the sequence to encode positive integers into binary code words. According to Zeckendorf's theorem, any natural number n can be uniquely represented as a sum of Fibonacci numbers:

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

such that $k_1 \geq k_2 + 2$, $k_2 \geq k_3 + 2$, \dots , $k_r \geq 2$ (i.e.: the representation cannot use two consecutive Fibonacci numbers).

It follows that any number can be uniquely encoded in the Fibonacci coding, for example:

1 =	1 =	$F_2 =$	$(11)_F$
2 =	2 =	$F_3 =$	$(011)_F$
6 =	$5 + 1 =$	$F_5 + F_2 =$	$(10011)_F$
8 =	8 =	$F_6 =$	$(000011)_F$
9 =	$8 + 1 =$	$F_6 + F_2 =$	$(100011)_F$
19 =	$13 + 5 + 1 =$	$F_7 + F_5 + F_2 =$	$(1010011)_F$

and there are no consecutive 1-bits, except for the ending 1 bit, which simply indicates the end of the code word.

The encoding of an integer n can be done with a simple greedy algorithm:

1. Iterate through the Fibonacci numbers from the largest to the smallest until you find one less than or equal to n .
2. Suppose this number was F_i . Subtract F_i from n and put a 1 in the $i - 2$ position of the code word (indexing from 0 from the leftmost to the rightmost bit).
3. Repeat until there is no remainder.
4. Add a final 1 to the codeword to indicate its end.

To decode a code word, first remove the final 1. Then, if the i -th bit is set (indexing from 0 from the leftmost to the rightmost bit), sum F_{i+2} to the number.

1.5.3 Formulas for the n -th Fibonacci number

The n -th Fibonacci number can be easily found in $O(n)$ by computing the numbers one by one up to n . However, there are also faster ways, as we will see.

Closed-form expression

There is a formula known as “Binet’s formula”, even though it was already known by 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 easy to prove by induction, but it can be deduced with the help of the concept of generating functions or by solving a functional equation.

You can immediately notice that the second term’s absolute value is always less than 1, and it also decreases very rapidly (exponentially). Hence the value of the first term alone is “almost” F_n . This can be written strictly as:

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

where the square brackets denote rounding to the nearest integer.

As these two formulas would require very high accuracy when working with fractional numbers, they are of little use in practical calculations.

Matrix form

It is easy to prove the following relation:

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

Denoting $P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, we have:

$$\begin{pmatrix} F_n & F_{n+1} \end{pmatrix} = \begin{pmatrix} F_0 & F_1 \end{pmatrix} \cdot P^n$$

Thus, in order to find F_n , we must raise the matrix P to n . This can be done in $O(\log n)$ (see [Binary exponentiation](#)).

Fast Doubling Method

Using above method we can find these equations:

$$\begin{aligned} F_{2k} &= F_k(2F_{k+1} - F_k) \\ F_{2k+1} &= F_{k+1}^2 + F_k^2 \end{aligned}$$

Thus using above two equations Fibonacci numbers can be calculated easily by the following code:

The above code returns F_n and F_{n+1} as a pair.

```
pair<int, int> fib (int n) {
    if (n == 0)
        return {0, 1};

    auto p = fib(n >> 1);
    int c = p.first * (2 * p.second - p.first);
    int d = p.first * p.first + p.second * p.second;
    if (n & 1)
        return {d, c + d};
    else
        return {c, d};
}
```

1.5.4 Periodicity modulo p

Consider the Fibonacci sequence modulo p . We will prove the sequence is periodic and the period begins with $F_1 = 1$ (that is, the pre-period contains only F_0).

Let us prove this by contradiction. Consider the first $p^2 + 1$ pairs of Fibonacci numbers taken modulo p :

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

There can only be p different remainders modulo p , and at most p^2 different remainders, so there are at least two identical pairs among them. Thus the sequence is periodic.

We now choose two pairs of identical remainders with the smallest indices in the sequence. Let the pairs be (F_a, F_{a+1}) and (F_b, F_{b+1}) . We will prove that $a = 1$. If this was false, there would be two previous pairs (F_{a-1}, F_a) and (F_{b-1}, F_b) , which, by the property of Fibonacci numbers, would also be equal. However, this contradicts the fact that we had chosen pairs with the smallest indices, completing our proof.

1.5.5 Practice Problems

- [SPOJ - Euclid Algorithm Revisited](#)
- [SPOJ - Fibonacci Sum](#)
- [HackerRank - Is Fib](#)
- [Project Euler - Even Fibonacci numbers](#)

Chapter 2

Prime numbers

2.1 Sieve of Eratosthenes

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

The algorithm is very simple: at the beginning we write down all numbers between 2 and n . We mark all proper multiples of 2 (since 2 is the smallest prime number) as composite. A proper multiple of a number x , is a number greater than x and divisible by x . Then we find the next number that hasn't been marked as composite, in this case it is 3. Which means 3 is prime, and we mark all proper multiples of 3 as composite. The next unmarked number is 5, which is the next prime number, and we mark all proper multiples of it. And we continue this procedure until we processed all numbers in the row.

In the following image you can see a visualization of the algorithm for computing all prime numbers in the range $[1; 16]$. It can be seen, that quite often we mark numbers as composite multiple times.



Figure 2.1: Sieve of Eratosthenes

The idea behind is this: A number is prime, if none of the smaller prime numbers divides it. Since we iterate over the prime numbers in order, we already marked all numbers, who are divisible by at least one of the prime numbers, as divisible. Hence if we reach a cell and it is not marked, then it isn't divisible by any smaller prime number and therefore has to be prime.

2.1.1 Implementation

```
int n;
vector<char> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i <= n; i++) {
    if (is_prime[i] && (long long)i * i <= n) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

This code first marks all numbers except zero and one as potential prime numbers, then it begins the process of sifting composite numbers. For this it iterates over all numbers from 2 to n . If the current number i is a prime number, it marks all numbers that are multiples of i as composite numbers, starting from i^2 . This is already an optimization over naive way of implementing it, and is allowed as all smaller numbers that are multiples of i necessary also have a prime factor which is less than i , so all of them were already sifted earlier. Since i^2 can easily overflow the type `int`, the additional verification is done using type `long long` before the second nested loop.

Using such implementation the algorithm consumes $O(n)$ of the memory (obviously) and performs $O(n \log \log n)$ (see next section).

2.1.2 Asymptotic analysis

Let's prove that algorithm's running time is $O(n \log \log n)$. The algorithm will perform $\frac{n}{p}$ operations for every prime $p \leq n$ the inner loop. Hence, we need to evaluate the next expression:

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

Let's recall two known facts.

- The number of prime numbers less than or equal to n is approximately $\frac{n}{\ln n}$.
- The k -th prime number approximately equals $k \ln k$ (that follows immediately from the previous fact).

Thus we can write down the sum in the following way:

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

Here we extracted the first prime number 2 from the sum, because $k = 1$ in approximation $k \ln k$ is 0 and causes a division by zero.

Now, let's evaluate this sum using the integral of a same function over k from 2 to $\frac{n}{\ln n}$ (we can make such approximation because, in fact, the sum is related to the integral as its approximation using the rectangle method):

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

The antiderivative for the integrand is $\ln \ln k$. Using a substitution and removing terms of lower order, we'll get the result:

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

Now, returning to the original sum, we'll get its approximate evaluation:

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

You can find a more strict proof (that gives more precise evaluation which is accurate within constant multipliers) in the book authored by Hardy & Wright "An Introduction to the Theory of Numbers" (p. 349).

2.1.3 Different optimizations of the Sieve of Eratosthenes

The biggest weakness of the algorithm is, that it "walks" along the memory multiple times, only manipulating single elements. This is not very cache friendly. And because of that, the constant which is concealed in $O(n \log \log n)$ is comparably big.

Besides, the consumed memory is a bottleneck for big n .

The methods presented below allow us to reduce the quantity of the performed operations, as well as to shorten the consumed memory noticeably.

Sieving till root

Obviously, to find all the prime numbers until n , it will be enough just to perform the sifting only by the prime numbers, which do not exceed the root of n .

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

Such optimization doesn't affect the complexity (indeed, by repeating the proof presented above we'll get the evaluation $n \ln \ln \sqrt{n} + o(n)$, which is asymptotically the same according to the properties of logarithms), though the number of operations will reduce noticeably.

Sieving by the odd numbers only

Since all even numbers (except 2) are composite, we can stop checking even numbers at all. Instead, we need to operate with odd numbers only.

First, it will allow us to half the needed memory. Second, it will reduce the number of operations performing by algorithm approximately in half.

Reducing consumed memory

We should notice that algorithm of Eratosthenes operates with n bits of memory. Hence, we can essentially reduce consumed memory by preserving not n bytes, which are the variables of Boolean type, but n bits, i.e. $\frac{n}{8}$ bytes of memory.

However, such an approach, which is called **bit-level compression**, will complicate the operations with these bits. Read or write operation on any bit will require several arithmetic operations and ultimately slow down the algorithm.

Thus, this approach is only justified, if n is so big that we cannot allocate n bytes of memory anymore. In this case we will trade saving memory (8 times less) with significant slowing down of the algorithm.

After all, it's worth mentioning there exist data structures that automatically do a bit-level compression, such as `vector<bool>` and `bitset<>` in C++.

Block sieving

It follows from the optimization "sieving till root" that there is no need to keep the whole array `is_prime[1...n]` at all time. For performing of sieving it's enough to keep just prime numbers until root of n , i.e. `prime[1... sqrt(n)]`, split the complete range into blocks, and sieve each block separately. In doing so, we never have to keep multiple blocks in memory at the same time, and the CPU handles caching a lot better.

Let s be a constant which determines the size of the block, then we have $\lceil \frac{n}{s} \rceil$ blocks altogether, and the block k ($k = 0 \dots \lceil \frac{n}{s} \rceil$) contains the numbers in a segment $[ks; ks + s - 1]$. We can work on blocks by turns, i.e. for every block k we will go through all the prime numbers (from 1 to \sqrt{n}) and perform sieving using them. It is worth noting, that we have to modify the strategy a little bit when handling the first numbers: first, all the prime numbers from $[1; \sqrt{n}]$ shouldn't remove themselves; and second, the numbers 0 and 1 should be marked as non-prime numbers. While working on the last block it should not be forgotten that the last needed number n is not necessarily located in the end of the block.

Here we have an implementation that counts the number of primes smaller than or equal to n using block sieving.

```
int count_primes(int n) {
    const int S = 10000;
```



```

vector<int> primes;
int nsqrt = sqrt(n);
vector<char> is_prime(nsqrt + 1, true);
for (int i = 2; i <= nsqrt; i++) {
    if (is_prime[i]) {
        primes.push_back(i);
        for (int j = i * i; j <= nsqrt; j += i)
            is_prime[j] = false;
    }
}

int result = 0;
vector<char> block(S);
for (int k = 0; k * S <= n; k++) {
    fill(block.begin(), block.end(), true);
    int start = k * S;
    for (int p : primes) {
        int start_idx = (start + p - 1) / p;
        int j = max(start_idx, p) * p - start;
        for (; j < S; j += p)
            block[j] = false;
    }
    if (k == 0)
        block[0] = block[1] = false;
    for (int i = 0; i < S && start + i <= n; i++) {
        if (block[i])
            result++;
    }
}
return result;
}

```

The running time of block sieving is the same as for regular sieve of Eratosthenes (unless the size of the blocks is very small), but the needed memory will shorten to $O(\sqrt{n} + S)$ and we have better caching results. On the other hand, there will be a division for each pair of a block and prime number from $[1; \sqrt{n}]$, and that will be far worse for smaller block sizes. Hence, it is necessary to keep balance when selecting the constant S . We achieved the best results for block sizes between 10^4 and 10^5 .

2.1.4 Linear time modification

We can modify the algorithm in a such a way, that it only has linear time complexity. This approach is described in the article [Sieve of Eratosthenes Having Linear Time Complexity](#). However, this algorithm also has its own weaknesses.

2.1.5 Practice Problems

- [SPOJ - Printing Some Primes](#)

- [SPOJ - A Conjecture of Paul Erdos](#)
- [SPOJ - Primal Fear](#)
- [SPOJ - Primes Triangle \(I\)](#)
- [Codeforces - Almost Prime](#)
- [Codeforces - Sherlock And His Girlfriend](#)
- [SPOJ - Namit in Trouble](#)
- [SPOJ - Bazinga!](#)
- [Project Euler - Prime pair connection](#)
- [SPOJ - N-Factorful](#)
- [SPOJ - Binary Sequence of Prime Numbers](#)
- [UVA 11353 - A Different Kind of Sorting](#)
- [SPOJ - Printing some primes \(hard\)](#)
- [Codeforces - Nodbach Problem](#)
- [Codeforces - Colliders](#)

2.2 Sieve of Eratosthenes Having Linear Time Complexity

Given a number n , find all prime numbers in a segment $[2; n]$.

The standard way of solving a task is to use [the sieve of Eratosthenes](#). This algorithm is very simple, but it has runtime $O(n \log \log n)$.

Although there are a lot of known algorithms with sublinear runtime (i.e. $o(n)$), the algorithm described below is interesting by its simplicity: it isn't any more complex than the classic sieve of Eratosthenes.

Besides, the algorithm given here calculates **factorizations of all numbers** in the segment $[2; n]$ as a side effect, and that can be helpful in many practical applications.

The weakness of the given algorithm is in using more memory than the classic sieve of Eratosthenes': it requires an array of n numbers, while for the classic sieve of Eratosthenes it is enough to have n bits of memory (which is 32 times less).

Thus, it makes sense to use the described algorithm only until for numbers of order 10^7 and not greater.

The algorithm's authorship appears to belong to Gries & Misra (Gries, Misra, 1978: see references in the end of the article). And, strictly speaking, this algorithm shouldn't be called "sieve of Eratosthenes" since it's too different from the classic one.

2.2.1 Algorithm

Our goal is to calculate **minimum prime factor** $lp[i]$ for every number i in the segment $[2; n]$.

Besides, we need to store the list of all the found prime numbers - let's call it $pr[]$.

We'll initialize the values $lp[i]$ with zeros, which means that we assume all numbers are prime. During the algorithm execution this array will be filled gradually.

Now we'll go through the numbers from 2 to n . We have two cases for the current number i :

- $lp[i] = 0$ - that means that i is prime, i.e. we haven't found any smaller factors for it.
Hence, we assign $lp[i] = i$ and add i to the end of the list $pr[]$.
- $lp[i] \neq 0$ - that means that i is composite, and its minimum prime factor is $lp[i]$.

In both cases we update values of $lp[]$ for the numbers that are divisible by i . However, our goal is to learn to do so as to set a value $lp[]$ at most once for every number. We can do it as follows:

Let's consider numbers $x_j = i \cdot p_j$, where p_j are all prime numbers less than or equal to $lp[i]$ (this is why we need to store the list of all prime numbers).

We'll set a new value $lp[x_j] = p_j$ for all numbers of this form.

The proof of correctness of this algorithm and its runtime can be found after the implementation.

2.2.2 Implementation

```
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];
}
```

We can speed it up a bit by replacing vector pr with a simple array and a counter, and by getting rid of the second multiplication in the nested `for` loop (for that we just need to remember the product in a variable).

2.2.3 Correctness Proof

We need to prove that the algorithm sets all values $lp[]$ correctly, and that every value will be set exactly once. Hence, the algorithm will have linear runtime, since all the remaining actions of the algorithm, obviously, work for $O(n)$.

Notice that every number i has exactly one representation in form:

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

where $lp[i]$ is the minimal prime factor of i , and the number x doesn't have any prime factors less than $lp[i]$, i.e.

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

Now, let's compare this with the actions of our algorithm: in fact, for every x it goes through all prime numbers it could be multiplied by, i.e. all prime numbers up to $lp[x]$ inclusive, in order to get the numbers in the form given above.

Hence, the algorithm will go through every composite number exactly once, setting the correct values $lp[]$ there. Q.E.D.

2.2.4 Runtime and Memory

Although the running time of $O(n)$ is better than $O(n \log \log n)$ of the classic sieve of Eratosthenes, the difference between them is not so big. In practice that means just double difference in speed, and the optimized versions of the sieve run as fast as the algorithm given here.

Considering the memory requirements of this algorithm - an array $lp[]$ of length n , and an array of $pr[]$ of length $\frac{n}{\ln n}$, this algorithm seems to worse than the classic sieve in every way.

However, its redeeming quality is that this algorithm calculates an array $lp[]$, which allows us to find factorization of any number in the segment $[2; n]$ in the time of the size order of this factorization. Moreover, using just one extra array will allow us to avoid divisions when looking for factorization.

Knowing the factorizations of all numbers is very useful for some tasks, and this algorithm is one of the few which allow to find them in linear time.

2.2.5 References

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

Chapter 3

Number-theoretic functions

3.1 Euler's totient function

Euler's totient function, also known as **phi-function** $\phi(n)$, counts the number of integers between 1 and n inclusive, which are coprime to n . Two numbers are coprime if their greatest common divisor equals 1 (1 is considered to be coprime to any number).

Here are values of $\phi(n)$ for the first few positive integers:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$\phi(n)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8	12

3.1.1 Properties

The following properties of Euler totient function are sufficient to calculate it for any number:

- If p is a prime number, then $\gcd(p, q) = 1$ for all $1 \leq q < p$. Therefore we have:

$$\phi(p) = p - 1.$$

- If p is a prime number and $k \geq 1$, then there are exactly p^k/p numbers between 1 and p^k that are divisible by p . Which gives us:

$$\phi(p^k) = p^k - p^{k-1}.$$

- If a and b are relatively prime, then:

$$\phi(ab) = \phi(a) \cdot \phi(b).$$

This relation is not trivial to see. It follows from the [Chinese remainder theorem](#). The Chinese remainder theorem guarantees, that for each $0 \leq x < a$ and each $0 \leq y < b$, there exists a unique $0 \leq z < ab$ with $z \equiv x \pmod{a}$ and $z \equiv y \pmod{b}$. It's not hard to show that z is coprime to ab if and only if x is coprime to a and y is coprime to b . Therefore the amount of integers coprime to ab is equal to product of the amounts of a and b .

- In general, for not coprime a and b , the equation

$$\phi(ab) = \phi(a) \cdot \phi(b) \cdot \frac{d}{\phi(d)}$$

with $d = \gcd(a, b)$ holds.

Thus, using the first three properties, we can compute $\phi(n)$ through the factorization of n (decomposition of n into a product of its prime factors). If $n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$, where p_i are prime factors of n ,

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

3.1.2 Implementation

Here is an implementation using factorization in $O(\sqrt{n})$:

```
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;
}
```

3.1.3 Application in Euler's theorem

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

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

if a and m are relatively prime.

In the particular case when m is prime, Euler's theorem turns into **Fermat's little theorem**:

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

Euler's theorem and Euler's totient function occur quite often in practical applications, for example both are used to compute the [modular multiplicative inverse](#).

As immediate consequence we also get the equivalence:

$$a^n \equiv a^{n \bmod \phi(m)} \pmod{m}$$

This allows computing $x^n \bmod m$ for very big n , especially if n is the result of another computation, as it allows to compute n under a modulo.

3.1.4 Generalization

There is a less known version of the last equivalence, that allows computing $x^n \bmod m$ efficiently for not coprime x and m . For arbitrary x, m and $n \geq \log_2 m$:

$$x^n \equiv x^{\varphi(m) + [n \bmod \varphi(m)]} \pmod{m}$$

Proof:

Let p_1, \dots, p_t be common prime divisors of x and m , and k_i their exponents in m . With those we define $a = p_1^{k_1} \dots p_t^{k_t}$, which makes $\frac{m}{a}$ coprime to x . And let k be the smallest number such that a divides x^k . Assuming $n \geq k$, we can write:

$$\begin{aligned} x^n \bmod m &= \frac{x^k}{a} a x^{n-k} \bmod m \\ &= \frac{x^k}{a} (a x^{n-k} \bmod m) \bmod m \\ &= \frac{x^k}{a} \left(a x^{n-k} \bmod a \frac{m}{a} \right) \bmod m \\ &= \frac{x^k}{a} a \left(x^{n-k} \bmod \frac{m}{a} \right) \bmod m \\ &= x^k \left(x^{n-k} \bmod \frac{m}{a} \right) \bmod m \end{aligned}$$

The equivalence between the third and forth line follows from the fact that $ab \bmod ac = a(b \bmod c)$. Indeed if $b = cd + r$ with $r < c$, then $ab = acd + ar$ with $ar < ac$.

Since x and $\frac{m}{a}$ are coprime, we can apply Euler's theorem and get the efficient (since k is very small; in fact $k \leq \log_2 m$) formula:

$$x^n \bmod m = x^k \left(x^{n-k \bmod \phi(\frac{m}{a})} \bmod \frac{m}{a} \right) \bmod m.$$

This formula is difficult to apply, but we can use it to analyze the behavior of $x^n \bmod m$. We can see that the sequence of powers ($x^1 \bmod m, x^2 \bmod m, x^3 \bmod m, \dots$) enters a cycle of length $\phi(\frac{m}{a})$ after the first k (or less) elements. $\phi(m)$ divides $\phi(\frac{m}{a})$ (because a and $\frac{m}{a}$ are coprime we have $\phi(a) \cdot \phi(\frac{m}{a}) = \phi(m)$), therefore we can also say that the period has length $\phi(m)$. And since $\phi(m) \geq \log_2 m \geq k$, we can conclude the desired, much simpler, formula:

$$x^n \equiv x^{\varphi(m)} x^{(n - \varphi(m)) \bmod \varphi(m)} \bmod m \equiv x^{\varphi(m) + [n \bmod \varphi(m)]} \pmod{m}.$$

3.1.5 Practice Problems

- [SPOJ #4141 “Euler Totient Function”](#) [Difficulty: CakeWalk]
- [UVA #10179 “Irreducible Basic Fractions”](#) [Difficulty: Easy]
- [UVA #10299 “Relatives”](#) [Difficulty: Easy]
- [UVA #11327 “Enumerating Rational Numbers”](#) [Difficulty: Medium]
- [TIMUS #1673 “Admission to Exam”](#) [Difficulty: High]
- [UVA 10990 - Another New Function](#)
- [Codechef - Golu and Sweetness](#)
- [SPOJ - LCM Sum](#)
- [GYM - Simple Calculations \(F\)](#)
- [UVA 13132 - Laser Mirrors](#)
- [SPOJ - GCDEX](#)
- [UVA 12995 - Farey Sequence](#)
- [SPOJ - Totient in Permutation \(easy\)](#)
- [LOJ - Mathematically Hard](#)
- [SPOJ - Totient Extreme](#)
- [SPOJ - Playing with GCD](#)
- [SPOJ - G Force](#)
- [SPOJ - Smallest Inverse Euler Totient Function](#)
- [Codeforces - Power Tower](#)

3.2 Number of divisors / sum of divisors

In this article we discuss how to compute the number of divisors $d(n)$ and the sum of divisors $\sigma(n)$ of a given number n .

3.2.1 Number of divisors

It should be obvious that the prime factorization of a divisor d has to be a subset of the prime factorization of n , e.g. $6 = 2 \cdot 3$ is a divisor of $60 = 2^2 \cdot 3 \cdot 5$. So we only need to find all different subsets of the prime factorization of n .

Usually the number of subsets is 2^x for a set with x elements. However this is no longer true, if there are repeated elements in the set. In our case some prime factors may appear multiple times in the prime factorization of n .

If a prime factor p appears e times in the prime factorization of n , then we can use the factor p up to e times in the subset. Which means we have $e + 1$ choices.

Therefore if the prime factorization of n is $p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$, where p_i are distinct prime numbers, then the number of divisors is:

$$d(n) = (e_1 + 1) \cdot (e_2 + 1) \cdots (e_k + 1)$$

A way of thinking about it is the following:

- If there is only one distinct prime divisor $n = p_1^{e_1}$, then there are obviously $e_1 + 1$ divisors $(1, p_1, p_1^2, \dots, p_1^{e_1})$.
- If there are two distinct prime divisors $n = p_1^{e_1} \cdot p_2^{e_2}$, then you can arrange all divisors in form of a tabular.

	1	p_2	p_2^2	\dots	$p_2^{e_2}$
1	1	p_2	p_2^2	\dots	$p_2^{e_2}$
p_1	p_1	$p_1 \cdot p_2$	$p_1 \cdot p_2^2$	\dots	$p_1 \cdot p_2^{e_2}$
p_1^2	p_1^2	$p_1^2 \cdot p_2$	$p_1^2 \cdot p_2^2$	\dots	$p_1^2 \cdot p_2^{e_2}$
\vdots	\vdots	\vdots	\vdots	\ddots	\vdots
$p_1^{e_1}$	$p_1^{e_1}$	$p_1^{e_1} \cdot p_2$	$p_1^{e_1} \cdot p_2^2$	\dots	$p_1^{e_1} \cdot p_2^{e_2}$

So the number of divisors is trivially $(e_1 + 1) \cdot (e_2 + 1)$.

- A similar argument can be made if there are more then two distinct prime factors.

3.2.2 Sum of divisors

We can use the same argument of the previous section.

- If there is only one distinct prime divisor $n = p_1^{e_1}$, then the sum is:

$$1 + p_1 + p_1^2 + \dots + p_1^{e_1} = \frac{p_1^{e_1+1} - 1}{p_1 - 1}$$

- If there are two distinct prime divisors $n = p_1^{e_1} \cdot p_2^{e_2}$, then we can make the same table as before. The only difference is that now we now want to compute the sum instead of counting the elements. It is easy to see, that the sum of each combination can be expressed as:

$$\begin{aligned} & \left(1 + p_1 + p_1^2 + \cdots + p_1^{e_1}\right) \cdot \left(1 + p_2 + p_2^2 + \cdots + p_2^{e_2}\right) \\ &= \frac{p_1^{e_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{e_2+1} - 1}{p_2 - 1} \end{aligned}$$

- In general, for $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$ we receive the formula:

$$\sigma(n) = \frac{p_1^{e_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{e_2+1} - 1}{p_2 - 1} \cdots \frac{p_k^{e_k+1} - 1}{p_k - 1}$$

3.2.3 Multiplicative functions

A multiplicative functions is a function $f(x)$ which satisfies

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

if a and b are coprime.

Both $d(n)$ and $\sigma(n)$ are multiplicative functions.

Multiplicative functions have a huge variety of interesting properties, which can be very useful in number theory problems. For instance the Dirichlet convolution of two multiplicative functions is also multiplicative.

3.2.4 Practice Problems

- [SPOJ - COMDIV](#)
- [SPOJ - DIVSUM](#)
- [SPOJ - DIVSUM2](#)

Chapter 4

Modular arithmetic

4.1 Modular Multiplicative Inverse

4.1.1 Definition

A [modular multiplicative inverse](#) of an integer a is an integer x such that $a \cdot x$ is congruent to 1 modular some modulus m . To write it in a formal way: we want to find an integer x so that

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

We will also denote x simply with a^{-1} .

We should note that the modular inverse does not always exist. For example, let $m = 4$, $a = 2$. By checking all possible values modulo m it should become clear that we cannot find a^{-1} satisfying the above equation. It can be proven that the modular inverse exists if and only if a and m are relatively prime (i.e. $\gcd(a, m) = 1$).

In this article, we present two methods for finding the modular inverse in case it exists, and one method for finding the modular inverse for all numbers in linear time.

4.1.2 Finding the Modular Inverse using Extended Euclidean algorithm

Consider the following equation (with unknown x and y):

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

This is a [Linear Diophantine equation in two variables](#). As shown in the linked article, when $\gcd(a, m) = 1$, the equation has a solution which can be found using the [extended Euclidean algorithm](#). Note that $\gcd(a, m) = 1$ is also the condition for the modular inverse to exist.

Now, if we take modulo m of both sides, we can get rid of $m \cdot y$, and the equation becomes:

$$a \cdot x \equiv 1 \pmod{m}$$

Thus, the modular inverse of a is x .

The implementation is as follows:

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

Notice that we way we modify x . The resulting x from the extended Euclidean algorithm may be negative, so $x \% m$ might also be negative, and we first have to add m to make it positive.

4.1.3 Finding the Modular Inverse using Binary Exponentiation

Another method for finding modular inverse is to use Euler's theorem, which states that the following congruence is true if a and m are relatively prime:

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

ϕ is [Euler's Totient function](#). Again, note that a and m being relative prime was also the condition for the modular inverse to exist.

If m is a prime number, this simplifies to [Fermat's little theorem](#):

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

Multiply both sides of the above equations by a^{-1} , and we get:

- For an arbitrary (but coprime) modulus m : $a^{\phi(m)-1} \equiv a^{-1} \pmod{m}$
- For a prime modulus m : $a^{m-2} \equiv a^{-1} \pmod{m}$

From these results, we can easily find the modular inverse using the [binary exponentiation algorithm](#), which works in $O(\log m)$ time.

Even though this method is easier to understand than the method described in previous paragraph, in the case when m is not a prime number, we need to calculate Euler phi function, which involves factorization of m , which might be very hard. If the prime factorization of m is known, then the complexity of this method is $O(\log m)$.

4.1.4 Finding the modular inverse for every number modulo m

The problem is the following: we want to compute the modular inverse for every number in the range $[1, m - 1]$.

Applying the algorithms described in the previous sections, we can obtain a solution with complexity $O(m \log m)$.

Here we present a better algorithm with complexity $O(m)$. However for this specific algorithm we require that the modulus m is prime.

We denote by $\text{inv}[i]$ the modular inverse of i . Then for $i > 1$ the following equation is valid:

$$\text{inv}[i] = - \left\lfloor \frac{m}{i} \right\rfloor \cdot \text{inv}[m \bmod i] \bmod m$$

Thus the implementation is very simple:

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

Proof

We have:

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

Taking both sides modulo m yields:

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

Multiply both sides by $i^{-1} \cdot (m \bmod i)^{-1}$ yields

$$(m \bmod i) \cdot i^{-1} \cdot (m \bmod i)^{-1} \equiv - \left\lfloor \frac{m}{i} \right\rfloor \cdot i \cdot i^{-1} \cdot (m \bmod i)^{-1} \pmod m,$$

which simplifies to:

$$i^{-1} \equiv - \left\lfloor \frac{m}{i} \right\rfloor \cdot (m \bmod i)^{-1} \pmod m,$$

4.1.5 Practice Problems

- [UVa 11904 - One Unit Machine](#)
- [Hackerrank - Longest Increasing Subsequence Arrays](#)
- [Codeforces 300C - Beautiful Numbers](#)
- [Codeforces 622F - The Sum of the k-th Powers](#)
- [Codeforces 717A - Festival Organization](#)
- [Codeforces 896D - Nephren Runs a Cinema](#)

4.2 Linear Congruence Equation

This equation is of the form:

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

where a , b and n are given integers and x is an unknown integer.

It is required to find the value x from the interval $[0, n - 1]$ (clearly, on the entire number line there can be infinitely many solutions that will differ from each other in $n \cdot k$, where k is any integer). If the solution is not unique, then we will consider how to get all the solutions.

4.2.1 Solution by finding the inverse element

Let us first consider a simpler case where a and n are **coprime** ($\gcd(a, n) = 1$). Then one can find the **inverse** of a , and multiplying both sides of the equation with the inverse, and we can get a **unique** solution.

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

Now consider the case where a and n are **not coprime** ($\gcd(a, n) \neq 1$). Then the solution will not always exist (for example $2 \cdot x = 1 \pmod{4}$ has no solution).

Let $g = \gcd(a, n)$, i.e. the **greatest common divisor** of a and n (which in this case is greater than one).

Then, if b is not divisible by g , there is no solution. In fact, for any x the left side of the equation $a \cdot x \pmod{n}$, is always divisible by g , while the right-hand side is not divisible by it, hence it follows that there are no solutions.

If g divides b , then by dividing both sides of the equation by g (i.e. dividing a , b and n by g), we receive a new equation:

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

in which a' and n' are already relatively prime, and we have already learned how to handle such an equation. We get x' as solution for x .

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

$$x_i = (x' + i \cdot n') \pmod{n} \quad \text{for } i = 0 \dots g - 1$$

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

4.2.2 Solution with the Advanced Euclidean Algorithm

We can rewrite the linear congruence to the following Diophantine equation:

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

where x and k are unknown integers.

The method of solving this equation is described in the corresponding article [Linear Diophantine equations](#) and it consists of applying the [Extended Euclidean Algorithm](#).

It also describes the method of obtaining all solutions of this equation from one found solution, and incidentally this method, when carefully considered, is absolutely equivalent to the method described in the previous section.

4.3 Chinese Remainder Theorem

The Chinese Remainder Theorem (which will be referred to as CRT in the rest of this article) was discovered by Chinese mathematician Sun Zi.

4.3.1 Formulation

Let $p = p_1 p_2 \cdots p_k$, where p_i are pairwise relatively prime. In addition to p_i , we are also given a set of congruence equations

$$\begin{aligned} a &\equiv a_1 \pmod{p_1} \\ a &\equiv a_2 \pmod{p_2} \\ &\dots \\ a &\equiv a_k \pmod{p_k} \end{aligned}$$

where a_i are some given constants. The original form of CRT then states that the given set of congruence equations always has *one and exactly one* solution modulo p .

Corollary

A consequence of the CRT is that the equation

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

is equivalent to the system of equations

$$\begin{aligned} x &\equiv a \pmod{p_1} \\ &\dots \\ x &\equiv a \pmod{p_k} \end{aligned}$$

(As above, assume that $p = p_1 p_2 \cdots p_k$ and p_i are pairwise relatively prime).

4.3.2 Garner's Algorithm

Another consequence of the CRT is that we can represent big numbers using an array of small integers. For example, let p be the product of the first 1000 primes. From calculations we can see that p has around 3000 digits.

Any number a less than p can be represented as an array a_1, \dots, a_k , where $a_i \equiv a \pmod{p_i}$. But to do this we obviously need to know how to get back the number a from its representation. In this section, we discuss Garner's Algorithm, which can be used for this purpose. We seek a representation on the form

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

which is called the mixed radix representation of a . Garner's algorithm computes the coefficients x_1, \dots, x_k .

Let r_{ij} denote the inverse of p_i modulo p_j

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

which can be found using the algorithm described in [Modular Inverse](#). Substituting a from the mixed radix representation into the first congruence equation we obtain

$$a_1 \equiv x_1 \pmod{p_1}.$$

Substituting into the second equation yields

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

which can be rewritten by subtracting x_1 and dividing by p_1 to get

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

Similarly we get that

$$x_3 \equiv ((a_3 - x_1) r_{13} - x_2) r_{23} \pmod{p_3}.$$

Now, we can clearly see an emerging pattern, which can be expressed by the following 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 how to calculate coefficients x_i in $O(k^2)$ time. The number a can now be calculated using the previously mentioned formula

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

It is worth noting that in practice, we almost always need to compute the answer using Big Integers, but the coefficients x_i can usually be calculated using built-in types, and therefore Garner's algorithm is very efficient.

4.3.3 Implementation of Garner's Algorithm

It is convenient to implement this algorithm using Java, because it has built-in support for large numbers through the `BigInteger` class.

Here we show an implementation that can store big numbers in the form of a set of congruence equations. It supports addition, subtraction and multiplication. And with Garner's algorithm we can convert the set of equations into the unique integer. In this code, we take 100 prime numbers greater than 10^9 , which allows numbers as large as 10^{900} .

```

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

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

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

class Number {
    int a[] = new int[SZ];

    public Number() {
    }

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

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

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

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

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

```

```

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

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

    return result;
}
}

```

Note on negative numbers

- Let p be the product of all primes.
- Modular scheme itself does not allow representing negative numbers. However, it can be seen that if we know that the absolute values our numbers are smaller than $p/2$, then we know that it must be negative when the resulting number is greater than $p/2$. The flag `can_be_negative` in this code allows converting it to negative in this case.

4.3.4 Practice Problems:

- [Hackerrank - Number of sequences](#)
- [Codeforces - Remainders Game](#)

4.4 Factorial modulo p in $O(p \log n)$

In some cases it is necessary to consider complex formulas modulo p , containing factorials in both numerator and denominator. We consider the case when p is relatively small. This problem makes sense only when factorials are included in both numerator and denominator of fractions. Otherwise $p!$ and subsequent terms will reduce to zero, but in fractions all multipliers containing p can be reduced, and the resulting expression will be non-zero modulo p .

Thus, formally the task is: You want to calculate $n! \bmod p$, without taking all the multiple factors of p into account that appear in the factorial. Imaging you write down the prime factorization of $n!$, remove all factors p , and compute the product modulo p . We will denote this modified factorial with $n!_{\%p}$.

Learning how to effectively calculate this modified factorial allows us to quickly calculate the value of the various combinatorial formulae (for example, [Binomial coefficients](#)).

4.4.1 Algorithm

Let's write this modified factorial explicitly.

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

It can be clearly seen that factorial is divided into several blocks of same length expect for the last one.

$$\begin{aligned}
 n!_{\%p} &= \underbrace{1 \cdot 2 \cdot 3 \cdot \dots \cdot (p-2) \cdot (p-1) \cdot 1}_{1\text{st}} \cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \dots \cdot (p-2) \cdot (p-1) \cdot 2 \cdot \dots}_{2\text{nd}} \\
 &\quad \cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \dots \cdot (p-2) \cdot (p-1) \cdot 1 \cdot \dots}_{p\text{th}} \cdot \underbrace{1 \cdot 2 \cdot \dots \cdot (n \bmod p)}_{\text{tail}} \pmod{p}.
 \end{aligned}$$

The general part of the blocks it is easy to count — it's just $(p-1)! \bmod p$ that you can calculate programmatically or via Wilson theorem, according to which $(p-1)! \bmod p = p-1$. To multiply these common parts of all blocks, we can raise the value to the higher power modulo p , which can be done in $O(\log n)$ operations using [Binary Exponentiation](#); however, you may notice that the result will always be either 1 or $p-1$, depending on the parity of the index. The value of the last partial block can be calculated separately in $O(p)$. Leaving only the last elements of the blocks, we can examine that:

$$n! \% p = \underbrace{\dots \cdot 1}_{\dots} \cdot \underbrace{\dots \cdot 2}_{\dots} \cdot \dots \cdot \underbrace{\dots \cdot (p-1)}_{\dots} \cdot \underbrace{\dots \cdot 1}_{\dots} \cdot \underbrace{\dots \cdot 1}_{\dots} \cdot \underbrace{\dots \cdot 2}_{\dots} \dots$$

And again, by removing the blocks that we already computed, we receive a “modified” factorial but with smaller dimension ($\lfloor n/p \rfloor$ blocks remain). Thus, in the calculation of “modified” the factorial $n \% p$ we did $O(p)$ operations and are left with the calculation of $(n/p)! \% p$. Revealing this recursive dependence, we obtain that the recursion depth is $O(\log_p n)$, the total asymptotic behavior of the algorithm is thus $O(p \log_p n)$.

4.4.2 Implementation

We don’t need recursion because this is a case of tail recursion and thus can be easily implemented using iteration.

```
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 in $O(p \log_p n)$.

4.5 Discrete Root

The problem of finding discrete root is defined as follows. Given a prime n and two integers a and k , find all x for which:

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

4.5.1 The algorithm

We will solve this problem by reducing it to the [discrete logarithm problem](#).

Let's apply the concept of a [primitive root](#) modulo n . Let g be a primitive root modulo n . Note that since n is prime, it must exist, and it can be found in $O(\text{Ans} \cdot \log \phi(n) \cdot \log n) = O(\text{Ans} \cdot \log^2 n)$ plus time of factoring $\phi(n)$.

We can easily discard the case where $a = 0$. In this case, obviously there is only one answer: $x = 0$.

Since we know that n is a prime, any number between 1 and $n - 1$ can be represented as a power of the primitive root, and we can represent the discrete root problem as follows:

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

where

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

This, in turn, can be rewritten as

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

Now we have one unknown y , which is a discrete logarithm problem. The solution can be found using Shanks' baby-step-giant-step algorithm in $O(\sqrt{n} \log n)$ (or we can verify that there are no solutions).

Having found one solution y_0 , one of solutions of discrete root problem will be $x_0 = g^{y_0} \pmod{n}$.

4.5.2 Finding all solutions from one known solution

To solve the given problem in full, we need to find all solutions knowing one of them: $x_0 = g^{y_0} \pmod{n}$.

Let's recall the fact that a primitive root always has order of $\phi(n)$, i.e. the smallest power of g which gives 1 is $\phi(n)$. Therefore, if we add the term $\phi(n)$ to the exponential, we still get the same value:

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

Hence, all the solutions are of the form:

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

where l is chosen such that the fraction must be an integer. For this to be true, the numerator has to be divisible by the least common multiple of $\phi(n)$ and k . Remember that least common multiple of two numbers $\text{lcm}(a, b) = \frac{a \cdot b}{\text{gcd}(a, b)}$; we'll get

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

This is the final formula for all solutions of discrete root problem.

4.5.3 Implementation

Here is a full implementation, including routines for finding the primitive root, discrete log and finding and printing all solutions.

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

int powmod (int a, int b, int p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = int (res * 1ll * a % p), --b;
        else
            a = int (a * 1ll * 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]);
}

```

4.6 Primitive Root

4.6.1 Definition

In modular arithmetic, a number g is called a **primitive root modulo n** if every number coprime to n is congruent to a power of g modulo n . Mathematically, g is a **primitive root modulo n** if and only if for any integer a such that $\gcd(a, n) = 1$, there exists an integer k such that:

$$g^k \equiv a \pmod{n}.$$

k is then called the **index** or **discrete logarithm** of a to the base g modulo n . g is also called the **generator** of the multiplicative group of integers modulo n .

In particular, for the case where n is a prime, the powers of primitive root runs through all numbers from 1 to $n - 1$.

4.6.2 Existence

Primitive root modulo n exists if and only if:

- n is 1, 2, 4, or
- n is power of an odd prime number ($n = p^k$), or
- n is twice power of an odd prime number ($n = 2.p^k$).

This theorem was proved by Gauss in 1801.

4.6.3 Relation with the Euler function

Let g be a primitive root modulo n . Then we can show that the smallest number k for which $g^k \equiv 1 \pmod{n}$ is equal $\phi(n)$. Moreover, the reverse is also true, and this fact will be used in this article to find a primitive root.

Furthermore, the number of primitive roots modulo n , if there are any, is equal to $\phi(\phi(n))$.

4.6.4 Algorithm for finding a primitive root

A naive algorithm is to consider all numbers in range $[1, n - 1]$. And then check if each one is a primitive root, by calculating all its power to see if they are all different. This algorithm has complexity $O(g.n)$, which would be too slow. In this section, we propose a faster algorithm using several well-known theorems.

From previous section, we know that if the smallest number k for which $g^k \equiv 1 \pmod{n}$ is $\phi(n)$, then g is a primitive root. Since for any number a relative prime to n , we know from Euler's theorem that $a^{\phi(n)} \equiv 1 \pmod{n}$, then to check if g is primitive root, it is enough to check that for all d less than $\phi(n)$, $g^d \not\equiv 1 \pmod{n}$. However, this algorithm is still too slow.

From Lagrange's theorem, we know that the index of any number modulo n must be a divisor of $\phi(n)$. Thus, it is sufficient to verify for all proper divisor $d \mid \phi(n)$ that $g^d \not\equiv 1 \pmod{n}$. This is already a much faster algorithm, but we can still do better.

Factorize $\phi(n) = p_1^{a_1} \dots p_s^{a_s}$. We prove that in the previous algorithm, it is sufficient to consider only the values of d which has the form $\frac{\phi(n)}{p_j}$. Indeed, let d be any proper divisor of $\phi(n)$. Then, obviously, there exists such j that $d \mid \frac{\phi(n)}{p_j}$, i.e. $d.k = \frac{\phi(n)}{p_j}$. However, if $g^d \equiv 1 \pmod{n}$, we would get:

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

i.e. among the numbers of the form $\frac{\phi(n)}{p_i}$, there would be at least one such that the conditions are not met.

Now we have a complete algorithm for finding the primitive root:

- First, find $\phi(n)$ and factorize it.
- Then iterate through all numbers $g = 1 \dots n$, and for each number, to check if it is primitive root, we do the following:
 - Calculate all $g^{\frac{\phi(n)}{p_i}} \pmod{n}$.
 - If all the calculated values are different from 1, then g is a primitive root.

Running time of this algorithm is $O(\text{Ans} \cdot \log \phi(n) \cdot \log n)$ (assume that $\phi(n)$ has $\log \phi(n)$ divisors).

Shoup (1990, 1992) proved, assuming the [generalized Riemann hypothesis](#), that g is $O(\log^6 p)$.

4.6.5 Implementation

The following code assumes that the modulo p is a prime number. To make it works for any value of p , we must add calculation of $\phi(p)$.

```
int powmod (int a, int b, int p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = int (res * 1ll * a % p), --b;
        else
            a = int (a * 1ll * 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;
}
```

4.7 Discrete Logarithm

The discrete logarithm is an integer x solving the equation

$$a^x \equiv b \pmod{m}$$

where a and m are relatively prime. (Note: if they are not relatively prime, then the algorithm described below is incorrect, though it can be modified so that it can work).

In this article, we describe the **Baby step - giant step** algorithm, proposed by Shanks in 1971, which has complexity $O(\sqrt{m} \log m)$. This algorithm is also known as **meet-in-the-middle**, because of it uses technique separation of tasks in half.

4.7.1 Algorithm

Consider the equation:

$$a^x \equiv b \pmod{m}$$

where a and m are relatively prime.

Let $x = np - q$, where n is some pre-selected constant (we will describe how to select n later). p is known as **giant step**, since increasing it by one increases x by n . Similarly, q is known as **baby step**.

Obviously, any value of x in the interval $[0; m)$ can be represented in this form, where $p \in [1; \lceil \frac{m}{n} \rceil]$ and $q \in [0; n]$.

Then, the equation becomes:

$$a^{np-q} \equiv b \pmod{m}.$$

Using the fact that a and m are relatively prime, we obtain:

$$a^{np} \equiv ba^q \pmod{m}$$

This new equation can be rewritten in a simplified form:

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

This problem can be solved using the method meet-in-the-middle as follows:

- We calculate f_1 for all possible values of p . Sort these values.
- For each value of q , calculate f_2 , and look for the corresponding value of p using the sorted array of f_1 using binary search.

4.7.2 Complexity

For each value of p , we can calculate $f_1(p)$ in $O(\log m)$ using [binary exponentiation algorithm](#). Similar for $f_2(q)$.

In the first step of the algorithm, we need to calculate f_1 for every possible values of p , and then sort them. Thus, this step has complexity:

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

In the second step of the algorithm, we need to calculate $f_2(q)$ for each possible value of q , and then do a binary search on the array of values of f_1 , thus this step has complexity:

$$O(n(\log m + \log \frac{m}{n})) = O(n \log m).$$

Now, when we add these two complexity, we would get $\log m$ multiplied by n and m/n , which has minimum value when $n = \sqrt{m}$, which means, to achieve optimal performance, n should be chosen such that:

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

Then, the complexity of the algorithm becomes:

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

4.7.3 Implementation

The simplest implementation

In the following code, function `powmod` performs binary exponential $a^b \pmod{m}$, and function `solve` produces a proper solution to the problem. It will return `-1` if there is no solution, and returns one possible solution in case a solution exists.

```
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;
}
```

In this code, we used `map` from C++ STL to store the values of $f_1(i)$. Internally, `map` uses red-black-tree to store values. This code is a little bit slower than if we use array and binary search for f_1 , but is much easier to write.

Another thing to note is that, if there are multiple values of p that have the same value of f_1 , we only store one such value. This works in this case because we only want to return one possible solution. If we need to return all possible solutions,

we need to change `map<int,int>` to, say, `map<int, vector<int> >`. And we also need to change the second step accordingly.

4.7.4 Improved implementation

A possible improvement is to get rid of binary exponentiation in the second phase of the algorithm. This can be done by keeping a variable that multiplies by a each time we increase q . With this change, the complexity of the algorithm is still the same, but now the log part is only for *map*. Instead of *map*, we can also use hash table (`unordered_map` in GNU C++) which has complexity $O(1)$ for inserting and searching. And when the value of m is small enough, we can also get rid of *map*, and use a regular array to store and lookup values of f_1 .

```
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;
}
```

4.7.5 Practice Problems

- [Spoj - Power Modulo Inverted](#)
- [Topcoder - SplittingFoxes3](#)

Chapter 5

Number systems

5.1 Balanced Ternary

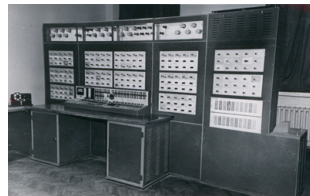


Figure 5.1: “Setun computer using Balanced Ternary system”

This is a non-standard but still positional **numeral system**. It's feature is that digits can have one of values -1 , 0 and 1 . Nevertheless, its base is still 3 (because there are three possible values). Since it is not convenient to write -1 as a digit, we'll use letter Z further for this purpose. If you think it is quite a strange system - look at the picture - here is one of the computers utilizing it.

So here are few first numbers written in balanced ternary:

0	0
1	1
2	1Z
3	10
4	11
5	1ZZ
6	1Z0
7	1Z1
8	10Z
9	100

This system allows you to write negative values without leading minus sign: you can simply invert digits in any positive number.

-1	Z
-2	Z1
-3	Z0
-4	ZZ
-5	Z11

Note that a negative number starts with Z and positive with 1.

5.1.1 Conversion algorithm

It is easy to represent a given number in **balanced ternary** via temporary representing it in normal ternary number system. When value is in standard ternary, its digits are either 0 or 1 or 2. Iterating from the lowest digit we can safely skip any 0s and 1s, however 2 should be turned into Z with adding 1 to the next digit. Digits 3 should be turned into 0 on the same terms - such digits are not present in the number initially but they can be encountered after increasing some 2s.

Example 1: Let us convert 64 to balanced ternary. At first we use normal ternary to rewrite the number:

$$64_{10} = 02101_3$$

Let us process it from the least significant (rightmost) digit:

- 1,0 and 1 are skipped as it is.(Because 0 and 1 are allowed in balanced ternary)
- 2 is turned into Z increasing the digit to its left, so we get 1Z101.

The final result is 1Z101.

Let us convert it back to the decimal system by adding the weighted positional values:

$$1Z101 = 81 \cdot 1 + 27 \cdot (-1) + 9 \cdot 1 + 3 \cdot 0 + 1 \cdot 1 = 64_{10}$$

Example 2: Let us convert 237 to balanced ternary. At first we use normal ternary to rewrite the number:

$$237_{10} = 22210_3$$

Let us process it from the least significant (rightmost) digit:

- 0 and 1 are skipped as it is.(Because 0 and 1 are allowed in balanced ternary)
- 2 is turned into Z increasing the digit to its left, so we get 23Z10.
- 3 is turned into 0 increasing the digit to its left, so we get 30Z10.
- 3 is turned into 0 increasing the digit to its left(which is by default 0), and so we get 100Z10.

The final result is 100Z10.

Let us convert it back to the decimal system by adding the weighted positional values:

$$100Z10 = 243 \cdot 1 + 81 \cdot 0 + 27 \cdot 0 + 9 \cdot (-1) + 3 \cdot 1 + 1 \cdot 0 = 237_{10}$$

5.1.2 Practice Problems

- [Topcoder SRM 604, Div1-250](#)

5.2 Gray code

Gray code is a binary numeral system where two successive values differ in only one bit.

For example, the sequence of Gray codes for 3-bit numbers is: 000, 001, 011, 010, 110, 111, 101, 100, so $G(4) = 6$.

This code was invented by Frank Gray in 1953.

5.2.1 Finding Gray code

Let's look at the bits of number n and the bits of number $G(n)$. Notice that i -th bit of $G(n)$ equals 1 only when i -th bit of n equals 1 and $i + 1$ -th bit equals 0 or the other way around (i -th bit equals 0 and $i + 1$ -th bit equals 1). Thus, $G(n) = n \oplus (n \gg 1)$:

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

5.2.2 Finding inverse Gray code

Given Gray code g , restore the original number n .

We will move from the most significant bits to the least significant ones (the least significant bit has index 1 and the most significant bit has index k). The relation between the bits n_i of number n and the bits g_i of number g :

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

The easiest way to write it in code is:

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

5.2.3 Practical applications

Gray codes have some useful applications, sometimes quite unexpected:

- Gray code of n bits forms a Hamiltonian cycle on a hypercube, where each bit corresponds to one dimension.

- Gray codes are used to minimize the errors in digital-to-analog signals conversion (for example, in sensors).
- Gray code can be used to solve the Towers of Hanoi problem. Let n denote number of disks. Start with Gray code of length n which consists of all zeroes ($G(0)$) and move between consecutive Gray codes (from $G(i)$ to $G(i + 1)$). Let i -th bit of current Gray code represent n -th disk (the least significant bit corresponds to the smallest disk and the most significant bit to the biggest disk). Since exactly one bit changes on each step, we can treat changing i -th bit as moving i -th disk. Notice that there is exactly one move option for each disk (except the smallest one) on each step (except start and finish positions). There are always two move options for the smallest disk but there is a strategy which will always lead to answer: if n is odd then sequence of the smallest disk moves looks like $f \rightarrow t \rightarrow r \rightarrow f \rightarrow t \rightarrow r \rightarrow \dots$ where f is the initial rod, t is the terminal rod and r is the remaining rod), and if n is even: $f \rightarrow r \rightarrow t \rightarrow f \rightarrow r \rightarrow t \rightarrow \dots$
- Gray codes are also used in genetic algorithms theory.

5.2.4 Practice Problems

- SGU #249 “Matrix” [Difficulty: medium]

Chapter 6

Miscellaneous

6.1 Submask Enumeration

6.1.1 Enumerating all submasks of a given mask

Given a bitmask m , you want to efficiently iterate through all its submasks, that is, masks s in which only bits that were included in mask m are set.

Consider the implementation of this algorithm, based on tricks with bit operations:

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

or, using a more compact for statement:

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

In both variants of the code, the submask equal to zero will not be processed. We can either process it outside the loop, or use a less elegant design, for example:

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

Let us examine why the above code visits all submasks of m , without repetition, and in descending order.

Suppose we have a current bitmask s , and we want to move on to the next bitmask. By subtracting from the mask s one unit, we will remove the rightmost set bit and all bits to the right of it will become 1. Then we remove all the “extra” one bits that are not included in the mask m and therefore can’t be a part of a submask. We do this removal by using the bitwise operation $(s-1) \& m$. As a

result, we “cut” mask $s-1$ to determine the highest value that it can take, that is, the next submask after s in descending order.

Thus, this algorithm generates all submasks of this mask in descending order, performing only two operations per iteration.

A special case is when $s = 0$. After executing $s - 1$ we get a mask where all bits are set (bit representation of -1), and after $(s-1) \& m$ we will have that s will be equal to m . Therefore, with the mask $s = 0$ be careful — if the loop does not end at zero, the algorithm may enter an infinite loop.

6.1.2 Iterating through all masks with their submasks. Complexity $O(3^n)$

In many problems, especially those that use bitmask dynamic programming, you want to iterate through all bitmasks and for each mask, iterate through all of its submasks:

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

Let’s prove that the inner loop will execute a total of $O(3^n)$ iterations.

First proof: Consider the i -th bit. There are exactly three options for it: it is not included in the mask m (and therefore not included in submask s); it is included in m , but not included in s , or it’s included in both m and s . As there are a total of n bits, there will be 3^n different combinations.

Second proof: Note that if mask m has k enabled bits, then it will have 2^k submasks. As we have a total of $\binom{n}{k}$ masks with k enabled bits (see “binomial coefficients”), then the total number of combinations for all masks will be:

$$\sum_{k=0}^n \binom{n}{k} \cdot 2^k$$

To calculate this number, note that the sum above is equal to the expansion of $(1 + 2)^n$ using the binomial theorem. Therefore, we have 3^n combinations, as we wanted to prove.

6.1.3 Practice Problems

- [Codeforces - Nuclear Fusion](#)
- [Codeforces - Sandy and Nuts](#)
- [Uva 1439 - Exclusive Access 2](#)
- [UVa 11825 - Hackers’ Crackdown](#)

6.2 Arbitrary-Precision Arithmetic

Arbitrary-Precision arithmetic, also known as “bignum” or simply “long arithmetic” is a set of data structures and algorithms which allows to process much greater numbers than can be fit in standard data types. Here are several types of arbitrary-precision arithmetic.

6.2.1 Classical Integer Long Arithmetic

The main idea is that the number is stored as an array of its “digits” in some base. Several most frequently used bases are decimal, powers of decimal (10^4 or 10^9) and binary.

Operations on numbers in this form are performed using “school” algorithms of column addition, subtraction, multiplication and division. It’s also possible to use fast multiplication algorithms: fast Fourier transform and Karatsuba algorithm.

Here we describe long arithmetic for only non-negative integers. To extend the algorithms to handle negative integers one has to introduce and maintain additional “negative number” flag or use two’s complement integer representation.

Data Structure

We’ll store numbers as a `vector<int>`, in which each element is a single “digit” of the number.

```
typedef vector<int> lnum;
```

To improve performance we’ll use 10^9 as the base, so that each “digit” of the long number contains 9 decimal digits at once.

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

Digits will be stored in order from least to most significant. All operations will be implemented so that after each of them the result doesn’t have any leading zeros, as long as operands didn’t have any leading zeros either. All operations which might result in a number with leading zeros should be followed by code which removes them. Note that in this representation there are two valid notations for number zero: and empty vector, and a vector with a single zero digit.

Output

Printing the long integer is the easiest operation. First we print the last element of the vector (or 0 if the vector is empty), followed by the rest of the elements padded with leading zeros if necessary so that they are exactly 9 digits long.

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

Note that we cast `a.size()` to integer to avoid unsigned integer underflow if vector contains less than 2 elements.

Input

To read a long integer, read its notation into a `string` and then convert it to “digits”:

```
for (int i=(int)s.length(); i>0; i-=9)
    if (i < 9)
        a.push_back (atoi (s.substr (0, i).c_str()));
    else
        a.push_back (atoi (s.substr (i-9, 9).c_str()));
```

If we use an array of `char` instead of a `string`, the code will be even shorter:

```
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 can contain leading zeros, they can be removed as follows:

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

Addition

Increment long integer a by b and store 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

Decrement long integer a by b ($a \geq b$) and store 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();
```

Note that after performing subtraction we remove leading zeros to keep up with the premise that our long integers don’t have leading zeros.

Multiplication by short integer

Multiply long integer a by short integer b ($b < base$) and store 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] * 1ll * b;
    a[i] = int(cur % base);
    carry = int(cur / base);
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Additional optimization: If runtime is extremely important, you can try to replace two divisions with one by finding only integer result of division (variable `carry`) and then use it to find modulo using multiplication. This usually makes the code faster, though not dramatically.

Multiplication by long integer

Multiply long integers a and b and store result 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] * 1ll * (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();
```

Division by short integer

Divide long integer a by short integer b ($b < base$), store integer result in a and 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();
```

6.2.2 Long Integer Arithmetic for Factorization Representation

The idea is to store the integer as its factorization, i.e. the powers of primes which divide it.

This approach is very easy to implement, and allows to do multiplication and division easily (asymptotically faster than the classical method), but not addition or subtraction. It is also very memory-efficient compared to the classical approach.

This method is often used for calculations modulo non-prime number M ; in this case a number is stored as powers of divisors of M which divide the number, plus the remainder modulo M .

6.2.3 Long Integer Arithmetic in prime modulus (Garner Algorithm)

The idea is to choose a set of prime numbers (typically they are small enough to fit into standard integer data type) and to store an integer as a vector of remainders from division of the integer by each of those primes.

Chinese remainder theorem states that this representation is sufficient to uniquely restore any number from 0 to product of these primes minus one. [Garner algorithm](#) allows to restore the number from such representation to normal integer.

This method allows to save memory compared to the classical approach (though the savings are not as dramatic as in factorization representation). Besides, it allows to perform fast addition, subtraction and multiplication in time proportional to the number of prime numbers used as modulus (see [Chinese remainder theorem](#) article for implementation).

The tradeoff is that converting the integer back to normal form is rather laborious and requires implementing classical arbitrary-precision arithmetic with multiplication. Besides, this method doesn't support division.

6.2.4 Fractional Arbitrary-Precision Arithmetic

Fractions occur in programming competitions less frequently than integers, and long arithmetic is much trickier to implement for fractions, so programming competitions feature only a small subset of fractional long arithmetic.

Arithmetic in Irreducible Fractions

A number is represented as an irreducible fraction $\frac{a}{b}$, where a and b are integers. All operations on fractions can be represented as operations on integer numerators and denominators of these fractions. Usually this requires using classical arbitrary-precision arithmetic for storing numerator and denominator, but sometimes a built-in 64-bit integer data type suffices.

Storing Floating Point Position as Separate Type

Sometimes a problem requires handling very small or very large numbers without allowing overflow or underflow. Built-in double data type uses 8-10 bytes and allows values of the exponent in $[-308; 308]$ range, which sometimes might be insufficient.

The approach is very simple: a separate integer variable is used to store the value of the exponent, and after each operation the floating-point number is normalized, i.e. returned to $[0.1; 1)$ interval by adjusting the exponent accordingly.

When two such numbers are multiplied or divided, their exponents should be added or subtracted, respectively. When numbers are added or subtracted, they have to be brought to common exponent first by multiplying one of them by 10 raised to the power equal to the difference of exponent values.

As a final note, the exponent base doesn't have to equal 10. Based on the internal representation of floating-point numbers, it makes most sense to use 2 as the exponent base.

6.2.5 Practice Problems

- [UVA - How Many Fibs?](#)
- [UVA - Product](#)
- [UVA - Maximum Sub-sequence Product](#)
- [SPOJ - Fast Multiplication](#)
- [SPOJ - GCD2](#)
- [UVA - Division](#)
- [UVA - Fibonacci Freeze](#)
- [UVA - Krakovia](#)
- [UVA - Simplifying Fractions](#)
- [UVA - 500!](#)
- [Hackerrank - Factorial digit sum](#)
- [UVA - Immortal Rabbits](#)
- [SPOJ - 0110SS](#)
- [Codeforces - Notepad](#)

6.3 Fast Fourier transform

In this article we will discuss an algorithm that allows us to multiply two polynomials of length n in $O(n \log n)$ time, which is better than the trivial multiplication which takes $O(n^2)$ time. Obviously also multiplying two long numbers can be reduced to multiplying polynomials, so also two long numbers can be multiplied in $O(n \log n)$ time (where n is the number of digits in the numbers).

The discovery of the **Fast Fourier transformation (FFT)** is attributed to Cooley and Tukey, who published an algorithm in 1965. But in fact the FFT has been discovered repeatedly before, but the importance of it was not understood before the inventions of modern computers. Some researchers attribute the discovery of the FFT to Runge and König in 1924. But actually Gauss developed such a method already in 1805, but never published it.

6.3.1 Discrete Fourier transform

Let there be a polynomial of degree $n - 1$:

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

Without loss of generality we assume that n - the number of coefficients - is a power of 2. If n is not a power of 2, then we simply add the missing terms a_ix^i and set the coefficients a_i to 0.

The theory of complex numbers tells us that the equation $x^n = 1$ has n complex solutions (called the n -th roots of unity), and the solutions are of the form $w_{n,k} = e^{\frac{2k\pi i}{n}}$ with $k = 0 \dots n - 1$. Additionally these complex numbers have some very interesting properties: e.g. the principal n -th root $w_n = w_{n,1} = e^{\frac{2\pi i}{n}}$ can be used to describe all other n -th roots: $w_{n,k} = (w_n)^k$.

The **discrete Fourier transform (DFT)** of the polynomial $A(x)$ (or equivalently the vector of coefficients $(a_0, a_1, \dots, a_{n-1})$) is defined as the values of the polynomial at the points $x = w_{n,k}$, i.e. it is the vector:

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

Similarly the **inverse discrete Fourier transform** is defined: The inverse DFT of values of the polynomial $(y_0, y_1, \dots, y_{n-1})$ are the 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 a direct DFT computes the values of the polynomial at the points at the n -th roots, the inverse DFT can restore the coefficients of the polynomial using those values.

Application of the DFT: fast multiplication of polynomials

Let there be two polynomials A and B . We compute the DFT for each of them: $\text{DFT}(A)$ and $\text{DFT}(B)$.

What happens if we multiply these polynomials? Obviously at each point the values are simply multiplied, i.e.

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

This means that if we multiply the vectors $\text{DFT}(A)$ and $\text{DFT}(B)$ - by multiplying each element of one vector by the corresponding element of the other vector - then we get nothing other than the DFT of the polynomial $\text{DFT}(A \cdot B)$:

$$\text{DFT}(A \cdot B) = \text{DFT}(A) \cdot \text{DFT}(B)$$

Finally, applying the inverse DFT, we obtain:

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

On the right the product of the two DFTs we mean the pairwise product of the vector elements. This can be computed in $O(n)$ time. If we can compute the DFT and the inverse DFT in $O(n \log n)$, then we can compute the product of the two polynomials (and consequently also two long numbers) with the same time complexity.

It should be noted, that the two polynomials should have the same degree. Otherwise the two result vectors of the DFT have different length. We can accomplish this by adding coefficients with the value 0.

And also, since the result of the product of two polynomials is a polynomial of degree $2(n-1)$, we have to double the degrees of each polynomial (again by padding 0s). From a vector with n values we cannot reconstruct the desired polynomial with $2n-1$ coefficients.

Fast Fourier Transform

The **fast Fourier transform** is a method that allows computing the DFT in $O(n \log n)$ time. The basic idea of the FFT is to apply divide and conquer. We divide the coefficient vector of the polynomial into two vectors, recursively compute the DFT for each of them, and combine the results to compute the DFT of the complete polynomial.

So let there be a polynomial $A(x)$ with degree $n-1$, where n is a power of 2, and $n > 1$:

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

We divide it into two smaller polynomials, the one containing only the coefficients of the even positions, and the one containing the coefficients of the odd positions:

$$\begin{aligned} A_0(x) &= a_0x^0 + a_2x^1 + \cdots + a_{n-2}x^{\frac{n}{2}-1} \\ A_1(x) &= a_1x^0 + a_3x^1 + \cdots + a_{n-1}x^{\frac{n}{2}-1} \end{aligned}$$

It is easy to see that

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

The polynomials A_0 and A_1 are only half as much coefficients as the polynomial A . If we can compute the $\text{DFT}(A)$ in linear time using $\text{DFT}(A_0)$ and $\text{DFT}(A_1)$, then we get the recurrence $T_{\text{DFT}}(n) = 2T_{\text{DFT}}(\frac{n}{2}) + O(n)$ for the time complexity, which results in $T_{\text{DFT}}(n) = O(n \log n)$ by the **master theorem**.

Lets learn how we can accomplish that.

Suppose we have computed the vectors $(y_k^0)_{k=0}^{n/2-1} = \text{DFT}(A_0)$ and $(y_k^1)_{k=0}^{n/2-1} = \text{DFT}(A_1)$. Let us find a expression for $(y_k)_{k=0}^{n-1} = \text{DFT}(A)$.

For the first $\frac{n}{2}$ values we can just use the previously noted equation $A(x) = A_0(x^2) + xA_1(x^2)$:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \dots \frac{n}{2} - 1.$$

However for the second $\frac{n}{2}$ values we need to find a slightly, different expression:

$$\begin{aligned} y_{k+n/2} &= A(w_n^{k+n/2}) \\ &= A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) \\ &= A_0(w_n^{2k} w_n^n) + w_n^k w_n^{n/2} A_1(w_n^{2k} w_n^n) \\ &= A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) \\ &= y_k^0 - w_n^k y_k^1 \end{aligned}$$

Here we used again $A(x) = A_0(x^2) + xA_1(x^2)$ and the two identities $w_n^n = 1$ and $w_n^{n/2} = -1$.

Therefore we get the desired formulas for computing the whole vector (y_k) :

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

(This pattern $a + b$ and $a - b$ is sometimes called a **butterfly**.)

Thus we learned how to compute the DFT in $O(n \log n)$ time.

Inverse FFT

Let the vector $(y_0, y_1, \dots, y_{n-1})$ - the values of polynomial A of degree $n - 1$ in the points $x = w_n^k$ - be given. We want to restore the coefficients $(a_0, a_1, \dots, a_{n-1})$ of the polynomial. This known problem is called **interpolation**, and there are general algorithms for solving it. But in this special case (since we know the values of the points at the roots of unity), we can obtains a much simpler algorithm (that is practically the same as the direct FFT).

We can write the DFT, according to its definition, in the matrix form:

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

This matrix is called the **Vandermonde matrix**.

Thus we can compute the vector $(a_0, a_1, \dots, a_{n-1})$ by multiplying the vector $(y_0, y_1, \dots, y_{n-1})$ from the left with the inverse of the 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 quick check can verify that the inverse of the matrix has the following form:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}$$

Thus we obtain the formula:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Comparing this to the formula for y_k

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj},$$

we notice that these problems are almost the same, so the coefficients a_k can be found by the same divide and conquer algorithm, as well as the direct FFT, only instead of w_n^k we have to use w_n^{-k} , and at the end we need to divide the resulting coefficients by n .

Thus the computation of the inverse DFT is almost the same as the calculation of the direct DFT, and it also can be performed in $O(n \log n)$ time.

Implementation

Here we present a simple recursive **implementation of the FFT** and the inverse FFT, both in one function, since the difference between the forward and the inverse FFT are so minimal. To store the complex numbers we use the complex type in the C++ STL.

```
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    if (n == 1)
        return;

    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; 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;
    }
}
```

The function gets passed a vector of coefficients, and the function will compute the DFT or inverse DFT and store the result again in this vector. The argument `invert` shows whether the direct or the inverse DFT should be computed. Inside the function we first check if the length of the vector is equal to one, if this is the case then we don't have to do anything. Otherwise we divide the vector *a* into two vectors *a0* and *a1* and compute the DFT for both recursively. Then we initialize the value *wn* and a variable *w*, which will contain the current power of *wn*. Then the values of the resulting DFT are computed using the above formulas.

If the flag `invert` is set, then we replace *wn* with wn^{-1} , and each of the values of the result is divided by 2 (since this will be done in each level of the recursion, this will end up dividing the final values by *n*).

Using this function we can create a function for **multiplying two polynomials**:


```

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}

```

This function works with polynomials with integer coefficients, however you can also adjust it to work with other types. Since there is some error when working with complex numbers, we need round the resulting coefficients at the end.

Finally the function for **multiplying** two long numbers practically doesn't differ from the function for multiplying polynomials. The only thing we have to do afterwards, is to normalize the number:

```

int carry = 0;
for (int i = 0; i < n; i++)
    result[i] += carry;
    carry = result[i] / 10;
    result[i] %= 10;
}

```

Since the length of the product of two numbers never exceed the total length of both numbers, the size of the vector is enough to perform all carry operations.

Improved implementation: in-place computation

To increase the efficiency we will switch from the recursive implementation to an iterative one. In the above recursive implementation we explicitly separated the vector a into two vectors - the element on the even positions got assigned to one temporary vector, and the elements on odd positions to another. However if we reorder the elements in a certain way, we don't need to create these temporary vectors (i.e. all the calculations can be done “in-place”, right in the vector A itself).

Note that at the first recursion level, the elements whose lowest bit of the position was zero got assigned to the vector a_0 , and the ones with a one as the lowest bit of the position got assigned to a_1 . In the second recursion level the

same thing happens, but with the second lowest bit instead, etc. Therefore if we reverse the bits of the position of each coefficient, and sort them by these reversed values, we get the desired order (it is called the bit-reversal permutation).

For example the desired order for $n = 8$ has the form:

$$a = \{[(a_0, a_4), (a_2, a_6)], [(a_1, a_5), (a_3, a_7)]\}$$

Indeed in the first recursion level (surrounded by curly braces), the vector gets divided into two parts $[a_0, a_2, a_4, a_6]$ and $[a_1, a_3, a_5, a_7]$. As we see, in the bit-reversal permutation this corresponds to simply dividing the vector into two halves: the first $\frac{n}{2}$ elements and the last $\frac{n}{2}$ elements. Then there is a recursive call for each half. Let the resulting DFT for each of them be returned in place of the elements themselves (i.e. the first half and the second half of the vector a respectively).

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

Now we want to combine the two DFTs into one for the complete vector. The order of the elements is ideal, and we can also perform the union directly in this vector. We can take the elements y_0^0 and y_0^1 and perform the butterfly transform. The place of the resulting two values is the same as the place of the two initial values, so we get:

$$a = \{[y_0^0 + w_n^0 y_0^1, y_1^0 + w_n^0 y_1^1], [y_0^0 - w_n^0 y_0^1, y_1^0 - w_n^0 y_1^1]\}$$

Similarly we can compute the butterfly transform of y_1^0 and y_1^1 and put the results in their place, and so on. As a result we get:

$$a = \{[y_0^0 + w_n^0 y_0^1, y_1^0 + w_n^1 y_1^1, y_2^0 + w_n^2 y_2^1, y_3^0 + w_n^3 y_3^1], [y_0^0 - w_n^0 y_0^1, y_1^0 - w_n^1 y_1^1, y_2^0 - w_n^2 y_2^1, y_3^0 - w_n^3 y_3^1]\}$$

Thus we computed the required DFT from the vector a .

Here we described the process of computing the DFT only at the first recursion level, but the same works obviously also for all other levels. Thus, after applying the bit-reversal permutation, we can compute the DFT in-place, without any additional memory.

This additionally allows us to get rid of the recursion. We just start at the lowest level, i.e. we divide the vector into pairs and apply the butterfly transform to them. This results with the vector a with the work of the last level applied. In the next step we divide the vector into vectors of size 4, and again apply the butterfly transform, which gives us the DFT for each block of size 4. And so on. Finally in the last step we obtained the result of the DFTs of both halves of a , and by applying the butterfly transform we obtain the DFT for the complete vector a .

```
using cd = complex<double>;
const double PI = acos(-1);

int reverse(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<cd> & a, bool invert) {
    int n = a.size();
    int lg_n = 0;
    while ((1 << lg_n) < n)
        lg_n++;

    for (int i = 0; i < n; i++) {
        if (i < reverse(i, lg_n))
            swap(a[i], a[reverse(i, lg_n)]);
    }

    for (int len = 2; len <= n; len <= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd 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 (cd & x : a)
            x /= n;
    }
}

```

At first we apply the bit-reversal permutation by swapping the each element with the element of the reversed position. Then the $\log n - 1$ states of the algorithm we compute the DFT for each block of the corresponding size len . For all those blocks we have the same root of unity w_{len} . We iterate all blocks and perform the butterfly transform on each of them.

We can further optimize the reversal of the bits. In the previous implementation we iterated all bits of the index and created the bitwise reversed index. However we can reverse the bits in a different way.

Suppose that j already contains the reverse of i . Then by to go to $i + 1$, we have to increment i , and we also also have to increment j , but in a “reversed” number system. Adding one in the conventional binary system is equivalent to flip all trailing ones into zeros and flipping the zero right before them into a one. Equivalently in the “reversed” number system, we flip all leading ones, and the

also the next zero.

Thus we get the following implementation:

```
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
    int n = 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);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd 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 (cd & x : a)
            x /= n;
    }
}
```

Additionally we can precompute the bit-reversal permutation beforehand. This is especially useful when the size n is the same for all calls. But even when we only have three calls (which are necessary for multiplying two polynomials), the effect is noticeable. Also we can precompute all roots of unity and their powers.

6.3.2 Number theoretic transform

Now we switch the objective a little bit. We still want to multiply two polynomials in $O(n \log n)$ time, but this time we want to compute the coefficients modulo some prime number p . Of course for this task we can use the normal DFT and apply the modulo operator to the result. However, doing so might lead

to rounding errors, especially when dealing with large numbers. The **number theoretic transform (NTT)** has the advantage, that it only works with integer, and therefore the result are guaranteed to be correct.

The discrete Fourier transform is based on complex numbers, and the n -th roots of unity. To efficiently compute it, we extensively use properties of the roots (e.g. that there is one root that generates all other roots by exponentiation).

But the same properties hold for the n -th roots of unity in modular arithmetic. A n -th root of unity under a primitive field is such a number w_n that satisfies:

$$\begin{aligned}(w_n)^n &= 1 \pmod{p}, \\ (w_n)^k &\neq 1 \pmod{p}, \quad 1 \leq k < n.\end{aligned}$$

The other $n - 1$ roots can be obtained as powers of the root w_n .

To apply it in the fast Fourier transform algorithm, we need a root to exist for some n , which is a power of 2, and also for all smaller powers. We can notice the following interesting property:

$$\begin{aligned}(w_n^2)^m &= w_n^n = 1 \pmod{p}, \quad \text{with } m = \frac{n}{2} \\ (w_n^2)^k &= w_n^{2k} \neq 1 \pmod{p}, \quad 1 \leq k < m.\end{aligned}$$

Thus if w_n is a n -th root of unity, then w_n^2 is a $\frac{n}{2}$ -th root of unity. And consequently for all smaller powers of two there exist roots of the required degree, and they can be computed using w_n .

For computing the inverse DFT, we need need the inverse w_n^{-1} of w_n . But for a prime modulus the inverse always exists.

Thus all the properties that we need from the complex roots are also available in modular arithmetic, provided that we have a large enough module p for which a n -th root of unity exists.

For example we can take the following values: module $p = 7340033$, $w_{2^{20}} = 5$. If this module is not enough, we need to find a different pair. We can use that fact that for modules of the form $p = c2^k + 1$ (and p is prime), there always exists the 2^k -th root of unity. It can be shown that g^c is such a 2^k -th root of unity, where g is a [primitive root](#) of p .

```
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 = 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)(1LL * wlen * 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)(1LL * a[i+j+len/2] * 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)(1LL * w * wlen % mod);
            }
        }
    }

    if (invert) {
        int n_1 = inverse(n, mod);
        for (int & x : a)
            x = (int)(1LL * x * n_1 % mod);
    }
}

```

Here the function `inverse` computes the modular inverse (see [Modular Multiplicative Inverse](#)). The constants `mod`, `root`, `root_pw` determine the module and the root, and `root_1` is the inverse of `root` modulo `mod`.

In practice this implementation is slower than the implementation using complex numbers (due to the huge number of modulo operations), but it has some advantages such as less memory usage and no rounding errors.

6.3.3 Multiplication with arbitrary modulus

Here we want to achieve the same goal as in previous section. Multiplying two polynomial $A(x)$ and $B(x)$, and computing the coefficients modulo some number M . The number theoretic transform only works for certain prime numbers. What about the case when the modulus is not of the desired form?

One option would be to perform multiple number theoretic transforms with different prime numbers of the form $c2^k + 1$, then apply the [Chinese Remainder Theorem](#) to compute the final coefficients.

Another options is to distribute the polynomials $A(x)$ and $B(x)$ into two smaller polynomials each

$$\begin{aligned}
 A(x) &= A_1(x) + A_2(x) \cdot C \\
 B(x) &= B_1(x) + B_2(x) \cdot C
 \end{aligned}$$

with $C \approx \sqrt{M}$.

Then the product of $A(x)$ and $B(x)$ can then be represented as:

$$A(x) \cdot B(x) = A_1(x) \cdot B_1(x) + (A_1(x) \cdot B_2(x) + A_2(x) \cdot B_1(x)) \cdot C + (A_2(x) \cdot B_2(x)) \cdot C^2$$

The polynomials $A_1(x)$, $A_2(x)$, $B_1(x)$ and $B_2(x)$ contain only coefficients smaller than \sqrt{M} , therefore the coefficients of all the appearing products are smaller than $M \cdot n$, which is usually small enough to handle with typical floating point types.

This approach therefore requires computing the products of polynomials with smaller coefficients (by using the normal FFT and inverse FFT), and then the original product can be restored using modular addition and multiplication in $O(n)$ time.

6.3.4 Applications

DFT can be used in a huge variety of other problems, which at the first glance have nothing to do with multiplying polynomials.

All possible sums

We are given two arrays $a[]$ and $b[]$. We have to find all possible sums $a[i] + b[j]$, and for each sum count how often it appears.

For example for $a = [1, 2, 3]$ and $b = [2, 4]$ we get: then sum 3 can be obtained in 1 way, the sum 4 also in 1 way, 5 in 2, 6 in 1, 7 in 1.

We construct for the arrays a and b two polynomials A and B . The numbers of the array will act as the exponents in the polynomial ($a[i] \Rightarrow x^{a[i]}$); and the coefficients of this term will be how often the number appears in the array.

Then, by multiplying these two polynomials in $O(n \log n)$ time, we get a polynomial C , where the exponents will tell us which sums can be obtained, and the coefficients tell us how often. To demonstrate this on the example:

$$(1x^1 + 1x^2 + 1x^3)(1x^2 + 1x^4) = 1x^3 + 1x^4 + 2x^5 + 1x^6 + 1x^7$$

All possible scalar products

We are given two arrays $a[]$ and $b[]$ of length n . We have to compute the products of a with every cyclic shift of b .

We generate two new arrays of size $2n$: We reverse a and append n zeros to it. And we just append b to itself. When we multiply these two arrays as polynomials, and look at the coefficient $c[n-1]$, $c[n]$, $c[2n-2]$ of the product c , we get:

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

And since all the elements $a[i] = 0$ for $i \geq n$:

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

It is easy to see that this sum is just the scalar product of the vector a with the $(k - (n - 1))$ -th cyclic left shift. Thus these coefficients are the answer to the problem, and we were still able to obtain it in $O(n \log n)$ time. Note here that $c[2n - 1]$ also gives us the n th cyclic shift but that is the same as the 0th cyclic shift so we don't need to consider that separately into our answer.

Two stripes

We are given two Boolean stripes (cyclic arrays of values 0 and 1) a and b . We want to find all ways to attach the first stripe to the second one, such that at no position we have a 1 of the first stripe next to a 1 of the second stripe.

The problem doesn't actually differ much from the previous problem. Attaching two stripes just means that we perform a cyclic shift on the second array, and we can attach the two stripes, if scalar product of the two arrays is 0.

String matching

We are given two strings, a text T and a pattern P , consisting of lowercase letters. We have to compute all the occurrences of the pattern in the text.

We create a polynomial for each string ($T[i]$ and $P[i]$ are numbers between 0 and 25 corresponding to the 26 letters of the alphabet):

$$A(x) = a_0x^0 + a_1x^1 + \cdots + a_{n-1}x^{n-1}, \quad n = |T|$$

with

$$a_i = \cos(\alpha_i) + i \sin(\alpha_i), \quad \alpha_i = \frac{2\pi T[i]}{26}.$$

And

$$B(x) = b_0x^0 + b_1x^1 + \cdots + b_{m-1}x^{m-1}, \quad m = |P|$$

with

$$b_i = \cos(\beta_i) - i \sin(\beta_i), \quad \beta_i = \frac{2\pi P[m - i - 1]}{26}.$$

Notice that with the expression $P[m - i - 1]$ explicitly reverses the pattern.

The $(m - 1 + i)$ th coefficients of the product of the two polynomials $C(x) = A(x) \cdot B(x)$ will tell us, if the pattern appears in the text at position i .

$$c_{m-1+i} = \sum_{j=0}^{m-1} a_{i+j} \cdot b_{m-1-j} = \sum_{j=0}^{m-1} (\cos(\alpha_{i+j}) + i \sin(\alpha_{i+j})) \cdot (\cos(\beta_j) - i \sin(\beta_j))$$

with $\alpha_{i+j} = \frac{2\pi T[i+j]}{26}$ and $\beta_j = \frac{2\pi P[j]}{26}$

If there is a match, than $T[i + j] = P[j]$, and therefore $\alpha_{i+j} = \beta_j$. This gives (using the Pythagorean trigonometric identity):

$$\begin{aligned} c_{m-1+i} &= \sum_{j=0}^{m-1} (\cos(\alpha_{i+j}) + i \sin(\alpha_{i+j})) \cdot (\cos(\alpha_{i+j}) - i \sin(\alpha_{i+j})) \\ &= \sum_{j=0}^{m-1} \cos(\alpha_{i+j})^2 + \sin(\alpha_{i+j})^2 = \sum_{j=0}^{m-1} 1 = m \end{aligned}$$

If there isn't a match, then at least a character is different, which leads that one of the products $a_{i+1} \cdot b_{m-1-j}$ is not equal to 1, which leads to the coefficient $c_{m-1+i} \neq m$.

String matching with wildcards

This is an extension of the previous problem. This time we allow that the pattern contains the wildcard character $*$, which can match every possible letter. E.g. the pattern $a * c$ appears in the text $abccaacc$ at exactly three positions, at index 0, index 4 and index 5.

We create the exact same polynomials, except that we set $b_i = 0$ if $P[m - i - 1] = *$. If x is the number of wildcards in P , then we will have a match of P in T at index i if $c_{m-1+i} = m - x$.

6.3.5 Practice problems

- [SPOJ - POLYMUL](#)
- [SPOJ - MAXMATCH](#)
- [SPOJ - ADAMATCH](#)
- [Codeforces - Yet Another String Matching Problem](#)
- [Codeforces - Lightsabers \(hard\)](#)
- [Kattis - K-Inversions](#)
- [Codeforces - Dasha and cyclic table](#)

Part II

Data Structures

Chapter 1

Fundamentals

1.1 Minimum stack / Minimum queue

In this article we will consider three problems: first we will modify a stack in a way that allows up to find the smallest element of the stack in $O(1)$, then we will do the same thing with a queue, and finally we will use these data structures to find the minimum in all subarrays of a fixed length in an array in $O(n)$

1.1.1 Stack modification

We want to modify the stack data structure in such a way, that it possible to find the smallest element in the stack in $O(1)$ time, while maintaining the same asymptotic behavior for adding and removing elements from the stack. Quick reminder, on a stack we only add and remove elements on one end.

To do this, we will not only store the elements in the stack, but we will store them in pairs: the element itself and the minimum in the stack starting from this element and below.

```
stack<pair<int, int>> st;
```

It is clear that finding the minimum in the whole stack consists only of looking at the value `stack.top().second`.

It is also obvious that adding or removing a new element to the stack can be done in constant time.

Implementation:

- Adding an element:

```
int new_min = st.empty() ? new_elem : min(new_elem, st.top().second);
st.push({new_elem, new_min});
```

- Removing an element:

```
int removed_element = st.top().first;
st.pop();
```

- Finding the minimum:

```
int minimum = st.top().second;
```

1.1.2 Queue modification (method 1)

Now we want to achieve the same operations with a queue, i.e. we want to add elements at the end and remove them from the front.

Here we consider a simple method for modifying a queue. It has a big disadvantage though, because the modified queue will actually not store all elements.

The key idea is to only store the items in the queue that are needed to determine the minimum. Namely we will keep the queue in nondecreasing order (i.e. the smallest value will be stored in the head), and of course not in any arbitrary way, the actual minimum has to be always contained in the queue. This way the smallest element will always be in the head of the queue. Before adding a new element to the queue, it is enough to make a “cut”: we will remove all trailing elements of the queue that are larger than the new element, and afterwards add the new element to the queue. This way we don’t break the order of the queue, and we will also not lose the current element if it is at any subsequent step the minimum. All the elements that we removed can never be a minimum itself, so this operation is allowed. When we want to extract an element from the head, it actually might not be there (because we removed it previously while adding a smaller element). Therefore when deleting an element from a queue we need to know the value of the element. If the head of the queue has the same value, we can safely remove it, otherwise we do nothing.

Consider the implementations of the above operations:

```
deque<int> q;
```

- Finding the minimum:

```
int minimum = q.front();
```

- Adding an element:

```
while (!q.empty() && q.back() > new_element)
    q.pop_back();
q.push_back(new_element);
```

- Removing an element:

```
if (!q.empty() && q.front() == remove_element)
    q.pop_front();
```

It is clear that on average all these operation only take $O(1)$ time (because every element can only be pushed and popped once).

1.1.3 Queue modification (method 2)

This is a modification of method 1. We want to be able to remove elements without knowing which element we have to remove. We can accomplish that by storing the index for each element in the queue. And we also remember how many elements we already have added and removed.

```
deque<pair<int, int>> q;
int cnt_added = 0;
int cnt_removed = 0;
```

- Finding the minimum:

```
int minimum = q.front().first;
```

- Adding an element:

```
while (!q.empty() && q.back().first > new_element)
    q.pop_back();
q.push_back({new_element, cnt_added});
cnt_added++;
```

- Removing an element:

```
if (!q.empty() && q.front().second == cnt_removed)
    q.pop_front();
cnt_removed++;
```

1.1.4 Queue modification (method 3)

Here we consider another way of modifying a queue to find the minimum in $O(1)$. This way is somewhat more complicated to implement, but this time we actually store all elements. And we also can remove an element from the front without knowing its value.

The idea is to reduce the problem to the problem of stacks, which was already solved by us. So we only need to learn how to simulate a queue using two stacks.

We make two stacks, **s1** and **s2**. Of course these stack will be of the modified form, so that we can find the minimum in $O(1)$. We will add new elements to the stack **s1**, and remove elements from the stack **s2**. If at any time the stack **s2** is empty, we move all elements from **s1** to **s2** (which essentially reverses the order of those elements). Finally finding the minimum in a queue involves just finding the minimum of both stacks.

Thus we perform all operations in $O(1)$ on average (each element will be once added to stack **s1**, once transferred to **s2**, and once popped from **s2**)

Implementation:

```
stack<pair<int, int>> s1, s2;
```

- Finding the minimum:

```
if (s1.empty() || s2.empty())
    minimum = s1.empty() ? s2.top().second : s1.top().second;
else
    minimum = min(s1.top().second, s2.top().second());
```

- Add element:

```
int minimum = s1.empty() ? new_element : min(new_element, s1.top().second);
s1.push({new_element, minimum});
```

- Removing an element:

```
if (s2.empty()) {
    while (!s1.empty()) {
        int element = s1.top.first();
        s1.pop();
        int minimum = s2.empty() ? element : min(element, s2.top().second);
        s2.push({element, minimum});
    }
}
int remove_element = s2.top().first;
s2.pop();
```

1.1.5 Finding the minimum for all subarrays of fixed length

Suppose we are given an array A of length N and a given $M \leq N$. We have to find the minimum of each subarray of length M in this array, i.e. we have to find:

$$\min_{0 \leq i \leq M-1} A[i], \min_{1 \leq i \leq M} A[i], \min_{2 \leq i \leq M+1} A[i], \dots, \min_{N-M \leq i \leq N-1} A[i]$$

We have to solve this problem in linear time, i.e. $O(n)$.

We can use any of the three modified queues to solve the problem. The solutions should be clear: we add the first M element of the array, find and output its minimum, then add the next element to the queue and remove the first element of the array, find and output its minimum, etc. Since all operations with the queue are performed in constant time on average, the complexity of the whole algorithm will be $O(n)$.

1.2 Sparse Table

Sparse Table is a data structure, that allows answering range queries. It can answer most range queries in $O(\log n)$, but its true power is answering range minimum queries (or equivalent range maximum queries). For those queries it can compute the answer in $O(1)$ time.

The only drawback of this data structure is, that it can only be used on *immutable* arrays. This means, that the array cannot be changed between two queries. If any element in the array changes, the complete data structure has to be recomputed.

1.2.1 Intuition

Any non-negative number can be uniquely represented as a sum of decreasing powers of two. This is just a variant of the binary representation of a number. E.g. $13 = (1101)_2 = 8 + 4 + 1$. For a number x there can be at most $\lceil \log_2 x \rceil$ summands.

By the same reasoning any interval can be uniquely represented as a union of intervals with lengths that are decreasing powers of two. E.g. $[2, 14] = [2, 9] \cup [10, 13] \cup [14, 14]$, where the complete interval has length 13, and the individual intervals have the lengths 8, 4 and 1 respectively. And also here the union consists of at most $\lceil \log_2(\text{length of interval}) \rceil$ many intervals.

The main idea behind Sparse Tables is to precompute all answers for range queries with power of two length. Afterwards a different range query can be answered by splitting the range into ranges with power of two lengths, looking up the precomputed answers, and combining them to receive a complete answer.

1.2.2 Precomputation

We will use a 2-dimensional array for storing the answers to the precomputed queries. $st[i][j]$ will store the answer for the range $[i, i + 2^j - 1]$ of length 2^j . The size of the 2-dimensional array will be $MAXN \times (K + 1)$, where $MAXN$ is the biggest possible array length. K has to satisfy $K \geq \lfloor \log_2 MAXN \rfloor + 1$, because $2^{\lfloor \log_2 MAXN \rfloor}$ is the biggest power of two range, that we have to support. For arrays with reasonable length ($\leq 10^7$ elements), $K = 25$ is a good value.

```
int st[MAXN][K + 1];
```

Because the range $[i, i + 2^j - 1]$ of length 2^j splits nicely into the ranges $[i, i + 2^{j-1} - 1]$ and $[i + 2^{j-1}, i + 2^j - 1]$, both of length 2^{j-1} , we can generate the table efficiently using dynamic programming:

```
for (int i = 0; i < N; i++)
    st[i][0] = f(array[i]);

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = f(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);
```

The function f will depend on the type of query. For range sum queries it will compute the sum, for range minimum queries it will compute the minimum.

The time complexity of the precomputation is $O(N \log N)$.

1.2.3 Range Sum Queries

For this type of queries, we want to find the sum of all values in a range. Therefore the natural definition of the function f is $f(x, y) = x + y$. We can construct the data structure with:

```
long long st[MAXN][K];

for (int i = 0; i < N; i++)
    st[i][0] = array[i];

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = st[i][j-1] + st[i + (1 << (j - 1))][j - 1];
```

To answer the sum query for the range $[L, R]$, we iterate over all powers of two, starting from the biggest one. As soon as a power of two 2^j is smaller or equal to the length of the range ($= R - L + 1$), we process the first part of range $[L, L + 2^j - 1]$, and continue with the remaining range $[L + 2^j, R]$.

```
long long sum = 0;
for (int j = K; j >= 0; j--) {
    if ((1 << j) <= R - L + 1) {
        sum += st[L][j];
        L += 1 << j;
    }
}
```

Time complexity for a Range Sum Query is $O(K) = O(\log \text{MAXN})$.

1.2.4 Range Minimum Queries (RMQ)

These are the queries where the Sparse Table shines. When computing the minimum of a range, it doesn't matter if we process a value in the range once or twice. Therefore instead of splitting a range into multiple ranges, we can also split the range into only two overlapping ranges with power of two length. E.g. we can split the range $[1, 6]$ into the ranges $[1, 4]$ and $[3, 6]$. The range minimum of $[1, 6]$ is clearly the same as the minimum of the range minimum of $[1, 4]$ and the range minimum of $[3, 6]$. So we can compute the minimum of the range $[L, R]$ with:

$$\min(\text{st}[L][j], \text{st}[R - 2^j + 1][j]) \quad \text{where } j = \log_2(R - L + 1)$$

This requires that we are able to compute $\log_2(R - L + 1)$ fast. You can accomplish that by precomputing all logarithms:


```

int log[MAXN+1];
log[1] = 0;
for (int i = 2; i <= MAXN; i++)
    log[i] = log[i/2] + 1;

```

Afterwards we need to precompute the Sparse Table structure. This time we define f with $f(x, y) = \min(x, y)$.

```

int st[MAXN][K];

for (int i = 0; i < N; i++)
    st[i][0] = array[i];

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = min(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);

```

And the minimum of a range $[L, R]$ can be computed with:

```

int j = log[R - L + 1];
int minimum = min(st[L][j], st[R - (1 << j) + 1][j]);

```

Time complexity for a Range Minimum Query is $O(1)$.

1.2.5 Similar data structures supporting more types of queries

One of the main weakness of the $O(1)$ approach discussed in the previous section is, that this approach only supports queries of [idempotent functions](#). I.e. it works great for range minimum queries, but it is not possible to answer range sum queries using this approach.

There are similar data structures that can handle any type of associative functions and answer range queries in $O(1)$. One of them is called [Disjoint Sparse Table](#). Another one would be the [Sqrt Tree](#).

1.2.6 Practice Problems

- [SPOJ - RMQSQ](#)
- [SPOJ - THRBL](#)
- [Codechef - MSTICK](#)
- [Codechef - SEAD](#)
- [Codeforces - CGCDSSQ](#)
- [Codeforces - R2D2 and Droid Army](#)
- [Codeforces - Maximum of Maximums of Minimums](#)
- [SPOJ - Miraculous](#)
- [DevSkills - Multiplication Interval](#)
- [Codeforces - Animals and Puzzles](#)
- [Codeforces - Trains and Statistics](#)
- [SPOJ - Postering](#)
- [SPOJ - Negative Score](#)

- [SPOJ - A Famous City](#)
- [SPOJ - Diferencija](#)
- [Codeforces - Turn off the TV](#)
- [Codeforces - Map](#)
- [Codeforces - Awards for Contestants](#)
- [Codeforces - Longest Regular Bracket Sequence](#)

Chapter 2

Trees

2.1 Disjoint Set Union

This article discusses the data structure **Disjoint Set Union** or **DSU**. Often it is also called **Union Find** because of its two main operations.

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, it can create a set from a new element.

Thus the basic interface of this data structure consists of only three operations:

- `make_set(v)` - creates a new set consisting of the new element `v`
- `union_sets(a, b)` - merges the two specified sets (the set in which the element `a` is located, and the set in which the element `b` is located)
- `find_set(v)` - returns the representative (also called leader) of the set that contains the element `v`. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after `union_sets` calls). This representative can be used to check if two elements are part of the same set or not. `a` and `b` are exactly in the same set, if `find_set(a) == find_set(b)`. Otherwise they are in different sets.

As described in more detail later, the data structure allows you to do each of these operations in almost $O(1)$ time on average.

Also in one of the subsections an alternative structure of a DSU is explained, which achieves a slower average complexity of $O(\log n)$, but can be more powerful than the regular DSU structure.

2.1.1 Build an efficient data structure

We will store the sets in the form of **trees**: each tree will correspond to one set. And the root of the tree will be the representative/leader of the set.

In the following image you can see the representation of such trees.

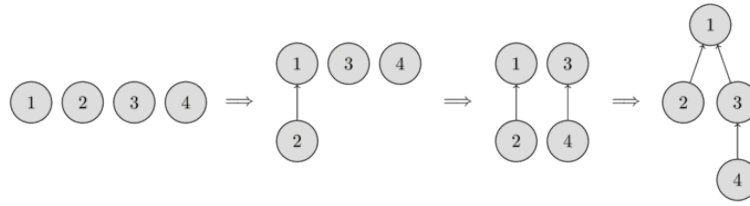


Figure 2.1: Example-image of the set representation with trees

At the beginning every element starts as a single set, therefore each vertex is its own tree. Then we combine the set containing the element 1 and the set containing the element 2. Then we combine the set containing the element 3 and the set containing the element 4. And in the last step we can the sets containing the elements 1 and 3 are merged.

For the implementation this means that we will have to maintain an array **parent** that stores a reference to its immediate ancestor in the tree.

Naive implementation

We can already write the first implementation of the Disjoint Set Union data structure. It will be pretty inefficient at first, but later we can improve it using two optimizations, so that it will take nearly constant time for each function call.

As we said, all the information about the sets of elements will be kept in an array **parent**.

To create a new set (operation **make_set(v)**), we simply create a tree with root in the vertex **v**, meaning that it is its own ancestor.

To combine two sets (operation **union_sets(a, b)**), we first find the representative of the set in which **a** is located, and the representative of the set in which **b** is located. If the representatives are identical, that we have nothing to do, the sets are already merged. Otherwise, we can simply specify that one of the representatives is the parent of the other representative - thereby combining the two trees.

Finally the implementation of the find representative function (operation **find_set(v)**): we simply climb the ancestors of the vertex **v** until we reach the root, i.e. a vertex such that the reference to the ancestor leads to itself. This operation is easily implemented recursively.

```
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 this implementation is inefficient. It is easy to construct an example, so that the trees degenerate into long chains. In that case each call `find_set(v)` can take $O(n)$ time.

This is far away from the complexity that we want to have (nearly constant time). Therefore we will consider two optimizations that will allow to significantly accelerate the work.

Path compression optimization

This optimization is designed for speeding up `find_set`.

If we call `find_set(v)` for some vertex v , we actually find the representative p for all vertices that we visit on the path between v and the actual representative p . The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to p .

You can see the operation in the following image. On the left there is a tree, and on the right side there is the compressed tree after calling `find_set(7)`, which shortens the paths for the visited nodes 7, 5, 3 and 2.

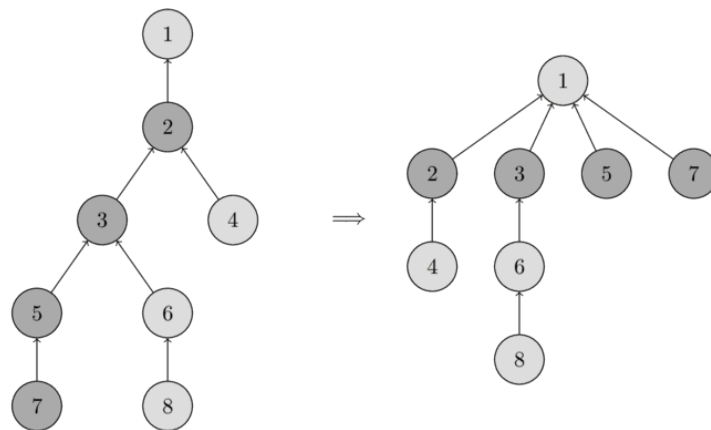


Figure 2.2: Path compression of call `find_set(7)`

The new implementation of `find_set` is as follows:

```

int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}

```

The simple implementation does what was intended: first find the representative of the set (root vertex), and then in the process of stack unwinding the visited nodes are attached directly to the representative.

This simple modification of the operation already achieves the time complexity $O(\log n)$ per call on average (here without proof). There is a second modification, that will make it even faster.

Union by size / rank

In this optimization we will change the `union_set` operation. To be precise, we will change which tree gets attached to the other one. In the native implementation the second tree always got attached to the first one. In practice that can lead to trees containing chains of length $O(n)$. With this optimization we will avoid this by choosing very carefully which tree gets attached.

There are many possible heuristics that can be used. Most popular are the following two approaches: In the first approach we use the size of the trees as rank, and in the second one we use the depth of the tree (more precisely, the upper bound on the tree depth, because the depth will get smaller when applying path compression).

In both approaches the essence of the optimization is the same: we attach the tree with the lower rank to the one with the bigger rank.

Here is the implementation of union by size:

```
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];
    }
}
```

And here is the implementation of union by rank based on the depth of the 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 optimizations are equivalent in terms of time and space complexity. So in practice you can use any of them.

Randomized linking

This optimization may be a substitute for the union by size / rank. Basically you will randomly attach a tree to the other instead of choosing based on the size/rank of each tree.

The complexity of this optimization is more difficult to be analyzed, but it can be proved that the complexity is the same of the union by rank. You can read more about [here](#).

Since you will not need to store any additional information over the sets, this implementation will be cleaner and easier than the implementation of union by rank.

```

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rand() % 2)
            swap(a, b);
        parent[b] = a;
    }
}

```

Time complexity

As mentioned before, if we combine both optimizations - path compression with union by size / rank - we will reach nearly constant time queries. It turns out, that the final amortized time complexity is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very slowly. In fact it grows so slowly, that it doesn't exceed 4 for all reasonable n (approximately $n < 10^{600}$).

Amortized complexity is the total time per operation, evaluated over a sequence of multiple operations. The idea is to guarantee the total time of the entire sequence, while allowing single operations to be much slower than the amortized time. E.g. in our case a single call might take $O(\log n)$ in the worst case, but if we do m such calls back to back we will end up with an average time of $O(\alpha(n))$.

We will also not present a proof for this time complexity, since it is quite long and complicated.

Also, it's worth mentioning that DSU with union by size / rank, but without path compression works in $O(\log n)$ time per query.

2.1.2 Applications and various improvements

In this section we consider several applications of the data structure, both the trivial uses and some improvements to the data structure.

Connected components in a graph

This is one of the obvious applications of DSU.

Formally the problem is defined in the following way: Initially we have an empty graph. We have to add vertices and undirected edges, and answer queries of the form (a, b) - "are the vertices a and b in the same connected component of the graph?"

Here we can directly apply the data structure, and get a solution that handles an addition of a vertex or an edge and a query in nearly constant time on average.

This application is quite important, because nearly the same problem appears in [Kruskal's algorithm for finding a minimum spanning tree](#). Using DSU we can [improve](#) the $O(m \log n + n^2)$ complexity to $O(m \log n)$.

Search for connected components in an image

One of the applications of DSU is the following task: there is an image of $n \times m$ pixels. Originally all are white, but then a few black pixels are drawn. You want to determine the size of each white connected component in the final image.

For the solution we simply iterate over all white pixels in the image, for each cell iterate over its four neighbors, and if the neighbor is white call `union_sets`. Thus we will have a DSU with nm nodes corresponding to image pixels. The resulting trees in the DSU are the desired connected components.

The problem can also be solved by [DFS](#) or [BFS](#), but the method described here has an advantage: it can process the matrix row by row (i.e. to process a row we only need the previous and the current row, and only need a DSU built for the elements of one row) in $O(\min(n, m))$ memory.

Store additional information for each set

DSU allows you to easily store additional information in the sets.

A simple example is the size of the sets: storing the sizes was already described in the Union by size section (the information was stored by the current representative of the set).

In the same way - by storing it at the representative nodes - you can also store any other information about the sets.

Compress jumps along a segment / Painting subarrays offline

One common application of the DSU is the following: There is a set of vertices, and each vertex has an outgoing edge to another vertex. With DSU you can find

the end point, to which we get after following all edges from a given starting point, in almost constant time.

A good example of this application is the **problem of painting subarrays**. We have a segment of length L , each element initially has the color 0. We have to repaint the subarray $[l; r]$ with the color c for each query (l, r, c) . At the end we want to find the final color of each cell. We assume that we know all the queries in advance, i.e. the task is offline.

For the solution we can make a DSU, which for each cell stores a link to the next unpainted cell. Thus initially each cell points to itself. After painting one requested repaint of a segment, all cells from that segment will point to the cell after the segment.

Now to solve this problem, we consider the queries **in the reverse order**: from last to first. This way when we execute a query, we only have to paint exactly the unpainted cells in the subarray $[l; r]$. All other cells already contain their final color. To quickly iterate over all unpainted cells, we use the DSU. We find the left-most unpainted cell inside of a segment, repaint it, and with the pointer we move to the next empty cell to the right.

Here we can use the DSU with path compression, but we cannot use union by rank / size (because it is important who becomes the leader after the merge). Therefore the complexity will be $O(\log n)$ per union (which is also quite fast).

Implementation:

```
for (int i = 0; i < L; i++) {
    make_set(i);
}

for (int i = m-1; i >= 0; i--) {
    int l = query[i].l;
    int r = query[i].r;
    int c = query[i].c;
    for (int v = find_set(l); v <= r; v = find_set(v)) {
        answer[v] = c;
        parent[v] = v + 1;
    }
}
```

There is one optimization: We can use **union by rank**, if we store the next unpainted cell in an additional array `end[]`. Then we can merge two sets into one ranked according to their heuristics, and we obtain the solution in $O(\alpha(n))$.

Support distances up to representative

Sometimes in specific applications of the DSU you need to maintain the distance between a vertex and the representative of its set (i.e. the path length in the tree from the current node to the root of the tree).

If we don't use path compression, the distance is just the number of recursive calls. But this will be inefficient.

However it is possible to do path compression, if we store the **distance to the parent** as additional information for each node.

In the implementation it is convenient to use an array of pairs for `parent[]` and the function `find_set` now returns two numbers: the representative of the set, 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 the parity of the path length / Checking bipartiteness online

In the same way as computing the path length to the leader, it is possible to maintain the parity of the length of the path before him. Why is this application in a separate paragraph?

The unusual requirement of storing the parity of the path comes up in the following task: initially we are given an empty graph, it can be added edges, and we have to answer queries of the form “is the connected component containing this vertex **bipartite**?”.

To solve this problem, we make a DSU for storing of the components and store the parity of the path up to the representative for each vertex. Thus we can quickly check if adding an edge leads to a violation of the bipartiteness or not: namely if the ends of the edge lie in the same connected component and have the same parity length to the leader, then adding this edge will produce a cycle of odd length, and the component will lose the bipartiteness property.

The only difficulty that we face is to compute the parity in the `union_find` method.

If we add an edge (a, b) that connects two connected components into one, then when you attach one tree to another we need to adjust the parity.

Lets derive a formula, which computes the parity issued to the leader of the set that will get attached to another set. Let x be the parity of the path length from vertex a up to its leader A , and y as the parity of the path length from vertex b up to its leader B , and t the desired parity that we have to assign to B after the merge. The path contains the of the three parts: from B to b , from b to a , which is connected by one edge and therefore has parity 1, and from a to A . Therefore we receive the formula (\oplus denotes the XOR operation):

$$t = x \oplus y \oplus 1$$

Thus regardless of how many joins we perform, the parity of the edges is carried from on leader to another.

We give the implementation of the DSU that supports parity. As in the previous section we use a pair to store the ancestor and the parity. 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];
}

```

Offline RMQ (range minimum query) in $O(\alpha(n))$ on average / Arpa's trick

We are given an array `a[]` and we have to compute some minima in given segments of the array.

The idea to solve this problem with DSU is the following: We will iterate over the array and when we are at the `i`th element we will answer all queries (`L, R`) with `R == i`. To do this efficiently we will keep a DSU using the first `i` elements with the following structure: the parent of an element is the next smaller element to the right of it. Then using this structure the answer to a query will be the `a[find_set(L)]`, the smallest number to the right of `L`.

This approach obviously only works offline, i.e. if we know all queries beforehand.

It is easy to see that we can apply path compression. And we can also use Union by rank, if we store the actual leader in an separate array.

```

struct Query {
    int L, R, idx;
};

vector<int> answer;
vector<vector<Query>> container;

    container[i] contains all queries with R == i.

stack<int> s;
for (int i = 0; i < n; i++) {
    while (!s.empty() && a[s.top()] > a[i]) {
        parent[s.top()] = i;
        s.pop();
    }
    s.push(i);
    for (Query q : container[i]) {
        answer[q.idx] = a[find_set(q.L)];
    }
}

```

Nowadays this algorithm is known as Arpa's trick. It is named after AmirReza Poorakhavan, who independently discovered and popularized this technique. Although this algorithm existed already before his discovery.

Offline LCA (lowest common ancestor in a tree) in $O(\alpha(n))$ on average

The algorithm for finding the LCA is discussed in the article [Lowest Common Ancestor - Tarjan's off-line algorithm](#). This algorithm compares favorable with

other algorithms for finding the LCA due to its simplicity (especially compared to an optimal algorithm like the one from [Farach-Colton and Bender](#)).

Storing the DSU explicitly in a set list / Applications of this idea when merging various data structures

One of the alternative ways of storing the DSU is the preservation of each set in the form of an **explicitly stored list of its elements**. At the same time each element also stores the reference to the representative of his set.

At first glance this looks like an inefficient data structure: by combining two sets we will have to add one list to the end of another and have to update the leadership in all elements of one of the lists.

However it turns out, the use of a **weighting heuristic** (similar to Union by size) can significantly reduce the asymptotic complexity: $O(m + n \log n)$ to perform m queries on the n elements.

Under weighting heuristic we mean, that we will always **add the smaller of the two sets to the bigger set**. Adding one set to another is easy to implement in `union_sets` and will take time proportional to the size of the added set. And the search for the leader in `find_set` will take $O(1)$ with this method of storing.

Let us prove the **time complexity** $O(m + n \log n)$ for the execution of m queries. We will fix an arbitrary element x and count how often it was touched in the merge operation `union_sets`. When the element x gets touched the first time, the size of the new set will be at least 2. When it gets touched the second time, the resulting set will have size of at least 4, because the smaller set gets added to the bigger one. And so on. This means, that x can only be moved in at most $\log n$ merge operations. Thus the sum over all vertices gives $O(n \log n)$ plus $O(1)$ for each request.

Here is an 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);
    }
}
}

```

This idea of adding the smaller part to a bigger part can also be used in a lot of solutions that have nothing to do with DSU.

For example consider the following **problem**: we are given a tree, each leaf has a number assigned (same number can appear multiple times on different leaves). We want to compute the number of different numbers in the subtree for every node of the tree.

Applying to this task the same idea it is possible to obtain this solution: we can implement a **DFS**, which will return a pointer to a set of integers - the list of numbers in that subtree. Then to get the answer for the current node (unless of course it is a leaf), we call DFS for all children of that node, and merge all the received sets together. The size of the resulting set will be the answer for the current node. To efficiently combine multiple sets we just apply the above-described recipe: we merge the sets by simply adding smaller ones to larger. In the end we get a $O(n \log^2 n)$ solution, because one number will only be added to a set at most $O(\log n)$ times.

Storing the DSU by maintaining a clear tree structure / Online bridge finding in $O(\alpha(n))$ on average

One of the most powerful applications of DSU is that it allows you to store both as **compressed and uncompressed trees**. The compressed form can be used for merging of trees and for the verification if two vertices are in the same tree, and the uncompressed form can be used - for example - to search for paths between two given vertices, or other traversals of the tree structure.

In the implementation this means that in addition to the compressed ancestor array `parent[]` we will need to keep the array of uncompressed ancestors `real_parent[]`. It is trivial that maintaining this additional array will not worsen the complexity: changes in it only occur when we merge two trees, and only in one element.

On the other hand when applied in practice, we often need to connect trees using a specified edge other than using the two root nodes. This means that we have no other choice but to re-root one of the trees (make the ends of the edge the new root of the tree).

At first glance it seems that this re-rooting is very costly and will greatly worsen the time complexity. Indeed, for rooting a tree at vertex v we must go from the vertex to the old root and change directions in `parent[]` and `real_parent[]` for all nodes on that path.

However in reality it isn't so bad, we can just re-root the smaller of the two trees similar to the ideas in the previous sections, and get $O(\log n)$ on average.

More details (including proof of the time complexity) can be found in the article [Finding Bridges Online](#).

2.1.3 Historical retrospective

The data structure DSU has been known for a long time.

This way of storing this structure in the form **of a forest of trees** was apparently first described by Galler and Fisher in 1964 (Galler, Fisher, "An Improved Equivalence Algorithm"), however the complete analysis of the time complexity was conducted much later.

The optimizations path compression and Union by rank has been developed by McIlroy and Morris, and independently of them also by Titter.

Hopcroft and Ullman showed in 1973 the time complexity $O(\log^* n)$ (Hopcroft, Ullman "Set-merging algorithms") - here \log^* is the **iterated logarithm** (this is a slow-growing function, but still not as slow as the inverse Ackermann function).

For the first time the evaluation of $O(\alpha(n))$ was shown in 1975 (Tarjan "Efficiency of a Good But Not Linear Set Union Algorithm"). Later in 1985 he, along with Leeuwen, published multiple complexity analyses for several different rank heuristics and ways of compressing the path (Tarjan, Leeuwen "Worst-case Analysis of Set Union Algorithms").

Finally in 1989 Fredman and Sachs proved that in the adopted model of computation **any** algorithm for the disjoint set union problem has to work in at least $O(\alpha(n))$ time on average (Fredman, Saks, "The cell probe complexity of dynamic data structures").

However it should also be noted, that there are several articles **disputing** this provisional valuation and asserting that the DSU with path compression and Union by rank runs in $O(1)$ time on average (Zhang "The Union-Find Problem Is Linear", Wu, Otoo "A Simpler Proof of the Average Case Complexity of Union-Find with Path Compression").

2.1.4 Problems

- [TIMUS - Anansi's Cobweb](#)
- [Codeforces - Roads not only in Berland](#)
- [TIMUS - Parity](#)
- [SPOJ - Strange Food Chain](#)

2.2 Fenwick Tree

Let, f be some *reversible* function and A be an array of integers of length N .

Fenwick tree is a data structure which:

- calculates the value of function f in the given range $[l; r]$ (i.e. $f(A_l, A_{l+1}, \dots, A_r)$) in $O(\lg n)$ time;
- updates the value of an element of A in $O(\lg n)$ time;
- requires $O(N)$ memory, or in other words, exactly the same memory required for A ;
- is easy to use and code, especially, in the case of multidimensional arrays.

Fenwick tree is also called Binary Indexed Tree.

The most common application of Fenwick tree is *calculating the sum of a range* (i.e. $f(A_1, A_2, \dots, A_k) = A_1 + A_2 + \dots + A_k$).

Fenwick tree was first described in a paper titled “A new data structure for cumulative frequency tables” (Peter M. Fenwick, 1994).

Description

For the sake of simplicity, we will assume that function f is just a *sum function*.

Given an array of integers $A[0 \dots N-1]$. Fenwick tree is an array $T[0 \dots N-1]$, where each of its elements is equal to the sum of elements of A in some range $[g(i); i]$:

$$T_i = \sum_{j=g(i)}^i A_j$$

where g is some function that satisfies $(g(i) \leq i)$ and we will define it a bit later.

Note: Fenwick tree presented here **does support** 0-based indexing (in case you were told that it does not support it).

Now we can write pseudo-code for the two operations mentioned above — get the sum of elements of A in range $[0; r]$ and update some element A_i :

```
def sum(int r):
    res = 0
    while (r >= 0):
        res += t[r]
        r = g(r) - 1
    return res

def inc(int i, int delta):
    for all j, where g(j) <= i <= j
        t[j] += delta
```

The function `sum` works as follows:

1. first, it adds the sum of the range $[g(r); r]$ (i.e. $T[r]$) to the **result**

2. then, it “jumps” to the range $[g(g(r) - 1); g(r) - 1]$, and adds this range’s sum to the **result**
3. and so on, until it “jumps” from $[0; g(g(\dots g(r) - 1 \dots - 1) - 1)]$ to $[g(-1); -1]$; that is where the **sum** function stops jumping.

The function **inc** works with the same analogy, but “jumps” in the direction of increasing indices:

1. sums of the ranges $[g(j); j]$ that satisfy the condition $g(j) \leq i \leq j$ are increased by **delta**, that is **t[j] += delta**.

It is obvious that complexity of both **sum** and **inc** do depend on the function g . We will define the function g in such a way that both of the operations have a logarithmic complexity $O(\lg N)$.

Definition of $g(i)$. Let us consider the least significant digit of i in binary. If this digit is 0, then let $g(i) = i$. Otherwise, the binary representation of i will end with at least one 1 bit. We will flip all these trailing 1’s (so they become 0’s) and define the result as a value of $g(i)$.

There exists a trivial solution for the non-trivial operation described above:

$g(i) = i \& (i+1)$

where **&** is a logical AND operator. It is not hard to convince yourself that this solution does the same thing as the operation described above.

Now, we need to find a way to find all j ’s, such that $g(j) \leq i \leq j$.

It is easy to see that we can find all such j ’s by starting with i and flipping the least significant zero bit. For example, for $i = 10$ we have:

```
j = 10, binary 0001010
j = 11, binary 0001011
j = 15, binary 0001111
j = 31, binary 0011111
j = 63, binary 0111111
...
```

Unsurprisingly, there also exists a simple way to do the above operation:

$h(j) = j | (j+1)$

where **|** is a logical OR operator.

Implementation: finding sum in one-dimensional array

```
struct FenwickTree {
    vector<int> bit; // binary indexed tree
    int n;

    void init(int n) {
        this->n = n;
    }
};
```

```

        bit.assign(n, 0);
    }
    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }
    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }
    void init(vector<int> a) {
        init(a.size());
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }
};

```

Implementation: finding minimum of $[0; r]$ in one-dimensional array

It is obvious that there is no way of finding minimum of range $[l; r]$ using Fenwick tree, as Fenwick tree can only answer queries of type $[0; r]$. Additionally, each time a value is update'd, new value should be smaller than the current value (because, the *min* function is not reversible). These, of course, are significant limitations.

```

struct FenwickTreeMin {
    vector<int> bit;
    int n;
    const int INF = (int)1e9;
    void init(int n) {
        this->n = n;
        bit.assign(n, INF);
    }
    int getmin(int r) {
        int ret = INF;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret = min(ret, bit[r]);
        return ret;
    }
    void update(int idx, int val) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] = min(bit[idx], val);
    }
    void init(vector<int> a) {
        init(a.size());
        for (size_t i = 0; i < a.size(); i++)
            update(i, a[i]);
    }
};

```

```

    }
};

```

Implementation: finding sum in two-dimensional array

As claimed before, it is easy to implement Fenwick Tree for multidimensional array.

```

struct FenwickTree2D {
    vector<vector<int>>> bit;
    int n, m;
    // init(...) { ... }
    int sum(int x, int y) {
        int ret = 0;
        for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
            for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
                ret += bit[i][j];
        return ret;
    }
    void add(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))
                bit[i][j] += delta;
    }
};

```

2.2.1 Practice Problems

- [UVA 12086 - Potentiometers](#)
- [LOJ 1112 - Curious Robin Hood](#)
- [LOJ 1266 - Points in Rectangle](#)
- [Codechef - SPREAD](#)
- [SPOJ - CTRICK](#)
- [SPOJ - MATSUM](#)
- [SPOJ - DQUERY](#)
- [SPOJ - NKTEAM](#)
- [SPOJ - YODANESS](#)
- [SRM 310 - FloatingMedian](#)
- [SPOJ - Ada and Beives](#)
- [Hackerearth - Counting in Byteland](#)
- [DevSkills - Shan and String](#)
- [Codeforces - Little Artem and Time Machine](#)
- [Codeforces - Hanoi Factory](#)
- [SPOJ - Tulip and Numbers](#)
- [SPOJ - SUMSUM](#)
- [SPOJ - Sabir and Gifts](#)
- [SPOJ - The Permutation Game Again](#)
- [SPOJ - Zig when you Zag](#)
- [SPOJ - Cryon](#)

- [SPOJ - Weird Points](#)
- [SPOJ - Its a Murder](#)
- [SPOJ - Bored of Suffixes and Prefixes](#)
- [SPOJ - Mega Inversions](#)
- [Codeforces - Subsequences](#)
- [Codeforces - Ball](#)
- [GYM - The Kamphaeng Phet's Chedis](#)
- [Codeforces - Garlands](#)
- [Codeforces - Inversions after Shuffle](#)
- [GYM - Cairo Market](#)
- [Codeforces - Goodbye Souvenir](#)
- [SPOJ - Ada and Species](#)
- [Codeforces - Thor](#)
- [Latin American Regionals 2017 - Fundraising](#)

Other sources

- [Fenwick tree on Wikipedia](#)
- [Binary indexed trees tutorial on TopCoder](#)

2.3 Sqrt Decomposition

Sqrt Decomposition is a method (or a data structure) that allows you to perform some common operations (finding sum of the elements of the sub-array, finding the minimal/maximal element, etc.) in $O(\sqrt{n})$ operations, which is much faster than $O(n)$ for the trivial algorithm.

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

2.3.1 Sqrt-decomposition based data structure

Given an array $a[0 \dots n-1]$, implement a data structure that allows to find the sum of the elements $a[l \dots r]$ for arbitrary l and r in $O(\sqrt{n})$ operations.

Description

The basic idea of sqrt decomposition is preprocessing. We'll divide the array a into blocks of length approximately \sqrt{n} , and for each block i we'll precalculate the sum of elements in it $b[i]$.

We can assume that both the size of the block and the number of blocks are equal to \sqrt{n} rounded up:

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

Then the array a is divided into blocks in the following way:

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

The last block may have fewer elements than the others (if n not a multiple of s), it is not important to the discussion (as it can be handled easily). Thus, for each block k , we know the sum of elements on it $b[k]$:

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

So, we have calculated the values of $b[k]$ (this required $O(n)$ operations). How can they help us to answer each query $[l; r]$? Notice that if the interval $[l; r]$ is long enough, it will contain several whole blocks, and for those blocks we can find the sum of elements in them in a single operation. As a result, the interval $[l; r]$ will contain parts of only two blocks, and we'll have to calculate the sum of elements in these parts trivially.

Thus, in order to calculate the sum of elements on the interval $[l; r]$ we only need to sum the elements of the 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 from $k+1$ to $p-1$:

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

Note: When $k = p$, i.e. l and r belong to the same block, the formula can't be applied, and the sum should be calculated trivially.

This approach allows us to significantly reduce the number of operations. Indeed, the size of each “tail” does not exceed the block length s , and the number of blocks in the sum does not exceed s . Since we have chosen $s \approx \sqrt{n}$, the total number of operations required to find the sum of elements on the interval $[l; r]$ is $O(\sqrt{n})$.

Implementation

Let's start with the simplest implementation:

```
// input data
int n;
vector<int> a (n);

// preprocessing
int len = (int) sqrt (n + .0) + 1; // size of the block and the number of blocks
vector<int> b (len);
for (int i=0; i<n; ++i)
    b[i / len] += a[i];

// answering the queries
for (;;) {
    int l, r;
    // read input data for the next query
    int sum = 0;
    for (int i=l; i<=r; )
        if (i % len == 0 && i + len - 1 <= r) {
            // if the whole block starting at i belongs to [l; r]
            sum += b[i / len];
            i += len;
        }
        else {
            sum += a[i];
            ++i;
        }
}
```

This implementation has unreasonably many division operations (which are much slower than other arithmetical operations). Instead, we can calculate the indices of the blocks c_l and c_r which contain indices l and r , and loop through blocks $c_l + 1 \dots c_r - 1$ with separate processing of the “tails” in blocks c_l and c_r . This approach corresponds to the last formula in the description, and makes the case $c_l = c_r$ a special case.

```

int sum = 0;
int c_l = l / len,    c_r = r / len;
if (c_l == c_r)
    for (int i=l; i<=r; ++i)
        sum += a[i];
else {
    for (int i=l, end=(c_l+1)*len-1; i<=end; ++i)
        sum += a[i];
    for (int i=c_l+1; i<=c_r-1; ++i)
        sum += b[i];
    for (int i=c_r*len; i<=r; ++i)
        sum += a[i];
}

```

2.3.2 Other problems

So far we were discussing the problem of finding the sum of elements of a continuous subarray. This problem can be extended to allow to **update individual array elements**. If an element $a[i]$ changes, it's sufficient to update the value of $b[k]$ for the block to which this element belongs ($k = i/s$) in one operation:

$$b[k] += a_{new}[i] - a_{old}[i]$$

On the other hand, the task of finding the sum of elements can be replaced with the task of finding minimal/maximal element of a subarray. If this problem has to address individual elements' updates as well, updating the value of $b[k]$ is also possible, but it will require iterating through all values of block k in $O(s) = O(\sqrt{n})$ operations.

Sqrt decomposition can be applied in a similar way to a whole class of other problems: finding the number of zero elements, finding the first non-zero element, counting elements which satisfy a certain property etc.

Another class of problems appears when we need to **update array elements on intervals**: increment existing elements or replace them with a given value.

For example, let's say we can do two types of operations on an array: add a given value δ to all array elements on interval $[l; r]$ or query the value of element $a[i]$. Let's store the value which has to be added to all elements of block k in $b[k]$ (initially all $b[k] = 0$). During each "add" operation we need to add δ to $b[k]$ for all blocks which belong to interval $[l; r]$ and to add δ to $a[i]$ for all elements which belong to the "tails" of the interval. The answer a query i is simply $a[i] + b[i/s]$. This way "add" operation has $O(\sqrt{n})$ complexity, and answering a query has $O(1)$ complexity.

Finally, those two classes of problems can be combined if the task requires doing **both** element updates on an interval and queries on an interval. Both operations can be done with $O(\sqrt{n})$ complexity. This will require two block arrays b and c : one to keep track of element updates and another to keep track of answers to the query.

There exist other problems which can be solved using sqrt decomposition, for example, a problem about maintaining a set of numbers which would allow

adding/deleting numbers, checking whether a number belongs to the set and finding k -th largest number. To solve it one has to store numbers in increasing order, split into several blocks with \sqrt{n} numbers in each. Every time a number is added/deleted, the blocks have to be rebalanced by moving numbers between beginnings and ends of adjacent blocks.

2.3.3 Mo's algorithm

A similar idea, based on sqrt decomposition, can be used to answer range queries (Q) offline in $O((N+Q)\sqrt{N})$. This might sound like a lot worse than the methods in the previous section, since this is a slightly worse complexity than we had earlier and cannot update values between two queries. But in a lot of situations this method has advantages. During a normal sqrt decomposition, we have to precompute the answers for each block, and merge them during answering queries. In some problems this merging step can be quite problematic. E.g. when each queries asks to find the **mode** of its range (the number that appears the most often). For this each block would have to store the count of each number in it in some sort of data structure, and we cannot longer perform the merge step fast enough any more. **Mo's algorithm** uses a completely different approach, that can answer these kind of queries fast, because it only keeps track of one data structure, and the only operations with it are easy and fast.

The idea is to answer the queries in a special order based on the indices. We will first answer all queries which have the left index in block 0, then answer all queries which have left index in block 1 and so on. And also we will have to answer the queries of a block in a special order, namely sorted by the right index of the queries.

As already said we will use a single data structure. This data structure will store information about the range. At the beginning this range will be empty. When we want to answer the next query (in the special order), we simply extend or reduce the range, by adding/removing elements on both sides of the current range, until we transformed it into the query range. This way, we only need to add or remove a single element once at a time, which should be pretty easy operations in our data structure.

Since we change the order of answering the queries, this is only possible when we are allowed to answer the queries in offline mode.

Implementation

In Mo's algorithm we use two functions for adding an index and for removing an index from the range which we are currently maintaining.

```
void remove(idx); // TODO: remove value at idx from data structure
void add(idx);    // TODO: add value at idx from data structure
int get_answer(); // TODO: extract the current answer of the data structure

int block_size;

struct Query {
```



```

    int l, r, idx;
    bool operator<(Query other) const
    {
        return make_pair(l / block_size, r) <
               make_pair(other.l / block_size, other.r);
    }
};

vector<int> mo_s_algorithm(vector<Query> queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());

    // TODO: initialize data structure

    int cur_l = 0;
    int cur_r = -1;
    // invariant: data structure will always reflect the range [cur_l, cur_r]
    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
            add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            add(cur_r);
        }
        while (cur_l < q.l) {
            remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}

```

Based on the problem we can use a different data structure and modify the `add/remove/get_answer` functions accordingly. For example if we are asked to find range sum queries then we use a simple integer as data structure, which is 0 at the beginning. The `add` function will simply add the value of the position and subsequently update the answer variable. On the other hand `remove` function will subtract the value at position and subsequently update the answer variable. And `get_answer` just returns the integer.

For answering mode-queries, we can use a binary search tree (e.g. `map<int, int>`) for storing how often each number appears in the current range, and a second binary search tree (e.g. `set<pair<int, int>>`) for keeping counts of the numbers (e.g. as count-number pairs) in order. The `add` method removes the current number from the second BST, increases the count in the first one, and

inserts the number back into the second one. `remove` does the same thing, it only decreases the count. And `get_answer` just looks at second tree and returns the best value in $O(1)$.

Complexity

Sorting all queries will take $O(Q \log Q)$.

How about the other operations? How many times will the `add` and `remove` be called?

Let's say the block size is S .

If we look only look at all queries that with the left index in the same block. The queries are sorted by the right index. Therefore we will call `add(cur_r)` and `remove(cur_r)` only $O(N)$ times for all these queries combined. This gives $O(\frac{N}{S}N)$ calls for all blocks.

The value of `cur_l` can change by at most $O(S)$ during between two queries. Therefore we have an additional $O(SQ)$ calls of `add(cur_l)` and `remove(cur_l)`.

For $S \approx \sqrt{N}$ this gives $O((N + Q)\sqrt{N})$ operations in total. Thus the complexity is $O((N + Q)F\sqrt{N})$ where $O(F)$ is the complexity of `add` and `remove` function.

Tips for improving runtime

- Block size of precisely \sqrt{N} doesn't always offer the best runtime. For example, if $\sqrt{N} = 750$ then it may happen that block size of 700 or 800 may run better.
- In odd blocks sort the right index in ascending order and in even blocks sort it in descending order. This will minimize the movement of right pointer, as the normal sorting will move the right pointer from the end back to the beginning at the start of every block. With the improved version this resetting is no more necessary.

```
bool cmp(pair<int, int> p, pair<int, int> q) {
    if (p.first / BLOCK_SIZE != q.first / BLOCK_SIZE)
        return p < q;
    return (p.first / BLOCK_SIZE & 1) ? (p.second < q.second) : (p.second > q.second);
}
```

You can read about even faster sorting approach [here](#).

2.3.4 Practice Problems

- [UVA - 12003 - Array Transformer](#)
- [UVA - 11990 Dynamic Inversion](#)
- [SPOJ - Give Away](#)
- [Codeforces - Till I Collapse](#)
- [Codeforces - Destiny](#)
- [Codeforces - Holes](#)
- [Codeforces - XOR and Favorite Number](#)

- [Codeforces - Powerful array](#)
- [SPOJ - DQUERY](#)

2.4 Segment Tree

A Segment Tree is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array. This includes finding the sum of consecutive array elements $a[l \dots r]$, or finding the minimum element in a such a range in $O(\log n)$ time. Between answering such queries the Segment Tree allows modifying the array by replacing one element, or even change the elements of a whole subsegment (e.g. assigning all elements $a[l \dots r]$ to any value, or adding a value to all element in the subsegment).

In general a Segment Tree is a very flexible data structure, and a huge number of problems can be solved with it. Additionally it is also possible to apply more complex operations and answer more complex queries (see Advanced versions of Segment Trees). In particular the Segment Tree can be easily generalized to larger dimensions. For instance with a two-dimensional Segment Tree you can answer sum or minimum queries over some subrectangle of a given matrix. However only in $O(\log^2 n)$ time.

One important property of Segment Trees is, that they require only a linear amount of memory. The standard Segment Tree requires $4n$ vertices for working on an array of size n .

2.4.1 Simplest form of a Segment Tree

To start easy, we consider the simplest form of a Segment Tree. We want to answer sum queries efficiently. The formal definition of our task is: We have an array $a[0 \dots n - 1]$, and the Segment Tree must be able to find the sum of elements between the indices l and r (i.e. computing the sum $\sum_{i=l}^r a[i]$), and also handle changing values of the elements in the array (i.e. perform assignments of the form $a[i] = x$). The Segment Tree should be able to process both queries in $O(\log n)$ time.

Structure of the Segment Tree

So, what is a Segment Tree?

We compute and store the sum of the elements of the whole array, i.e. the sum of the segment $a[0 \dots n - 1]$. We then split the array into two halves $a[0 \dots n/2]$ and $a[n/2 + 1 \dots n - 1]$ and compute the sum of each half and store them. Each of these two halves in turn also split in half, their sums are computed and stored. And this process repeats until all segments reach size 1. In other words we start with the segment $a[0 \dots n - 1]$, split the current segment in half (if it has not yet become a segment containing a single element), and then calling the same procedure for both halves. For each such segment we store the sum of the numbers on it.

We can say, that these segments form a binary tree: the root of this tree is the segment $a[0 \dots n - 1]$, and each vertex (except leaf vertices) has exactly two child vertices. This is why the data structure is called “Segment Tree”, even though in most implementations the tree is not constructed explicitly (see [Implementation](#)).

Here is a visual representation of such a Segment Tree over the array $a = [1, 3, -2, 8, -7]$:

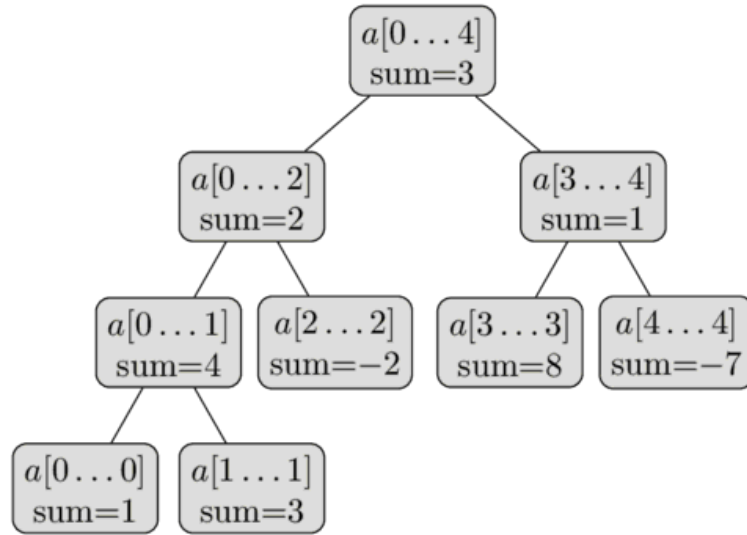


Figure 2.3: “Sum Segment Tree”

From this short description of the data structure, we can already conclude that a Segment Tree only requires a linear number of vertices. The first level of the tree contains a single node (the root), the second level will contain two vertices, in the third it will contain four vertices, until the number of vertices reaches n . Thus the number of vertices in the worst case can be estimated by the sum $1 + 2 + 4 + \dots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} < 4n$.

It is worth noting that whenever n is not a power of two, not all levels of the Segment Tree will be completely filled. We can see that behavior in the image. For now we can forget about this fact, but it will become important later during the implementation.

The height of the Segment Tree is $O(\log n)$, because when going down from the root to the leaves the size of the segments decreases approximately by half.

Construction

A Segment Tree can be constructed efficiently as follows: We start at the bottom level, the leaf vertices. A vertex is a leaf vertex, if its corresponding segment covers only one value. Therefore we can simply copy the values of the elements $a[i]$. On the basis of these values, we can compute the sums of the previous level. And on the basis of those, we can compute the sums of the previous, and repeat the procedure until we reach the root vertex. It is convenient to describe this operation recursively: we start the construction at the root vertex and the construction procedure, if called on a not-leaf vertex, first recursively constructs

the two child vertices, and then sums up the computed sums of these children. If it is called on a leaf vertex, it simply uses the value of the array.

The time complexity of the construction is $O(n)$.

Sum queries

For now we are going to answer sum queries. As an input we receive two integers l and r , and we have to compute the sum of the segment $a[l \dots r]$ in $O(\log n)$ time.

To do this, we will traverse the Segment Tree and use the precomputed sums of the segments. Lets assume that we are currently at the vertex that covers the segment $a[tl \dots tr]$. There are three possible cases.

The easiest case is when the segment $a[l \dots r]$ is equal to the corresponding segment of the current vertex (i.e. $a[l \dots r] = a[tl \dots tr]$), then we are finished and can return the precomputed sum that is stored in the vertex.

Alternatively the segment of the query can falls completely into the domain of either the left or the right child. Recall that the left child covers the segment $a[tl \dots tm]$ and the right vertex covers the segment $a[tm + 1 \dots tr]$ with $tm = (tl + tr)/2$. In this case we can simply go to the child vertex, which corresponding segment covers the query segment, and execute the algorithm described here with that vertex.

And then there is the last case, the query segment intersects with both children. In this case we have no other option as to make two recursive calls, one for each child. First we go to the left child, compute a partial answer for this vertex (i.e. the sum of values of the intersection between the segment of the query and the segment of the left child), then go to the right child, compute the partial answer using that vertex, and then combine the answers by adding them. In other words, since the left child represents the segment $a[tl \dots tm]$ and the right child the segment $a[tm + 1 \dots tr]$, we compute the sum query $a[l \dots tm]$ using the left child, and the sum query $a[tm + 1 \dots r]$ using the right child.

So processing a sum query is a function that recursively calls itself once with either the left or the right child (without changing the query boundaries), or twice, once for the left and once for the right child (by splitting the query into two subqueries). And the recursion ends, whenever the boundaries of the current query segment coincides with the boundaries of the segment of the current vertex. In that case the answer will be the precomputed value of the sum of this segment, which is stored in the tree.

In other words, the calculation of the query is a traversal of the tree, which spreads through all necessary branches of the tree, and uses the precomputed sum values of the segments in the tree.

Obviously we will start the traversal from the root vertex of the Segment Tree.

The procedure is illustrated in the following image. Again the array $a = [1, 3, -2, 8, -7]$ is used, and here we want to compute the sum $\sum_{i=2}^4 a[i]$. The colored vertices will be visited, and we will use the precomputed values of the green vertices. This gives us the result $-2 + 1 = -1$.

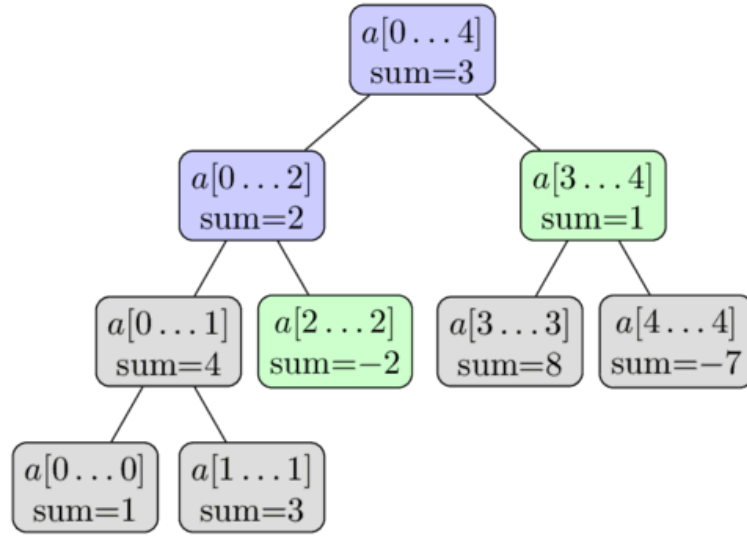


Figure 2.4: “Sum Segment Tree Query”

Why is the complexity of this algorithm $O(\log n)$? To show this complexity we look at each level of the tree. It turns out, that for each level we only visit not more than four vertices. And since the height of the tree is $O(\log n)$, we receive the desired running time.

We can show that this proposition (at most four vertices each level) is true by induction. At the first level, we only visit one vertex, the root vertex, so here we visit less than four vertices. Now let's look at an arbitrary level. By induction hypothesis, we visit at most four vertices. If we only visit at most two vertices, the next level has at most four vertices. That's trivial, because each vertex can only cause at most two recursive calls. So let's assume that we visit three or four vertices in the current level. From those vertices, we will analyze the vertices in the middle more carefully. Since the sum query asks for the sum of a continuous subarray, we know that segments corresponding to the visited vertices in the middle will be completely covered by the segment of the sum query. Therefore these vertices will not make any recursive calls. So only the most left, and the most right vertex will have the potential to make recursive calls. And those will only create at most four recursive calls, so also the next level will satisfy the assertion. We can say that one branch approaches the left boundary of the query, and the second branch approaches the right one.

Therefore we visit at most $4 \log n$ vertices in total, and that is equal to a running time of $O(\log n)$.

In conclusion the query works by dividing the input segment into several sub-segments for which all the sums are already precomputed and stored in the tree. And if we stop partitioning whenever the query segment coincides with the vertex segment, then we only need $O(\log n)$ such segments, which gives the effectiveness of the Segment Tree.

Update queries

Now we want to modify a specific element in the array, let's say we want to do the assignment $a[i] = x$. And we have to rebuild the Segment Tree, such that it correspond to the new, modified array.

This query is easier than the sum query. Each level of a Segment Tree forms a partition of the array. Therefore an element $a[i]$ only contributes to one segment from each level. Thus only $O(\log n)$ vertices need to be updated.

It is easy to see, that the update request can be implemented using a recursive function. The function gets passed the current tree vertex, and it recursively calls itself with one of the two child vertices (the one that contains $a[i]$ in its segment), and after that recomputes its sum value, similar how it is done in the build method (that is as the sum of its two children).

Again here is a visualization using the same array. Here we perform the update $a[2] = 3$. The green vertices are the vertices that we visit and update.

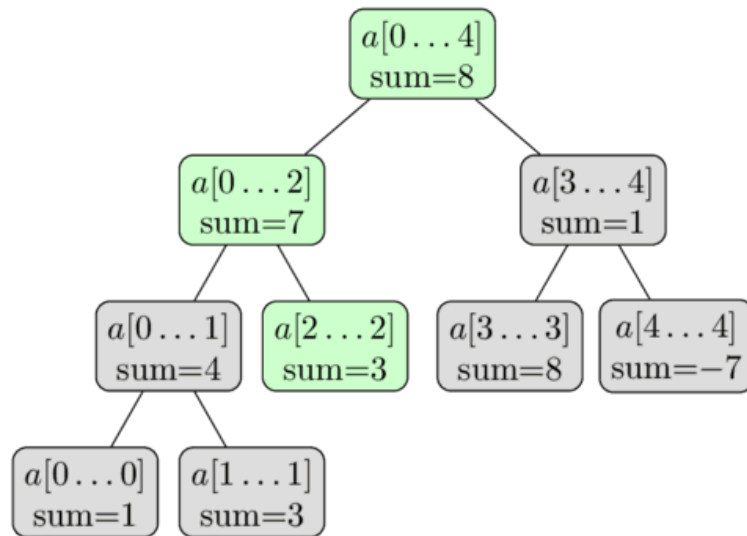


Figure 2.5: “Sum Segment Tree Update”

Implementation

The main consideration is how to store the Segment Tree. Of course we can define a Vertex struct and create objects, that store the boundaries of the segment, its sum and additionally also pointers to its child vertices. However this requires storing a lot of redundant information. We will use a simple trick, to make this a lot more efficient. We only store the sums in an array. The sum of the root vertex at index 1, the sums of its two child vertices at indices 2 and 3, the sums of the children of those two vertices at indices 4 to 7, and so on. It is easy to see, that the left child of a vertex at index i is stored at index $2i$, and the right one at index $2i + 1$.

This simplifies the implementation a lot. We don't need to store the structure of the tree in memory. It is defined implicitly. We only need one array which contains the sums of all segments.

As noted before, we need to store at most $4n$ vertices. It might be less, but for convenience we always allocate an array of size $4n$. There will be some elements in the sum array, that will not correspond to any vertices in the actual tree, but this doesn't complicate the implementation.

So, we store the Segment Tree simply as an array $t[]$ with a size of four times the input size n :

```
int n, t[4*MAXN];
```

The procedure for constructing the Segment Tree from a given array $a[]$ looks like this: it is a recursive function with the parameters $a[]$ (the input array), v (the index of the current vertex), and the boundaries tl and tr of the current segment. In the main program this function will be called with the parameters of the root vertex: $v = 1$, $tl = 0$, and $tr = n - 1$.

```
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

Further the function for answering sum queries is also a recursive function, which receives as parameters information about the current vertex/segment (i.e. the index v and the boundaries tl and tr) and also the information about the boundaries of the query, l and r . In order to simplify the code, this function always does two recursive calls, even if only one is necessary - in that case the superfluous recursive call will have $l > r$, and this can easily be caught using an 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 the update query. The function will also receive information about the current vertex/segment, and additionally also the parameter of the update query (i.e. the position of the element and 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];
    }
}

```

2.4.2 Advanced versions of Segment Trees

A Segment Tree is a very flexible data structure, and allows variations and extensions in many different directions. Let's try to categorize them below.

More complex queries

It can be quite easy to change the Segment Tree in a direction, such that it computes different queries (e.g. computing the minimum / maximum instead of the sum), but it also can be very nontrivial.

Finding the maximum Let us slightly change the condition of the problem described above: instead of querying the sum, we will now make maximum queries.

The tree will have exactly the same structure as the tree described above. We only need to change the way $t[v]$ is computed in the build and update functions. $t[v]$ will now store the maximum of the corresponding segment. And we also need to change the calculation of the returned value of the sum function (replacing the summation by the maximum).

Of course this problem can be easily changed into computing the minimum instead of the maximum.

Instead of showing an implementation to this problem, the implementation will be given to a more complex version of this problem in the next section.

Finding the maximum and the number of times it appears This task is very similar to the previous one. In addition of finding the maximum, we also have to find the number of occurrences of the maximum.

To solve this problem, we store a pair of numbers at each vertex in the tree: in addition to the maximum we also store the number of occurrences of it in the corresponding segment. Determining the correct pair to store at $t[v]$ can still be done in constant time using the information of the pairs stored at the child vertices. Combining two such pairs should be done in a separate function, since this will be a operation that we will do while building the tree, while answering maximum queries and while performing modifications.

```

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]);
    }
}

```

Compute the greatest common divisor / least common multiple In this problem we want to compute the GCD / LCM of all numbers of given ranges of the array.

This interesting variation of the Segment Tree can be solved in exactly the same way as the Segment Trees we derived for sum / minimum / maximum queries: it is enough to store the GCD / LCM of the corresponding vertex in each vertex of the tree. Combining two vertices can be done by computing the

GCM / LCM of both vertices.

Counting the number of zeros, searching for the k -th zero In this problem we want to find the number of zeros in a given range, and additionally find the index of the k -th zero using a second function.

Again we have to change the store values of the tree a bit: This time we will store the number of zeros in each segment in $t[]$. It is pretty clear, how to implement the build, update and count_zero functions, we can simple use the ideas from the sum query problem. Thus we solved the first part of the problem.

Now we learn how to solve the of problem of finding the k -th zero in the array $a[]$. To do this task, we will descend the Segment Tree, starting at the root vertex, and moving each time to either the left or the right child, depending on which segment contains the k -th zero. In order to decide to which child we need to go, it is enough to look at the number of zeros appearing in the segment corresponding to the left vertex. If this precomputed count is greater or equal to k , it is necessary to descend to the left child, and otherwise descent to the right child. Notice, if we chose the right child, we have to subtract the number of zeros of the left child from k .

In the implementation we can handle the special case, $a[]$ containing less than k zeros, by returning -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]);
}
```

Searching for an array prefix with a given amount The task is as follows: for a given value x we have to quickly find smallest index i such that the sum of the first i elements of the array $a[]$ is greater or equal to x (assuming that the array $a[]$ only contains non-negative values).

This task can be solved using binary search, computing the sum of the prefixes with the Segment Tree. However this will lead to a $O(\log^2 n)$ solution.

Instead we can use the same idea as in the previous section, and find the position by descending the tree: by moving each time to the left or the right, depending on the sum of the left child. Thus finding the answer in $O(\log n)$ time.

Finding subsegments with the maximal sum Here again we receive a range $a[l \dots r]$ for each query, this time we have to find a subsegment $a[l' \dots r']$ such that $l \leq l'$ and $r' \leq r$ and the sum of the elements of this segment is maximal. As before we also want to be able to modify individual elements of the array.

The elements of the array can be negative, and the optimal subsegment can be empty (e.g. if all elements are negative).

This problem is a non-trivial usage of a Segment Tree. This time we will store four values for each vertex: the sum of the segment, the maximum prefix sum, the maximum suffix sum, and the sum of the maximal subsegment in it. In other words for each segment of the Segment Tree the answer is already precomputed as well as the answers for segments touching the left and the right boundaries of the segment.

How to build a tree with such data? Again we compute it in a recursive fashion: we first compute all four values for the left and the right child, and then combine those to archive the four values for the current vertex. Note the answer for the current vertex is either:

- the answer of the left child, which means that the optimal subsegment is entirely placed in the segment of the left child
- the answer of the right child, which means that the optimal subsegment is entirely placed in the segment of the right child
- the sum of the maximum suffix sum of the left child and the maximum prefix sum of the right child, which means that the optimal subsegment intersects with both children.

Hence the answer to the current vertex is the maximum of these three values. Computing the maximum prefix / suffix sum is even easier. Here is the implementation of the combine function, which receives only data from the left and right child, and returns the data of the current vertex.

```
struct data {
    int sum, pref, suff, ans;
};

data combine(data l, data r) {
    data res;
    res.sum = l.sum + r.sum;
    res.pref = max(l.pref, l.sum + r.pref);
    res.suff = max(r.suff, r.sum + l.suff);
    res.ans = max(max(l.ans, r.ans), l.suff + r.pref);
    return res;
}
```

Using the combine function it is easy to build the Segment Tree. We can implement it in exactly the same way as in the previous implementations. To initialize the leaf vertices, we additionally create the auxiliary function `make_data`, which will return a data object holding the information of a single value.

```
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 only remains, how to compute the answer to a query. To answer it, we go down the tree as before, breaking the query into several subsegments that coincide with the segments of the Segment Tree, and combine the answers in them into a single answer for the query. Then it should be clear, that the work is exactly the same as in the simple Segment Tree, but instead of summing / minimizing / maximizing the values, we use the combine function.

```

data query(int v, int tl, int tr, int l, int r) {
    if (l > r)
        return make_data(0);
    if (l == tl && r == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    return combine(query(v*2, tl, tm, l, min(r, tm)),
                  query(v*2+1, tm+1, tr, max(l, tm+1), r));
}

```

Saving the entire subarrays in each vertex

This is a separate subsection that stands apart from the others, because at each vertex of the Segment Tree we don't store information about the corresponding segment in compressed form (sum, minimum, maximum, ...), but store all element of the segment. Thus the root of the Segment Tree will store all elements of the array, the left child vertex will store the first half of the array, the right vertex the second half, and so on.

In its simplest application of this technique we store the elements in sorted order. In more complex versions the elements are not stored in lists, but more advanced data structures (sets, maps, ...). But all these methods have the common factor, that each vertex requires linear memory (i.e. proportional to the length of the corresponding segment).

The first natural question, when considering these Segment Trees, is about memory consumption. Intuitively this might look like $O(n^2)$ memory, but it turns out that the complete tree will only need $O(n \log n)$ memory. Why is this so? Quite simply, because each element of the array falls into $O(\log n)$ segments (remember the height of the tree is $O(\log n)$).

So in spite of the apparent extravagance of such a Segment Tree, it consumes only slightly more memory than the usual Segment Tree.

Several typical applications of this data structure are described below. It is worth noting the similarity of these Segment Trees with 2D data structures (in fact this is a 2D data structure, but with rather limited capabilities).

Find the smallest number greater or equal to a specified number. No modification queries. We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x .

We construct a Segment Tree. In each vertex we store a sorted list of all numbers occurring in the corresponding segment, like described above. How to build such a Segment Tree as effectively as possible? As always we approach this problem recursively: let the lists of the left and right children already be constructed, and we want to build the list for the current vertex. From this view the operation is known trivial and can be accomplished in linear time: We only need to combine the two sorted lists into one, which can be done by iteration over them using two pointers. The C++ STL already has an implementation of this algorithm.

Because this structure of the Segment Tree and the similarities to the merge sort algorithm, the data structure is also often called “Merge Sort Tree”.

```
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 Segment Tree constructed in this way will require $O(n \log n)$ memory. And thanks to this implementation its construction also takes

$O(n \log n)$ time, after all each list is constructed in linear time in respect to its size.

Now consider the answer to the query. We will go down the tree, like in the regular Segment Tree, and breaking our segment $a[l \dots r]$ into several subsegments (into at most $O(\log n)$ pieces). It is clear that the answer of the whole answer is the minimum of each of the subqueries. So now we only need to understand, how to how to response to a query on one such subsegment that corresponds with some vertex of the tree.

We are at some vertex of the Segment Tree and we want to compute the answer to the query, i.e. find the minimum number greater than or equal to a given number x . Since the vertex contains the list of elements in sorted order, we can simply perform a binary search on this list and return the first number, greater than or equal to x .

Thus the answer to the query in one segment of the tree takes $O(\log n)$ time, and the entire query is processed in $O(\log^2 n)$.

```
int query(int v, int tl, int tr, int l, int r, int x) {
    if (l > r)
        return INF;
    if (l == tl && r == tr) {
        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));
}
```

The constant INF is equal to some large number that is bigger than all numbers in the array. Its usage means, that there is no number greater than or equal to x in the segment. It has the meaning of “there is no answer in the given interval”.

Find the smallest number greater or equal to a specified number. With modification queries. This task is similar to the previous. The last approach has a disadvantage, it was not possible to modify the array between answering queries. Now we want to do exactly this: a modification query will do the assignment $a[i] = y$.

The solution is similar to the solution of the previous problem, but instead of lists at each vertex of the Segment Tree, we will store a balanced list that allows you to quickly search for numbers, delete numbers, and insert new numbers. Since the array can contain a number repeated, the optimal choice is the data structure multiset.

The construction of such a Segment Tree is done in pretty much done in the same way as in the previous problem, only now we need to combine multisets and not sorted lists. This leads to a construction time of $O(n \log^2 n)$ (in general

merging two red-black trees can be done in linear time, but the C++ STL doesn't guarantee this time complexity).

The query function is also almost equivalent, only now the `lower_bound` function of the multiset function should be called instead (`std::lower_bound` only works in $O(\log n)$ time if used with random-access iterators).

Finally the modification request. To process it, we must do down the tree, and modify all multiset from the corresponding segments that contain the effected element. We simply delete the old value of this element (but only one occurrence), 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 of this modification query also takes $O(\log^2 n)$ time.

Find the smallest number greater or equal to a specified number. Acceleration with “fractional cascading”. We have the same problem statement, we want to find the minimal number greater than or equal to x in a segment, but this time in $O(\log n)$ time. We will improve the time complexity using the technique “fractional cascading”.

Fractional cascading is a simple technique that allows you to improve the running time of multiple binary searches, which are conducted at the same time. Our previous approach to the search query was, that we divide the task into several subtasks, each of which is solved with a binary search. Fractional cascading allows you to replace all of these binary searches with a single one.

The simplest and most obvious example of fractional cascading is the following problem: there are k sorted lists of numbers, and we must find in each list the first number greater than or equal to the given number.

Instead of performing a binary search for each list, we could merge all lists into one big sorted list. Additionally for each element y we store a list of results of searching for y in each of the k lists. Therefore if we want to find the smallest number greater than or equal to x , we just need to perform one single binary search, and from the list of indices we can determine the smallest number in each list. This approach however requires $O(n \cdot k)$ (n is the length of the combined lists), which can be quite inefficient.

Fractional cascading reduces this memory complexity to $O(n)$ memory, by creating from the k input lists k new lists, in which each list contains the

corresponding list and additionally also every second element of the following new list. Using this structure it is only necessary to store two indices, the index of the element in the original list, and the index of the element in the following new list. So this approach only uses $O(n)$ memory, and still can answer the queries using a single binary search.

But for our application we do not need the full power of fractional cascading. In our Segment Tree a vertex contains the list of all elements, that occur in either the left or the right child vertex. Therefore to avoid binary search in the lists of the children it is enough to store the position of each element in the left and the right child list.

Thus instead of storing the usual list of all numbers, we store three values for each element: the value of the element itself, the position of it in the list of the left child, and the position of it in the list of the right child. This allows us to determine the position in the list of the left and the right child in $O(1)$, instead of finding it with binary search.

It is simple to apply this technique to a problem, that doesn't require modification queries. In this problem the two positions are just numbers and can easily be computed by counting when merging two sorted sequences.

This type of Segment Tree is nowadays known as "Wavelet Tree". And a few more advanced usages have been explored (e.g. support for modification queries of the form "swap neighboring elements").

Other possible variations This technique implies a whole new class of possible applications. Instead of storing a vector or a multiset in each vertex, other data structures can be used: other Segment Trees (somewhat discussed in Generalization to higher dimensions), Fenwick Trees, Cartesian trees, etc.

Range updates (Lazy Propagation)

All problems in the above sections discussed modification queries that only effected a single element of the array each. However the Segment Tree allows applying modification queries to an entire segment of contiguous elements, and perform the query in the same time $O(\log n)$.

Addition on segments We begin by considering problems of the simplest form: the modification query should add a number x to all numbers in the segment $a[l \dots r]$. The second query, that we are supposed to answer, asked simply for the value of $a[i]$.

To make the addition query efficient, we store at each vertex in the Segment Tree how many we should add to all numbers in the corresponding segment. For example, if the query "add 3 to the whole array $a[0 \dots n - 1]$ " comes, then we place the number 3 in the root of the tree. In general we have to place this number multiple to multiple segments, which form a partition of the query segment. Thus we don't have to change all $O(n)$ values, but only $O(\log n)$ many.

If now there comes a query that asks the current value of a particular array entry, it is enough to go down the tree and add up all values found along the way.

```

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] = 0;
    }
}

void update(int v, int tl, int tr, int l, int r, int add) {
    if (l > r)
        return;
    if (l == tl && r == tr) {
        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);
}

```

Assignment on segments Suppose now that the modification query asks to assign each element of a certain segment $a[l \dots r]$ to some value p . As a second query we will again consider reading the value of the array $a[i]$.

To perform this modification query on a whole segment, you have to store at each vertex of the Segment Tree whether the corresponding segment is covered entirely with the same value or not. This allows us to make a “lazy” update: instead of changing all segments in the tree that cover the query segment, we only change some, and leave others unchanged. A marked vertex will mean, that every element of the corresponding segment is assigned to that value, and actually also the complete subtree should only contain this value. In a sense we are lazy and delay writing the new value to all those vertices. We can do this tedious task later, if this is necessary.

So after the modification query is executed, some parts of the tree become irrelevant - some modifications remain unfulfilled in it.

For example if a modification query “assign a number to the whole array $a[0 \dots n - 1]$ ” gets executed, in the Segment Tree only a single change is made -

the number is placed in the root of the tree and this vertex gets marked. The remaining segments remain unchanged, although in fact the number should be placed in the whole tree.

Suppose now that the second modification query says, that the first half of the array $a[0 \dots n/2]$ should be assigned with some other number. To process this query we must assign each element in the whole left child of the root vertex with that number. But before we do this, we must first sort out the root vertex first. The subtlety here is that the right half of the array should still be assigned to the value of the first query, and at the moment there is no information for the right half stored.

The way to solve this is to push the information of the root to its children, i.e. if the root of the tree was assigned with any number, then we assign the left and the right child vertices with this number and remove the mark of the root. After that, we can assign the left child with the new value, without losing any necessary information.

Summarizing we get: for any queries (a modification or reading query) during the descent along the tree we should always push information from the current vertex into both of its children. We can understand this in such a way, that when we descend the tree we apply delayed modifications, but exactly as much as necessary (so not to degrade the complexity of $O(\log n)$).

For the implementation we need to make a push function, which will receive the current vertex, and it will push the information for its vertex to both its children. We will call this function at the beginning of the query functions (but we will not call it from the leaves, because there is no need to push information from them any further).

```
void push(int v) {
    if (marked[v]) {
        t[v*2] = t[v*2+1] = t[v];
        marked[v*2] = marked[v*2+1] = true;
        marked[v] = false;
    }
}

void update(int v, int tl, int tr, int l, int r, int new_val) {
    if (l > r)
        return;
    if (l == tl && tr == r) {
        t[v] = new_val;
        marked[v] = true;
    } else {
        push(v);
        int tm = (tl + tr) / 2;
        update(v*2, tl, tm, l, min(r, tm), new_val);
        update(v*2+1, tm+1, tr, max(l, tm+1), r, new_val);
    }
}

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);
}

```

Notice: the function `get` can also be implemented in a different way: do not make delayed updates, but immediately return the value $t[v]$ if $marked[v]$ is true.

Adding on segments, querying for maximum Now the modification query is to add a number to all elements in a range, and the reading query is to find the maximum in a range.

So for each vertex of the Segment Tree we have to store the maximum of the corresponding subsegment. The interesting part is how to recompute these values during a modification request.

For this purpose we keep store an additional value for each vertex. In this value we store the addends we haven't propagated to the child vertices. Before traversing to a child vertex, we call `push` and propagate the value to both children. We have to do this in both the update function and the query function.

```

void push(int v) {
    t[v*2] += lazy[v];
    lazy[v*2] += lazy[v];
    t[v*2+1] += lazy[v];
    lazy[v*2+1] += lazy[v];
    lazy[v] = 0;
}

void update(int v, int tl, int tr, int l, int r, int addend) {
    if (l > r)
        return;
    if (l == tl && tr == r) {
        t[v] += addend;
        lazy[v] += addend;
    } else {
        push(v);
        int tm = (tl + tr) / 2;
        update(v*2, tl, tm, l, min(r, tm), addend);
        update(v*2+1, tm+1, tr, max(l, tm+1), r, addend);
        t[v] = max(t[v*2], t[v*2+1]);
    }
}

int query(int v, int tl, int tr, int l, int r) {
    if (l > r)

```

```

        return -INF;
    if (tl == tr)
        return t[v];
    push(v);
    int tm = (tl + tr) / 2;
    return max(query(v*2, tl, tm, l, min(r, tm)),
               query(v*2+1, tm+1, tr, max(l, tm+1), r));
}

```

Generalization to higher dimensions

A Segment Tree can be generalized quite natural to higher dimensions. If in the one-dimensional case we split the indices of the array into segments, then in the two-dimensional we make an ordinary Segment Tree with respect to the first indices, and for each segment we build an ordinary Segment Tree with respect to the second indices.

Simple 2D Segment Tree A matrix $a[0 \dots n - 1, 0 \dots m - 1]$ is given, and we have to find the sum (or minimum/maximum) on some submatrix $a[x_1 \dots x_2, y_1 \dots y_2]$, as well as perform modifications of individual matrix elements (i.e. queries of the form $a[x][y] = p$).

So we build a 2D Segment Tree: first the Segment Tree using the first coordinate (x), then the second (y).

To make the construction process more understandable, you can forget for a while that the matrix is two-dimensional, and only leave the first coordinate. We will construct an ordinary one-dimensional Segment Tree using only the first coordinate. But instead of storing a number in a segment, we store an entire Segment Tree: i.e. at this moment we remember that we also have a second coordinate; but because at this moment the first coordinate is already fixed to some interval $[l \dots r]$, we actually work with such a strip $a[l \dots r, 0 \dots m - 1]$ and for it we build a Segment Tree.

Here is the implementation of the construction of a 2D Segment Tree. It actually represents two separate blocks: the construction of a Segment Tree along the x coordinate (build_x), and the y coordinate (build_y). For the leaf nodes in build_y we have to separate two cases: when the current segment of the first coordinate $[tlx \dots trx]$ has length 1, and when it has a length greater than one. In the first case, we just take the corresponding value from the matrix, and in the second case we can combine the values of two Segment Trees from the left and the right son in the coordinate x .

```

void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
    if (ly == ry) {
        if (lx == rx)
            t[vx][vy] = a[lx][ly];
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    } else {
        int my = (ly + ry) / 2;

```

```

        build_y(vx, lx, rx, vy*2, ly, my);
        build_y(vx, lx, rx, vy*2+1, my+1, ry);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}

void build_x(int vx, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        build_x(vx*2, lx, mx);
        build_x(vx*2+1, mx+1, rx);
    }
    build_y(vx, lx, rx, 1, 0, m-1);
}

```

Such a Segment Tree still uses a linear amount of memory, but with a larger constant: $16nm$. It is clear that the described procedure build_x also works in linear time.

Now we turn to processing of queries. We will answer to the two-dimensional query using the same principle: first break the query on the first coordinate, and then for every reached vertex, we call the corresponding Segment Tree of 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 $O(\log n \log m)$ time, since it first descends the tree in the first coordinate, and for each traversed vertex in the tree it makes a query in the corresponding Segment Tree along the second coordinate.

Finally we consider the modification query. We want to learn how to modify the Segment Tree in accordance with the change in the value of some element $a[x][y] = p$. It is clear, that the changes will occur only in those vertices of the first Segment Tree that cover the coordinate x (and such will be $O(\log n)$), and for Segment Trees corresponding to them the changes will only occur at those

vertices that covers the coordinate y (and such will be $O(\log m)$). Therefore the implementation will be not very different from the one-dimensional case, only now we first descend the first coordinate, and then 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 2D Segment Tree Let the problem be the following: there are n points on the plane given by their coordinates (x_i, y_i) and queries of the form “count the number of points lying in the rectangle $((x_1, y_1), (x_2, y_2))$ ”. It is clear that in the case of such a problem it becomes unreasonably wasteful to construct a two-dimensional Segment Tree with $O(n^2)$ elements. Most on this memory will be wasted, since each single point can only get into $O(\log n)$ segments of the tree along the first coordinate, and therefore the total “useful” size of all tree segments on the second coordinate is $O(n \log n)$.

So we proceed as follows: at each vertex of the Segment Tree with respect to the first coordinate we store a Segment Tree constructed only by those second coordinates that occur in the current segment of the first coordinates. In other words, when constructing a Segment Tree inside some vertex with index vx and the boundaries tlx and trx , we only consider those points that fall into this interval $x \in [tlx, trx]$, and build a Segment Tree just using them.

Thus we will achieve that each Segment Tree on the second coordinate will occupy exactly as much memory as it should. As a result, the total amount of memory will decrease to $O(n \log n)$. We still can answer the queries in $O(\log^2 n)$ time, we just have to make a binary search on the second coordinate, but this

will not worsen the complexity.

But modification queries will be impossible with this structure: in fact if a new point appears, we have to add a new element in the middle of some Segment Tree along the second coordinate, which cannot be effectively done.

In conclusion we note that the two-dimensional Segment Tree contracted in the described way becomes practically equivalent to the modification of the one-dimensional Segment Tree (see Saving the entire subarrays in each vertex). In particular the two-dimensional Segment Tree is just a special case of storing a subarray in each vertex of the tree. It follows, that if you gave to abandon a two-dimensional Segment Tree due to the impossibility of executing a query, it makes sense to try to replace the nested Segment Tree with some more powerful data structure, for example a Cartesian tree.

Preserving the history of its values (Persistent Segment Tree)

A persistent data structure is a data structure that remembers its previous state for each modification. This allows to access any version of this data structure that interests us and execute a query on it.

Segment Tree is a data structure that can be turned into a persistent data structure efficiently (both in time and memory consumption). We want to avoid copying the complete tree before each modification, and we don't want to lose the $O(\log n)$ time behavior for answering range queries.

In fact, any change request in the Segment Tree leads to a change in the data of only $O(\log n)$ vertices along the path starting from the root. So if we store the Segment Tree using pointers (i.e. a vertex stores pointers to the left and the right child vertices), then when performing the modification query, we simply need to create new vertices instead of changing the available vertices. Vertices that are not affected by the modification query can still be used by pointing the pointers to the old vertices. Thus for a modification query $O(\log n)$ new vertices will be created, including a new root vertex of the Segment Tree, and the entire previous version of the tree rooted at the old root vertex will remain unchanged.

Let's give an example implementation for the simplest Segment Tree: when there is only a query asking for sums, and modification queries of single elements.

```
struct Vertex {
    Vertex *l, *r;
    int sum;

    Vertex(int val) : l(nullptr), r(nullptr), 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* v, int tl, int tr, int l, int r) {
    if (l > r)
        return 0;
    if (l == tl && tr == r)
        return v->sum;
    int tm = (tl + tr) / 2;
    return get_sum(v->l, tl, tm, l, min(r, tm))
        + get_sum(v->r, tm+1, tr, max(l, tm+1), r);
}

Vertex* update(Vertex* v, 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(v->l, tl, tm, pos, new_val), v->r);
    else
        return new Vertex(v->l, update(v->r, tm+1, tr, pos, new_val));
}

```

For each modification of the Segment Tree we will receive a new root vertex. To quickly jump between two different versions of the Segment Tree, we need to store this roots in an array. To use a specific version of the Segment Tree we simply call the query using the appropriate root vertex.

With the approach described above almost any Segment Tree can be turned into a persistent data structure.

Finding the k -th smallest number in a range This time we have to answer queries of the form "What is the k -th smallest element in the range $a[l \dots r]$ ". This query can be answered using a binary search and a Merge Sort Tree, but the time complexity for a single query would be $O(\log^3 n)$. We will accomplish the same task using a persistent Segment Tree in $O(\log n)$.

First we will discuss a solution for a simpler problem: We will only consider arrays in which the elements are bound by $0 \leq a[i] < n$. And we only want to find the k -th smallest element in some prefix of the array a . It will be very easy to extend the developed ideas later for not restricted arrays and not restricted range queries.

We will use a Segment Tree that counts all appearing numbers, i.e. in the Segment Tree we will store the histogram of the array. So the leaf vertices will store how often the values $0, 1, \dots, n-1$ will appear in the array, and the other vertices store how many numbers in some range are in the array. In other words we create a regular Segment Tree with sum queries over the histogram of the array. But instead of creating all n Segment Trees for every possible prefix, we will create one persistent one, that will contain the same information. We will start with an empty Segment Tree (all counts will be 0), and add the elements

$a[0], a[1], \dots, a[n-1]$ one after each other. For each modification we will receive a new root vertex, let's call $root_i$ the root of the Segment Tree after inserting the first i elements of the array a . The Segment Tree rooted at $root_i$ will contain the histogram of the prefix $a[0 \dots i-1]$. Using this Segment Tree we can find in $O(\log n)$ time the position of the k -th element using the same technique discussed in Counting the number of zeros, searching for the k -th zero.

Now to the not-restricted version of the problem.

First for the restriction on the array: We can actually transform any array to such an array by index compression. The smallest element in the array will get assigned the value 0, the second smallest the value 1, and so forth. It is easy to generate lookup tables (e.g. using map), that convert a value to its index and vice versa in $O(\log n)$ time.

Now to the queries restriction: instead of only performing these queries over a prefix of a , we want to use any arbitrary segments $a[l \dots r]$. Here we need a Segment Tree that represents the histogram of the elements in the range $a[l \dots r]$. It is easy to see that such a Segment Tree is just the difference between the Segment Tree rooted at $root_{r+1}$ and the Segment Tree rooted at $root_l$, i.e. every vertex in the $[l \dots r]$ Segment Tree can be computed with the vertex of the $root_{r+1}$ tree minus the vertex of the $root_l$ tree. In the implementation of the `get_kth` function this can be handled by passing two vertex pointer and computing the count/sum of the current segment as difference of the two counts/sums of the vertices.

2.5 Treap (Cartesian tree)

Treap is a data structure which combines binary tree and binary heap (hence the name: tree + heap \Rightarrow Treap).

More specifically, treap is a data structure that stores pairs (X, Y) in a binary tree in such a way that it is a binary search tree by X and a binary heap by Y . Assuming that all X and all Y are different, we can see that if some node of the tree contains values (X_0, Y_0) , all nodes in the left subtree have $X < X_0$, all nodes in the right subtree have $X > X_0$, and all nodes in both left and right subtrees have $Y < Y_0$.

Treaps have been proposed by Siedel and Aragon in 1989.

2.5.1 Advantages of such data organisation

In such implementation X values are the keys (and at same time the values stored in the treap), and Y values are called **priorities**. Without priorities, the treap would be a regular binary search tree by X , and one set of X values could correspond to a lot of different trees, some of them degenerate (for example, in the form of a linked list), and therefore extremely slow (the main operations would have $O(N)$ complexity).

At the same time, **priorities** allow to **uniquely** specify the tree that will be constructed (of course, it does not depend on the order in which values are added), which can be proven using corresponding theorem. Obviously, if you **choose the priorities randomly**, you will get non-degenerate trees on average, which will ensure $O(\log N)$ complexity for the main operations. Hence another name of this data structure - **randomized binary search tree**.

2.5.2 Operations

A treap provides the following operations:

- **Insert(X, Y)** in $O(\log N)$.
Adds a new node to the tree. One possible variant is to pass only X and generate Y randomly inside the operation (while ensuring that it's different from all other priorities in the tree).
- **Search (X)** in $O(\log N)$.
Looks for a node with the specified key value X . The implementation is the same as for an ordinary binary search tree.
- **Erase (X)** in $O(\log N)$.
Looks for a node with the specified key value X and removes it from the tree.
- **Build (X_1, \dots, X_N)** in $O(N)$.
Builds a tree from a list of values. This can be done in linear time (assuming that X_1, \dots, X_N are sorted), but we will not discuss this implementation here. We will just use N serial calls of **Insert** operation, which has $O(N \log N)$ complexity.

- **Union** (T_1, T_2) in $O(M \log(N/M))$.
Merges two trees, assuming that all the elements are different. It is possible to achieve the same complexity if duplicate elements should be removed during merge.
- **Intersect** (T_1, T_2) in $O(M \log(N/M))$.
Finds the intersection of two trees (i.e. their common elements). We will not consider the implementation of this operation here.

In addition, due to the fact that a treap is a binary search tree, it can implement other operations, such as finding the K-th largest element or finding the index of an element.

2.5.3 Implementation Description

In terms of implementation, each node contains X, Y and pointers to the left (L) and right (R) children.

We will implement all the required operations using just two auxiliary operations: Split and Merge.

Split (**T**, **X**) separates tree T in 2 subtrees L and R trees (which are the return values of split) so that L contains all elements with key $X_L < X$, and R contains all elements with key $X_R > X$. This operation has $O(\log N)$ complexity and is implemented using an obvious recursion.

Merge (T_1, T_2) combines two subtrees T_1 and T_2 and returns the new tree. This operation also has $O(\log N)$ complexity. It works under the assumption that T_1 and T_2 are ordered (all keys X in T_1 are smaller than keys in T_2). Thus, we need to combine these trees without violating the order of priorities Y. To do this, we choose as the root the tree which has higher priority Y in the root node, and recursively call Merge for the other tree and the corresponding subtree of the selected root node.

Now implementation of **Insert** (**X**, **Y**) becomes obvious. First we descend in the tree (as in a regular binary search tree by X), and stop at the first node in which the priority value is less than Y. We have found the place where we will insert the new element. Next, we call **Split** (**T**, **X**) on the subtree starting at the found node, and use returned subtrees L and R as left and right children of the new node.

Implementation of **Erase** (**X**) is also clear. First we descend in the tree (as in a regular binary search tree by X), looking for the element we want to delete. Once the node is found, we call **Merge** on it children and put the return value of the operation in the place of the element we're deleting.

We implement **Build** operation with $O(N \log N)$ complexity using N **Insert** calls.

Union (T_1, T_2) has theoretical complexity $O(M \log(N/M))$, but in practice it works very well, probably with a very small hidden constant. Let's assume without loss of generality that $T_1 \rightarrow Y > T_2 \rightarrow Y$, i. e. root of T_1 will be the root of the result. To get the result, we need to merge trees $T_1 \rightarrow L$, $T_1 \rightarrow R$ and T_2 in two trees which could be children of T_1 root. To do this, we call Split ($T_2, T_1 \rightarrow X$), thus splitting T_2 in two parts L and R, which we then recursively

combine with children of T_1 : Union ($T_1 \rightarrow L, L$) and Union ($T_1 \rightarrow R, R$), thus getting left and right subtrees of the result.

2.5.4 Implementation

```

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;
}

```

2.5.5 Maintaining the sizes of subtrees

To extend the functionality of the treap, it is often necessary to store the number of nodes in subtree of each node - field `int cnt` in the `item` structure. For example, it can be used to find K -th largest element of tree in $O(\log N)$, or to find the index of the element in the sorted list with the same complexity. The implementation of these operations will be the same as for the regular binary search tree.

When a tree changes (nodes are added or removed etc.), `cnt` of some nodes should be updated accordingly. We'll create two functions: `cnt()` will return the current value of `cnt` or 0 if the node does not exist, and `upd_cnt()` will update the value of `cnt` for this node assuming that for its children `L` and `R` the values of `cnt` have already been updated. Evidently it's sufficient to add calls of `upd_cnt()` to the end of `insert`, `erase`, `split` and `merge` to keep `cnt` values up-to-date.

```

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);
}

```

2.5.6 Building a Treap in $O(N)$ in offline mode

TODO

2.5.7 Implicit Treaps

Implicit treap is a simple modification of the regular treap which is a very powerful data structure. In fact, implicit treap can be considered as an array with the following procedures implemented (all in $O(\log N)$ in the online mode):

- Inserting an element in the array in any location
- Removal of an arbitrary element
- Finding sum, minimum / maximum element etc. on an arbitrary interval
- Addition, painting on an arbitrary interval
- Reversing elements on an arbitrary interval

The idea is that the keys should be **indices** of the elements in the array. But we will not store these values explicitly (otherwise, for example, inserting an element would cause changes of the key in $O(N)$ nodes of the tree).

Note that the key of a node is the number of nodes less than it (such nodes can be present not only in its left subtree but also in left subtrees of its ancestors). More specifically, the **implicit key** for some node T is the number of vertices $\text{cnt}(T \rightarrow L)$ in the left subtree of this node plus similar values $\text{cnt}(P \rightarrow L) + 1$ for each ancestor P of the node T , if T is in the right subtree of P .

Now it's clear how to calculate the implicit key of current node quickly. Since in all operations we arrive to any node by descending in the tree, we can just accumulate this sum and pass it to the function. If we go to the left subtree, the accumulated sum does not change, if we go to the right subtree it increases by $\text{cnt}(T \rightarrow L) + 1$.

Here are the new implementations of **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); //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);
}
```

Now let's consider the implementation of various operations on implicit treaps:

- **Insert element.**

Suppose we need to insert an element at position `pos`. We divide the treap into two parts, which correspond to arrays `[0..pos-1]` and `[pos..sz]`; to do this we call `split (T, T1, T2, pos)`. Then we can combine tree T_1 with the new vertex by calling `merge (T1, T1, new_item)` (it is easy to see that all preconditions are met). Finally, we combine trees T_1 and T_2 back into T by calling `merge (T, T1, T2)`.

- **Delete element.**

This operation is even easier: find the element to be deleted T , perform merge of its children L and R , and replace the element T with the result of merge. In fact, element deletion in the implicit treap is exactly the same as in the regular treap.

- Find **sum** / **minimum**, etc. on the interval.

First, create an additional field `F` in the `item` structure to store the value

of the target function for this node's subtree. This field is easy to maintain similarly to maintaining sizes of subtrees: create a function which calculates this value for a node based on values for its children and add calls of this function in the end of all functions which modify the tree.

Second, we need to know how to process a query for an arbitrary interval $[A; B]$.

To get a part of tree which corresponds to the interval $[A; B]$, we need to call `split (T, T1, T2, A)`, and then `split (T2, T2, T3, B - A + 1)`: after this T_2 will consist of all the elements in the interval $[A; B]$, and only of them. Therefore, the response to the query will be stored in the field F of the root of T_2 . After the query is answered, the tree has to be restored by calling `merge (T, T1, T2)` and `merge (T, T, T3)`.

- **Addition / painting** on the interval.

We act similarly to the previous paragraph, but instead of the field F we will store a field `add` which will contain the added value for the subtree (or the value to which the subtree is painted). Before performing any operation we have to “push” this value correctly - i.e. change $T \rightarrow L \rightarrow add$ and $T \rightarrow R \rightarrow add$, and to clean up `add` in the parent node. This way after any changes to the tree the information will not be lost.

- **Reverse** on the interval.

This is again similar to the previous operation: we have to add boolean flag ‘rev’ and set it to true when the subtree of the current node has to be reversed. “Pushing” this value is a bit complicated - we swap children of this node and set this flag to true for them.

Here is an example implementation of the implicit treap with reverse on the interval. For each node we store field called `value` which is the actual value of the array element at current position. We also provide implementation of the function `output()`, which outputs an array that corresponds to the current state of the implicit treap.

```
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);
}

```

2.5.8 Literature

- [Blleloch, Reid-Miller “Fast Set Operations Using Treaps”](#)

2.5.9 Practice Problems

- [SPOJ - Ada and Aphids](#)
- [SPOJ - Ada and Harvest](#)
- [Codeforces - Radio Stations](#)
- [SPOJ - Ghost Town](#)
- [SPOJ - Arrangement Validity](#)
- [SPOJ - All in One](#)
- [Codeforces - Dog Show](#)
- [Codeforces - Yet Another Array Queries Problem](#)
- [SPOJ - Mean of Array](#)
- [SPOJ - TWIST](#)
- [SPOJ - KOILINE](#)

2.6 Sqrt Tree

Given an array a that contains n elements and the operation \circ that satisfies associative property: $(x \circ y) \circ z = x \circ (y \circ z)$ is true for any x, y, z .

So, such operations as gcd, min, max, +, and, or, xor, etc. satisfy these conditions.

Also we have some queries $q(l, r)$. For each query, we need to compute $a_l \circ a_{l+1} \circ \dots \circ a_r$.

Sqrt Tree can process such queries in $O(1)$ time with $O(n \cdot \log \log n)$ preprocessing time and $O(n \cdot \log \log n)$ memory.

2.6.1 Description

Building sqrt decomposition

Let's make a [sqrt decomposition](#). We divide our array in \sqrt{n} blocks, each block has size \sqrt{n} . For each block, we compute:

1. Answers to the queries that lie in the block and begin at the beginning of the block (prefixOp)
2. Answers to the queries that lie in the block and end at the end of the block (suffixOp)

And we'll compute an additional array:

3. $\text{between}_{i,j}$ (for $i \leq j$) - answer to the query that begins at the start of block i and ends at the end of block j . Note that we have \sqrt{n} blocks, so the size of this array will be $O(\sqrt{n}^2) = O(n)$.

Let's see the example.

Let \circ be + (we calculate sum on a segment) and we have the following array a :

{1, 2, 3, 4, 5, 6, 7, 8, 9}

It will be divided onto three blocks: {1, 2, 3}, {4, 5, 6} and {7, 8, 9}.

For first block prefixOp is {1, 3, 6} and suffixOp is {6, 5, 3}.

For second block prefixOp is {4, 9, 15} and suffixOp is {15, 11, 6}.

For third block prefixOp is {7, 15, 24} and suffixOp is {24, 17, 9}.

between array is:

```
{
    {6, 21, 45},
    {0, 15, 39},
    {0, 0, 24}
}
```

(we assume that invalid elements where $i > j$ are filled with zeroes)

It's obvious to see that these arrays can be easily calculated in $O(n)$ time and memory.

We already can answer some queries using these arrays. If the query doesn't fit into one block, we can divide it into three parts: suffix of a block, then some segment of contiguous blocks and then prefix of some block. We can answer a query by dividing it into three parts and taking our operation of some value from suffixOp, then some value from between, then some value from prefixOp.

But if we have queries that entirely fit into one block, we cannot process them using these three arrays. So, we need to do something.

Making a tree

We cannot answer only the queries that entirely fit in one block. But what **if we build the same structure as described above for each block?** Yes, we can do it. And we do it recursively, until we reach the block size of 1 or 2. Answers for such blocks can be calculated easily in $O(1)$.

So, we get a tree. Each node of the tree represents some segment of the array. Node that represents array segment with size k has \sqrt{k} children – for each block. Also each node contains the three arrays described above for the segment it contains. The root of the tree represents the entire array. Nodes with segment lengths 1 or 2 are leaves.

Also it's obvious that the height of this tree is $O(\log \log n)$, because if some vertex of the tree represents an array with length k , then its children have length \sqrt{k} . $\log(\sqrt{k}) = \frac{\log k}{2}$, so $\log k$ decreases two times every layer of the tree and so its height is $O(\log \log n)$. The time for building and memory usage will be $O(n \cdot \log \log n)$, because every element of the array appears exactly once on each layer of the tree.

Now we can answer the queries in $O(\log \log n)$. We can go down on the tree until we meet a segment with length 1 or 2 (answer for it can be calculated in $O(1)$ time) or meet the first segment in which our query doesn't fit entirely into one block. See the first section on how to answer the query in this case.

OK, now we can do $O(\log \log n)$ per query. Can it be done faster?

Optimizing the query complexity

One of the most obvious optimization is to binary search the tree node we need. Using binary search, we can reach the $O(\log \log \log n)$ complexity per query. Can we do it even faster?

The answer is yes. Let's assume the following two things:

1. Each block size is a power of two.
2. All the blocks are equal on each layer.

To reach this, we can add some zero elements to our array so that its size becomes a power of two.

When we use this, some block sizes may become twice larger to be a power of two, but it still be $O(\sqrt{k})$ in size and we keep linear complexity for building the arrays in a segment.

Now, we can easily check if the query fits entirely into a block with size 2^k . Let's write the ranges of the query, l and r (we use 0-indexation) in binary form. For instance, let's assume $k = 4, l = 39, r = 46$. The binary representation of l and r is:

$$l = 39_{10} = 100111_2$$

$$r = 46_{10} = 101110_2$$

Remember that one layer contains segments of the equal size, and the block on one layer have also equal size (in our case, their size is $2^k = 2^4 = 16$. The blocks cover the array entirely, so the first block covers elements $(0 - 15)$ ($(000000_2 - 001111_2)$ in binary), the second one covers elements $(16 - 31)$ ($(010000_2 - 011111_2)$ in binary) and so on. We see that the indices of the positions covered by one block may differ only in k (in our case, 4) last bits. In our case l and r have equal bits except four lowest, so they lie in one block.

So, we need to check if nothing more than k smallest bits differ (or $l \text{ xor } r$ doesn't exceed $2^k - 1$).

Using this observation, we can find a layer that is suitable to answer the query quickly. How to do this:

1. For each i that doesn't exceed the array size, we find the highest bit that is equal to 1. To do this quickly, we use DP and a precalculated array.
2. Now, for each $q(l, r)$ we find the highest bit of $l \text{ xor } r$ and, using this information, it's easy to choose the layer on which we can process the query easily. We can also use a precalculated array here.

For more details, see the code below.

So, using this, we can answer the queries in $O(1)$ each. Hooray! :)

2.6.2 Updating elements

We can also update elements in Sqrt Tree. Both single element updates and updates on a segment are supported.

Updating a single element

Consider a query $\text{update}(x, val)$ that does the assignment $a_x = val$. We need to perform this query fast enough.

Naive approach First, let's take a look of what is changed in the tree when a single element changes. Consider a tree node with length l and its arrays: `prefixOp`, `suffixOp` and `between`. It is easy to see that only $O(\sqrt{l})$ elements from `prefixOp` and `suffixOp` change (only inside the block with the changed element). $O(l)$ elements are changed in `between`. Therefore, $O(l)$ elements in the tree node are updated.

We remember that any element x is present in exactly one tree node at each layer. Root node (layer 0) has length $O(n)$, nodes on layer 1 have length $O(\sqrt{n})$,

nodes on layer 2 have length $O(\sqrt{\sqrt{n}})$, etc. So the time complexity per update is $O(n + \sqrt{n} + \sqrt{\sqrt{n}} + \dots) = O(n)$.

But it's too slow. Can it be done faster?

An sqrt-tree inside the sqrt-tree Note that the bottleneck of updating is rebuilding between of the root node. To optimize the tree, let's get rid of this array! Instead of between array, we store another sqrt-tree for the root node. Let's call it index. It plays the same role as between— answers the queries on segments of blocks. Note that the rest of the tree nodes don't have index, they keep their between arrays.

A sqrt-tree is *indexed*, if its root node has index. A sqrt-tree with between array in its root node is *unindexed*. Note that index is ***unindexed*** itself.

So, we have the following algorithm for updating an *indexed* tree:

- Update prefixOp and suffixOp in $O(\sqrt{n})$.
- Update index. It has length $O(\sqrt{n})$ and we need to update only one item in it (that represents the changed block). So, the time complexity for this step is $O(\sqrt{n})$. We can use the the algorithm described in the beginning of this section (the “slow” one) to do it.
- Go into the child node that represents the changed block and update it in $O(\sqrt{n})$ with the “slow” algorithm.

Note that the query complexity is still $O(1)$: we need to use index in query no more than once, and this will take $O(1)$ time.

So, total time complexity for updating a single element is $O(\sqrt{n})$. Hooray! :)

Updating a segment

Sqrt-tree also can do things like assigning an element on a segment. $\text{massUpdate}(x, l, r)$ means $a_i = x$ for all $l \leq i \leq r$.

There are two approaches to do this: one of them does massUpdate in $O(\sqrt{n} \cdot \log \log n)$, keeping $O(1)$ per query. The second one does massUpdate in $O(\sqrt{n})$, but the query complexity becomes $O(\log \log n)$.

We will do lazy propagation in the same way as it is done in segment trees: we mark some nodes as *lazy*, meaning that we'll push them when it's necessary. But one thing is different from segment trees: pushing a node is expensive, so it cannot be done in queries. On the layer 0, pushing a node takes $O(\sqrt{n})$ time. So, we don't push nodes inside queries, we only look if the current node or its parent are *lazy*, and just take it into account while performing queries.

First approach In the first approach, we say that only nodes on layer 1 (with length $O(\sqrt{n})$) can be *lazy*. When pushing such node, it updates all its subtree including itself in $O(\sqrt{n} \cdot \log \log n)$. The massUpdate process is done as follows:

- Consider the nodes on layer 1 and blocks corresponding to them.

- Some blocks are entirely covered by massUpdate. Mark them as *lazy* in $O(\sqrt{n})$.
- Some blocks are partially covered. Note there are no more than two blocks of this kind. Rebuild them in $O(\sqrt{n} \cdot \log \log n)$. If they were *lazy*, take it into account.
- Update prefixOp and suffixOp for partially covered blocks in $O(\sqrt{n})$ (because there are only two such blocks).
- Rebuild the index in $O(\sqrt{n} \cdot \log \log n)$.

So we can do massUpdate fast. But how lazy propagation affects queries? They will have the following modifications:

- If our query entirely lies in a *lazy* block, calculate it and take *lazy* into account. $O(1)$.
- If our query consists of many blocks, some of which are *lazy*, we need to take care of *lazy* only on the leftmost and the rightmost block. The rest of the blocks are calculated using index, which already knows the answer on *lazy* block (because it's rebuilt after each modification). $O(1)$.

The query complexity still remains $O(1)$.

Second approach In this approach, each node can be *lazy* (except root). Even nodes in index can be *lazy*. So, while processing a query, we have to look for *lazy* tags in all the parent nodes, i. e. query complexity will be $O(\log \log n)$.

But massUpdate becomes faster. It looks in the following way:

- Some blocks are fully covered with massUpdate. So, *lazy* tags are added to them. It is $O(\sqrt{n})$.
- Update prefixOp and suffixOp for partially covered blocks in $O(\sqrt{n})$ (because there are only two such blocks).
- Do not forget to update the index. It is $O(\sqrt{n})$ (we use the same massUpdate algorithm).
- Update between array for *unindexed* subtrees.
- Go into the nodes representing partially covered blocks and call massUpdate recursively.

Note that when we do the recursive call, we do prefix or suffix massUpdate. But for prefix and suffix updates we can have no more than one partially covered child. So, we visit one node on layer 1, two nodes on layer 2 and two nodes on any deeper level. So, the time complexity is $O(\sqrt{n} + \sqrt{\sqrt{n}} + \dots) = O(\sqrt{n})$. The approach here is similar to the segment tree mass update.

2.6.3 Implementation

The following implementation of Sqrt Tree can perform the following operations: build in $O(n \cdot \log \log n)$, answer queries in $O(1)$ and update an element in $O(\sqrt{n})$.

```
SqrtTreeItem op(const SqrtTreeItem &a, const SqrtTreeItem &b);

inline int log2Up(int n) {
    int res = 0;
    while ((1 << res) < n) {
        res++;
    }
    return res;
}

class SqrtTree {
private:
    int n, lg, indexSz;
    vector<SqrtTreeItem> v;
    vector<int> clz, layers, onLayer;
    vector< vector<SqrtTreeItem> > pref, suf, between;

    inline void buildBlock(int layer, int l, int r) {
        pref[layer][l] = v[l];
        for (int i = l+1; i < r; i++) {
            pref[layer][i] = op(pref[layer][i-1], v[i]);
        }
        suf[layer][r-1] = v[r-1];
        for (int i = r-2; i >= l; i--) {
            suf[layer][i] = op(v[i], suf[layer][i+1]);
        }
    }

    inline void buildBetween(int layer, int lBound, int rBound, int betweenOffs) {
        int bSzLog = (layers[layer]+1) >> 1;
        int bCntLog = layers[layer] >> 1;
        int bSz = 1 << bSzLog;
        int bCnt = (rBound - lBound + bSz - 1) >> bSzLog;
        for (int i = 0; i < bCnt; i++) {
            SqrtTreeItem ans;
            for (int j = i; j < bCnt; j++) {
                SqrtTreeItem add = suf[layer][lBound + (j << bSzLog)];
                ans = (i == j) ? add : op(ans, add);
                between[layer-1][betweenOffs + lBound + (i << bCntLog) + j] = ans;
            }
        }
    }

    inline void buildBetweenZero() {
        int bSzLog = (lg+1) >> 1;
        for (int i = 0; i < indexSz; i++) {
            v[n+i] = suf[0][i << bSzLog];
        }
    }
};
```

```

        build(1, n, n + indexSz, (1 << lg) - n);
    }

    inline void updateBetweenZero(int bid) {
        int bSzLog = (lg+1) >> 1;
        v[n+bid] = suf[0][bid << bSzLog];
        update(1, n, n + indexSz, (1 << lg) - n, n+bid);
    }

    void build(int layer, int lBound, int rBound, int betweenOffs) {
        if (layer >= (int)layers.size()) {
            return;
        }
        int bSz = 1 << ((layers[layer]+1) >> 1);
        for (int l = lBound; l < rBound; l += bSz) {
            int r = min(l + bSz, rBound);
            buildBlock(layer, l, r);
            build(layer+1, l, r, betweenOffs);
        }
        if (layer == 0) {
            buildBetweenZero();
        } else {
            buildBetween(layer, lBound, rBound, betweenOffs);
        }
    }

    void update(int layer, int lBound, int rBound, int betweenOffs, int x) {
        if (layer >= (int)layers.size()) {
            return;
        }
        int bSzLog = (layers[layer]+1) >> 1;
        int bSz = 1 << bSzLog;
        int blockIdx = (x - lBound) >> bSzLog;
        int l = lBound + (blockIdx << bSzLog);
        int r = min(l + bSz, rBound);
        buildBlock(layer, l, r);
        if (layer == 0) {
            updateBetweenZero(blockIdx);
        } else {
            buildBetween(layer, lBound, rBound, betweenOffs);
        }
        update(layer+1, l, r, betweenOffs, x);
    }

    inline SqrtTreeItem query(int l, int r, int betweenOffs, int base) {
        if (l == r) {
            return v[l];
        }
        if (l + 1 == r) {
            return op(v[l], v[r]);
        }
        int layer = onLayer[clz[(l - base) ^ (r - base)]];

```

```

    int bSzLog = (layers[layer]+1) >> 1;
    int bCntLog = layers[layer] >> 1;
    int lBound = (((1 - base) >> layers[layer]) << layers[layer]) + base;
    int lBlock = ((1 - lBound) >> bSzLog) + 1;
    int rBlock = ((r - lBound) >> bSzLog) - 1;
    SqrtTreeItem ans = suf[layer][l];
    if (lBlock <= rBlock) {
        SqrtTreeItem add = (layer == 0) ? (
            query(n + lBlock, n + rBlock, (1 << lg) - n, n)
        ) : (
            between[layer-1][betweenOffs + lBound + (lBlock << bCntLog) + rBlock]
        );
        ans = op(ans, add);
    }
    ans = op(ans, pref[layer][r]);
    return ans;
}

public:
    inline SqrtTreeItem query(int l, int r) {
        return query(l, r, 0, 0);
    }

    inline void update(int x, const SqrtTreeItem &item) {
        v[x] = item;
        update(0, 0, n, 0, x);
    }

SqrtTree(const vector<SqrtTreeItem>& a)
    : n((int)a.size()), lg(log2Up(n)), v(a), clz(1 << lg), onLayer(lg+1) {
    clz[0] = 0;
    for (int i = 1; i < (int)clz.size(); i++) {
        clz[i] = clz[i >> 1] + 1;
    }
    int tlg = lg;
    while (tlg > 1) {
        onLayer[tlg] = (int)layers.size();
        layers.push_back(tlg);
        tlg = (tlg+1) >> 1;
    }
    for (int i = lg-1; i >= 0; i--) {
        onLayer[i] = max(onLayer[i], onLayer[i+1]);
    }
    int betweenLayers = max(0, (int)layers.size() - 1);
    int bSzLog = (lg+1) >> 1;
    int bSz = 1 << bSzLog;
    indexSz = (n + bSz - 1) >> bSzLog;
    v.resize(n + indexSz);
    pref.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
    suf.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
    between.assign(betweenLayers, vector<SqrtTreeItem>((1 << lg) + bSz));
    build(0, 0, n, 0);
}

```

```
};
```

2.6.4 Problems

[CodeChef - SEGPROD](#)

2.7 Randomized Heap

A randomized heap is a heap that, through using randomization, allows to perform all operations in expected logarithmic time.

A **min heap** is a binary tree in which the value of each vertex is less than or equal to the values of its children. Thus the minimum of the tree is always in the root vertex.

A max heap can be defined in a similar way: by replacing less with greater.

The default operations of a heap are:

- Adding a value
- Extracting the minimum
- Removing the minimum
- Merging two heaps (without deleting duplicates)
- Removing an arbitrary element (if its position in the tree is known)

A randomized heap can perform all these operations in expected $O(\log n)$ time with a very simple implementation.

2.7.1 Data structure

We can immediately describe the structure of the binary heap:

```
struct Tree {  
    int value;  
    Tree * l = nullptr;  
    Tree * r = nullptr;  
};
```

In the vertex we store a value. In addition we have pointers to the left and right children, which are point to null if the corresponding child doesn't exist.

2.7.2 Operations

It is not difficult to see, that all operations can be reduced to a single one: **merging** two heaps into one. Indeed, adding a new value to the heap is equivalent to merging the heap with a heap consisting of a single vertex with that value. Finding a minimum doesn't require any operation at all - the minimum is simply the value at the root. Removing the minimum is equivalent to the result of merging the left and right children of the root vertex. And removing an arbitrary element is similar. We merge the children of the vertex and replace the vertex with the result of the merge.

So we actually only need to implement the operation of merging two heaps. All other operations are trivially reduced to this operation.

Let two heaps T_1 and T_2 be given. It is clear that the root of each of these heaps contains its minimum. So the root of the resulting heap will be the minimum of these two values. So we compare both values, and use the smaller one as the new root. Now we have to combine the children of the selected vertex

with the remaining heap. For this we select one of the children, and merge it with the remaining heap. Thus we again have the operation of merging two heaps. Sooner or later this process will end (the number of such steps is limited by the sum of the heights of the two heaps)

To achieve logarithmic complexity on average, we need to specify a method for choosing one of the two children so that the average path length is logarithmic. It is not difficult to guess, that we will make this decision **randomly**. Thus the implementation of the merging operation is 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;
}
```

Here first we check if one of the heaps is empty, then we don't need to perform any merge action at all. Otherwise we make the heap `t1` the one with the smaller value (by swapping `t1` and `t2` if necessary). We want to merge the left child of `t1` with `t2`, therefore we randomly swap the children of `t1`, and then perform the merge.

2.7.3 Complexity

We introduce the random variable $h(T)$ which will denote the **length of the random path** from the root to the leaf (the length in the number of edges). It is clear that the algorithm `merge` performs $O(h(T_1) + h(T_2))$ steps. Therefore to understand the complexity of the operations, we must look into the random variable $h(T)$.

Expected value

We assume that the expectation $h(T)$ can be estimated from above by the logarithm of the number of vertices in the heap:

$$\mathbf{E}h(T) \leq \log(n + 1)$$

This can be easily proven by induction. Let L and R be the left and the right subtrees of the root T , and n_L and n_R the number of vertices in them ($n = n_L + n_R + 1$).

The following shows the induction step:

$$\begin{aligned}
\mathbf{E}h(T) &= 1 + \frac{\mathbf{E}h(L) + \mathbf{E}h(R)}{2} \leq 1 + \frac{\log(n_L + 1) \log(n_R + 1)}{2} \\
&= 1 + \log \sqrt{(n_L + 1)(n_R + 1)} = \log 2 \sqrt{(n_L + 1)(n_R + 1)} \\
&\leq \log \frac{2((n_L + 1) + (n_R + 1))}{2} = \log(n_L + n_R + 2) = \log(n + 1)
\end{aligned}$$

Exceeding the expected value

Of course we are still not happy. The expected value of $h(T)$ doesn't say anything about the worst case. It is still possible that the paths from the root to the vertices is on average much greater than $\log(n + 1)$ for a specific tree.

Let us prove that exceeding the expected value is indeed very small:

$$P\{h(T) > (c + 1) \log n\} < \frac{1}{n^c}$$

for any positive constant c .

Here we denote by P the set of paths from the root of the heap to the leaves where the length exceeds $(c + 1) \log n$. Note that for any path p of length $|p|$ the probability that it will be chosen as random path is $2^{-|p|}$. Therefore we get:

$$P\{h(T) > (c + 1) \log n\} = \sum_{p \in P} 2^{-|p|} < \sum_{p \in P} 2^{-(c+1) \log n} = |P| n^{-(c+1)} \leq n^{-c}$$

Complexity of the algorithm

Thus the algorithm **merge**, and hence all other operations expressed with it, can be performed in $O(\log n)$ on average.

Moreover for any positive constant ϵ there is a positive constant c , such that the probability that the operation will require more than $c \log n$ steps is less than $n^{-\epsilon}$ (in some sense this describes the worst case behavior of the algorithm).

Chapter 3

Advanced

3.1 Deleting from a data structure in $O(T(n) \log n)$

Suppose you have a data structure which allows adding elements in **true** $O(T(n))$. This article will describe a technique that allows deletion in $O(T(n) \log n)$ offline.

3.1.1 Algorithm

Each element lives in the data structure for some segments of time between additions and deletions. Let's build a segment tree over the queries. Each segment when some element is alive splits into $O(\log n)$ nodes of the tree. Let's put each query when we want to know something about the structure into the corresponding leaf. Now to process all queries we will run a DFS on the segment tree. When entering the node we will add all the elements that are inside this node. Then we will go further to the children of this node or answer the queries (if the node is a leaf). When leaving the node, we must undo the additions. Note that if we change the structure in $O(T(n))$ we can roll back the changes in $O(T(n))$ by keeping a stack of changes. Note that rollbacks break amortized complexity.

3.1.2 Notes

The idea of creating a segment tree over segments when something is alive may be used not only for data structure problems. See some problems below.

3.1.3 Implementation

This implementation is for the [dynamic connectivity](#) problem. It can add edges, remove edges and count the number of connected components.

```
struct dsu_save {
    int v, rnkv, u, rnku;

    dsu_save() {}

    dsu_save(int _v, int _rnkv, int _u, int _rnku)
```



```

        : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};

struct dsu_with_rollback {
    vector<int> p, rnk;
    int comps;
    stack<dsu_save> op;

    dsu_with_rollback() {}

    dsu_with_rollback(int n) {
        p.resize(n);
        rnk.resize(n);
        for (int i = 0; i < n; i++) {
            p[i] = i;
            rnk[i] = 0;
        }
        comps = n;
    }

    int find_set(int v) {
        return (v == p[v]) ? v : find_set(p[v]);
    }

    bool unite(int v, int u) {
        v = find_set(v);
        u = find_set(u);
        if (v == u)
            return false;
        comps--;
        if (rnk[v] > rnk[u])
            swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u]));
        p[v] = u;
        if (rnk[u] == rnk[v])
            rnk[u]++;
        return true;
    }

    void rollback() {
        if (op.empty())
            return;
        dsu_save x = op.top();
        op.pop();
        comps++;
        p[x.v] = x.v;
        rnk[x.v] = x.rnk;
        p[x.u] = x.u;
        rnk[x.u] = x.rnk;
    }
};

```

```

struct query {
    int v, u;
    bool united;
    query(int _v, int _u) : v(_v), u(_u) {}
};

struct QueryTree {
    vector<vector<query>> t;
    dsu_with_rollback dsu;
    int T;

    QueryTree() {}

    QueryTree(int _T, int n) : T(_T) {
        dsu = dsu_with_rollback(n);
        t.resize(4 * T + 4);
    }

    void add_to_tree(int v, int l, int r, int ul, int ur, query& q) {
        if (ul > ur)
            return;
        if (l == ul && r == ur) {
            t[v].push_back(q);
            return;
        }
        int mid = (l + r) / 2;
        add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
        add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
    }

    void add_query(query q, int l, int r) {
        add_to_tree(1, 0, T - 1, l, r, q);
    }

    void dfs(int v, int l, int r, vector<int>& ans) {
        for (query& q : t[v]) {
            q.united = dsu.unite(q.v, q.u);
        }
        if (l == r)
            ans[l] = dsu.comps;
        else {
            int mid = (l + r) / 2;
            dfs(2 * v, l, mid, ans);
            dfs(2 * v + 1, mid + 1, r, ans);
        }
        for (query q : t[v]) {
            if (q.united)
                dsu.rollback();
        }
    }
}

```

```
vector<int> solve() {  
    vector<int> ans(T);  
    dfs(1, 0, T - 1, ans);  
    return ans;  
}  
};
```

3.1.4 Problems

- [Codeforces - Connect and Disconnect](#)
- [Codeforces - Addition on Segments](#)

Part III

Dynamic Programming

Chapter 1

DP optimizations

1.1 Divide and Conquer DP

Divide and Conquer is a dynamic programming optimization.

Preconditions

Some dynamic programming problems have a recurrence of this form:

$$dp(i, j) = \min_{k \leq j} \{dp(i-1, k) + C(k, j)\}$$

where $C(k, j)$ is some cost function.

Say $1 \leq i \leq n$ and $1 \leq j \leq m$, and evaluating C takes $O(1)$ time. Straightforward evaluation of the above recurrence is $O(nm^2)$. There are $n \times m$ states, and m transitions for each state.

Let $opt(i, j)$ be the value of k that minimizes the above expression. If $opt(i, j) \leq opt(i, j+1)$ for all i, j , then we can apply divide-and-conquer DP. This known as the *monotonicity condition*. The optimal “splitting point” for a fixed i increases as j increases.

This lets us solve for all states more efficiently. Say we compute $opt(i, j)$ for some fixed i and j . Then for any $j' < j$ we know that $opt(i, j') \leq opt(i, j)$. This means when computing $opt(i, j')$, we don't have to consider as many splitting points!

To minimize the runtime, we apply the idea behind divide and conquer. First, compute $opt(i, n/2)$. Then, compute $opt(i, n/4)$, knowing that it is less than or equal to $opt(i, n/2)$ and $opt(i, 3n/4)$ knowing that it is greater than or equal to $opt(i, n/2)$. By recursively keeping track of the lower and upper bounds on opt , we reach a $O(mn \log n)$ runtime. Each possible value of $opt(i, j)$ only appears in $\log n$ different nodes.

Note that it doesn't matter how “balanced” $opt(i, j)$ is. Across a fixed level, each value of k is used at most twice, and there are at most $\log n$ levels.

1.1.1 Generic implementation

Even though implementation varies based on problem, here's a fairly generic template. The function `compute` computes one row i of states `dp_cur`, given the previous row $i - 1$ of states `dp_before`. It has to be called with `compute(0, n-1, 0, n-1)`.

```
int n;
long long C(int i, int j);
vector<long long> dp_before(n), dp_cur(n);

// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr)
{
    if (l > r)
        return;
    int mid = (l + r) >> 1;
    pair<long long, int> best = {INF, -1};

    for (int k = optl; k <= min(mid, optr); k++) {
        best = min(best, {dp_before[k - 1] + C(k, mid), k});
    }

    dp_cur[mid] = best.first;
    int opt = best.second;

    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
}
```

Things to look out for

The greatest difficulty with Divide and Conquer DP problems is proving the monotonicity of *opt*. Many Divide and Conquer DP problems can also be solved with the Convex Hull trick or vice-versa. It is useful to know and understand both!

1.1.2 Practice Problems

- [Codeforces - Ciel and Gondolas](#) (Be careful with I/O!)
- [SPOJ - LARMY](#)
- [Codechef - CHEFAOR](#)
- [Hackerrank - Guardians of the Lunatics](#)
- [ACM ICPC World Finals 2017 - Money](#)

1.1.3 References

- [Quora Answer by Michael Levin](#)
- [Video Tutorial by "Sothe" the Algorithm Wolf](#)

Chapter 2

Tasks

2.1 Dynamic Programming on Broken Profile. Problem “Parquet”

Common problems solved using DP on broken profile include:

- finding number of ways to fully fill an area (e.g. chessboard/grid) with some figures (e.g. dominoes)
- finding a way to fill an area with minimum number of figures
- finding a partial fill with minimum number of unfilled space (or cells, in case of grid)
- finding a partial fill with the minimum number of figures, such that no more figures can be added

2.1.1 Problem “Parquet”

Problem description. Given a grid of size $N \times M$. Find number of ways to fill the grid with figures of size 2×1 (no cell should be left unfilled, and figures should not overlap each other).

Let the DP state be: $D[i][mask]$, where $i = 1 \dots N$, and $mask = 0 \dots 2^M - 1$. i represents number of rows in the current grid, and $mask$ is the state of last row of current grid. If j -th bit of $mask$ is 0 then the corresponding cell is filled, otherwise it is unfilled. Clearly, the answer to the problem will be $D[N][0]$.

We will be building the DP state by iterating over each $i = 1 \dots N$ and each $mask = 0 \dots 2^M - 1$, and for each $mask$ we will be only transitioning forward, that is, we will be *adding* figures to the current grid.

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];
}
```


2.2 Finding the largest zero submatrix

You are given a matrix with n rows and m columns. Find the largest submatrix consisting of only zeros (a submatrix is a rectangular area of the matrix).

2.2.1 Algorithm

Elements of the matrix will be $a[i][j]$, where $i = 0 \dots n - 1$, $j = 0 \dots m - 1$. For simplicity, we will consider all non-zero elements equal to 1.

Step 1: Auxiliary dynamic

First, we calculate the following auxiliary matrix: $d[i][j]$, nearest row that has a 1 above $a[i][j]$. Formally speaking, $d[i][j]$ is the largest row number (from 0 to $i - 1$), in which there is a element equal to 1 in the j -th column. While iterating from top-left to bottom-right, when we stand in row i , we know the values from the previous row, so, it is enough to update just the elements with value 1. We can save the values in a simple array $d[i]$, $i = 1 \dots m - 1$, because in the further algorithm we will process the matrix one row at a time and only need the values of the current row.

```
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;
        }
    }
}
```

Step 2: Problem solving

We can solve the problem in $O(nm^2)$ iterating through rows, considering every possible left and right columns for a submatrix. The bottom of the rectangle will be the current row, and using $d[i][j]$ we can find the top row. However, it is possible to go further and significantly improve the complexity of the solution.

It is clear that the desired zero submatrix is bounded on all four sides by some ones, which prevent it from increasing in size and improving the answer. Therefore, we will not miss the answer if we act as follows: for every cell j in row i (the bottom row of a potential zero submatrix) we will have $d[i][j]$ as the top row of the current zero submatrix. It now remains to determine the optimal left and right boundaries of the zero submatrix, i.e. maximally push this submatrix to the left and right of the j -th column.

What does it mean to push the maximum to the left? It means to find an index $k1$ for which $d[i][k1] > d[i][j]$, and at the same time $k1$ - the closest one to the left of the index j . It is clear that then $k1 + 1$ gives the number of the left column of the required zero submatrix. If there is no such index at

all, then put $k1 = -1$ (this means that we were able to extend the current zero submatrix to the left all the way to the border of matrix a).

Symmetrically, you can define an index $k2$ for the right border: this is the closest index to the right of j such that $d[i][k2] > d[i][j]$ (or m , if there is no such index).

So, the indices $k1$ and $k2$, if we learn to search for them effectively, will give us all the necessary information about the current zero submatrix. In particular, its area will be equal to $(i - d[i][j]) * (k2 - k1 - 1)$.

How to look for these indexes $k1$ and $k2$ effectively with fixed i and j ? We can do that in $O(1)$ on average.

To achieve such complexity, you can use the stack as follows. Let's first learn how to search for an index $k1$, and save its value for each index j within the current row i in matrix $d1[i][j]$. To do this, we will look through all the columns j from left to right, and we will store in the stack only those columns that have $d[j]$ strictly greater than $d[i][j]$. It is clear that when moving from a column j to the next column, it is necessary to update the content of the stack. When there is an inappropriate element at the top of the stack (i.e. $d[j] \leq d[i][j]$) pop it. It is easy to understand that it is enough to remove from the stack only from its top, and from none of its other places (because the stack will contain an increasing d sequence of columns).

The value $d1[i][j]$ for each j will be equal to the value lying at that moment on top of the stack.

The dynamics $d2[i][j]$ for finding the indices $k2$ is considered similar, only you need to view the columns from right to left.

It is clear that since there are exactly m pieces added to the stack on each line, there could not be more deletions either, the sum of complexities will be linear, so the final complexity of the algorithm is $O(nm)$.

It should also be noted that this algorithm consumes $O(m)$ memory (not counting the input data - the matrix $a[j]$).

Implementation

```
int zero_matrix(vector<vector<int>> a) {
    int n = a.size();
    int m = a[0].size();

    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;
        }

        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);
    }
    while (!st.empty())
        st.pop();

    for (int j = 0; j < m; ++j)
        ans = max(ans, (i - d[j]) * (d2[j] - d1[j] - 1));
    }
    return ans;
}
```

Part IV

String Processing

Chapter 1

Fundamentals

1.1 String Hashing

Hashing algorithms are helpful in solving a lot of problems.

The problem we want to solve is the problem, we want to compare strings efficiently. The brute force way of doing so is just to compare the letters of both strings, which has a time complexity of $O(\min(n_1, n_2))$ if n_1 and n_2 are the sizes of the two strings. We want to do better. The idea behind strings is the following: we convert each string into an integer, and compare those instead of the strings. Comparing two strings is then an $O(1)$ operation.

For the conversion we need a so-called **hash function**. The goal of it is to convert a string into a integer, the so-called **hash** of the string. The following condition has to hold: if two two strings s and t are equal ($s = t$), then also their hashes have to be equal ($\text{hash}(s) = \text{hash}(t)$). Otherwise we will not be able to compare strings.

Notice, the opposite direction doesn't have to hold. If the hashes are equal ($\text{hash}(s) = \text{hash}(t)$), then the strings do not necessarily have to be equal. E.g. a valid hash function would be simply $\text{hash}(s) = 0$ for each s . Now, this is just a stupid example, because this function will be completely useless, but it is a valid hash function. The reason why the opposite direction doesn't have to hold, is because there are exponential many strings. If we only want this hash function to distinguish between all strings consisting of lowercase characters of length smaller than 15, then already the hash wouldn't fit into a 64 bit integer (e.g. unsigned long long) any more, because there are so many of them. And of course we don't want to compare arbitrary long integers, because this will also have the complexity $O(n)$.

So usually we want the hash function to map strings onto numbers of a fixed range $[0, m)$, then comparing strings is just comparison of two integers with fixed length. And of course we want $\text{hash}(s) \neq \text{hash}(t)$ to be very likely, if $s \neq t$.

That's the important part that you have to keep in mind. Using hashing will not be 100% deterministically correct, because two complete different strings might have the same hash (the hashes collide). However, in a wide majority of tasks this can be safely ignored as the probability of the hashes of two different strings colliding is still very small. And we will discuss some techniques in this

article how to keep the probability of collisions very low.

1.1.1 Calculation of the hash of a string

The good and widely used way to define the hash of a string s of length n is

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \dots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

where p and m are chosen, positive numbers. It is called a **polynomial rolling hash function**.

It is reasonable to make p a prime number roughly equal to the number of characters in the input alphabet. For example, if the input is composed of only lowercase letters of English alphabet, $p = 31$ is a good choice. If the input may contain both uppercase and lowercase letters, then $p = 53$ is a possible choice. The code in this article will use $p = 31$.

Obviously m should be a large number, since the probability of two random strings colliding is about $\approx \frac{1}{m}$. Sometimes $m = 2^{64}$ is chosen, since then the integer overflows of 64 bit integers work exactly like the modulo operation. However there exists a method, which generates colliding strings (which work independent from the choice of p). So in practice $m = 2^{64}$ is not recommended. A good choice for m is some large prime number. The code in this article will just use $m = 10^9 + 9$. This is a large number, but still small enough so that we can perform multiplication of two values using 64 bit integers.

Here is an example of calculating the hash of a string s , which contains only lowercase letters. We convert each character of s to an integer. Here we use the conversion $a \rightarrow 1, b \rightarrow 2, \dots, z \rightarrow 26$. Converting $a \rightarrow 0$ is not a good idea, because then the hashes of the strings a, aa, aaa, \dots all evaluate to 0.

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Precomputing the powers of p might give a performance boost.

1.1.2 Example tasks

Search for duplicate strings in an array of strings

Problem: Given a list of n strings s_i , each no longer than m characters, find all the duplicate strings and divide them into groups.

From the obvious algorithm involving sorting the strings, we would get a time complexity of $O(nm \log n)$ where the sorting requires $O(n \log n)$ comparisons and each comparison take $O(m)$ time. However by using hashes, we reduce the comparison time to $O(1)$, giving us an algorithm that runs in $O(nm + n \log n)$ time.

We calculate the hash for each string, sort the hashes together with the indices, and then group the indices by identical hashes.

```
vector<vector<int>> group_identical_strings(vector<string> const& s) {
    int n = s.size();
    vector<pair<long long, int>> hashes(n);
    for (int i = 0; i < n; i++)
        hashes[i] = {compute_hash(s[i]), i};

    sort(hashes.begin(), hashes.end());

    vector<vector<int>> groups;
    for (int i = 0; i < n; i++) {
        if (i == 0 || hashes[i].first != hashes[i-1].first)
            groups.emplace_back();
        groups.back().push_back(hashes[i].second);
    }
    return groups;
}
```

Fast hash calculation of substrings of given string

Problem: Given a string s and indices i and j , find the hash of the substring $s[i \dots j]$.

By definition, we have:

$$\text{hash}(s[i \dots j]) = \sum_{k=i}^j s[k] \cdot p^{k-i} \mod m$$

Multiplying by p^i gives:

$$\begin{aligned} \text{hash}(s[i \dots j]) \cdot p^i &= \sum_{k=i}^j s[k] \cdot p^k \mod m \\ &= \text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1]) \mod m \end{aligned}$$

So by knowing the hash value of each prefix of the string s , we can compute the hash of any substring directly using this formula. The only problem that we face in calculating it is that we must be able to divide $\text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1])$ by p^i . Therefore we need to find the [modular multiplicative inverse](#) of p^i and then perform multiplication with this inverse. We can precompute the inverse of every p^i , which allows computing the hash of any substring of s in $O(1)$ time.

However, there does exist an easier way. In most cases, rather than calculating the hashes of substring exactly, it is enough to compute the hash multiplied by

some power of p . Suppose we have two hashes of two substrings, one multiplied by p^i and the other by p^j . If $i < j$ then we multiply the first hash by p^{j-i} , otherwise we multiply the second hash by p^{i-j} . By doing this, we get both the hashes multiplied by the same power of p (which is the maximum of i and j) and now these hashes can be compared easily with no need for any division.

1.1.3 Applications of Hashing

Here are some typical applications of Hashing:

- [Rabin-Karp algorithm](#) for pattern matching in a string in $O(n)$ time
- Calculating the number of different substrings of a string in $O(n^2 \log n)$ (see below)
- Calculating the number of palindromic substrings in a string.

Determine the number of different substrings in a string

Problem: Given a string s of length n , consisting only of lowercase English letters, find the number of different substrings in this string.

To solve this problem, we iterate over all substring lengths $l = 1 \dots n$. For every substring length l we construct an array of hashes of all substrings of length l multiplied by the same power of p . The number of different elements in the array is equal to the number of distinct substrings of length l in the string. This number is added to the final answer.

For convenience we will use $h[i]$ as the hash of the prefix with i characters, and define $h[0] = 0$.

```
int count_unique_substrings(string const& s) {
    int n = s.size();

    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(n + 1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (s[i] - 'a' + 1) * p_pow[i]) % m;

    int cnt = 0;
    for (int l = 1; l <= n; l++) {
        set<long long> hs;
        for (int i = 0; i <= n - l; i++) {
            long long cur_h = (h[i + l] + m - h[i]) % m;
            cur_h = (cur_h * p_pow[n-i-1]) % m;
            hs.insert(cur_h);
        }
        cnt += hs.size();
    }
}
```



```
    }  
    return cnt;  
}
```

1.1.4 Improve no-collision probability

Quite often the above mentioned polynomial hash is good enough, and no collisions will happen during tests. Remember, the probability that collision happens is only $\approx \frac{1}{m}$. For $m = 10^9 + 9$ the probability is $\approx 10^{-9}$ which is quite low. But notice, that we only did one comparison. What if we compared a string s with 10^6 different strings. The probability that the at least one collision happens is now $\approx 10^{-3}$. And if we want to compare 10^6 different strings with each other (e.g. by counting how many unique strings exists), then the probability of at least one collision happening is already ≈ 1 . It is pretty much guaranteed that this task will end with a collision and returns the wrong result.

There is a really easy trick to get better probabilities. We can just compute two different hashes for each string (by using two different p , and/or different m , and compare these pairs instead. If m is about 10^9 for each of the two hash functions, than this is more or less equivalent as having one hash function with $m \approx 10^{18}$. When comparing 10^6 strings with each other, the probability that at least one collision happens is now reduced to $\approx 10^{-6}$.

1.1.5 Practice Problems

- [A Needle in the Haystack - SPOJ](#)
- [Double Profiles - Codeforces](#)
- [Password - Codeforces](#)
- [SUB_PROB - SPOJ](#)
- [INSQ15_A](#)
- [SPOJ - Ada and Spring Cleaning](#)
- [GYM - Text Editor](#)
- [12012 - Detection of Extraterrestrial](#)
- [Codeforces - Games on a CD](#)
- [UVA 11855 - Buzzwords](#)
- [Codeforces - Santa Claus and a Palindrome](#)
- [Codeforces - String Compression](#)
- [Codeforces - Palindromic Characteristics](#)
- [SPOJ - Test](#)
- [Codeforces - Palindrome Degree](#)
- [Codeforces - Deletion of Repeats](#)

1.2 Rabin-Karp Algorithm for string matching

This algorithm is based on the concept of hashing, so if you are not familiar with string hashing, refer to the [string hashing](#) article.

This algorithm was authored by Rabin and Karp in 1987.

Problem: Given two strings - a pattern s and a text t , determine if the pattern appears in the text and if it does, enumerate all its occurrences in $O(|s| + |t|)$ time.

Algorithm: Calculate the hash for the pattern s . Calculate hash values for all the prefixes of the text t . Now, we can compare a substring of length $|s|$ with s in constant time using the calculated hashes. So, compare each substring of length $|s|$ with the pattern. This will take a total of $O(|t|)$ time. Hence the final complexity of the algorithm is $O(|t| + |s|)$: $O(|s|)$ is required for calculating the hash of the pattern and $O(|t|)$ for comparing each substring of length $|s|$ with the pattern.

1.2.1 Implementation

```
vector<int> rabin_karp(string const& s, string const& t) {
    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();

    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;
    for (int i = 1; i < (int)p_pow.size(); i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(T + 1, 0);
    for (int i = 0; i < T; i++)
        h[i+1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % m;
    long long h_s = 0;
    for (int i = 0; i < S; i++)
        h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;

    vector<int> occurrences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur_h = (h[i+S] + m - h[i]) % m;
        if (cur_h == h_s * p_pow[i] % m)
            occurrences.push_back(i);
    }
    return occurrences;
}
```

1.2.2 Practice Problems

- [SPOJ - Pattern Find](#)
- [Codeforces - Good Substrings](#)
- [Codeforces - Palindromic characteristics](#)

1.3 Prefix function. Knuth–Morris–Pratt algorithm

1.3.1 Prefix function definition

You are given a string s of length n . The **prefix function** for this string is defined as an array π of length n , where $\pi[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$ which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $\pi[0] = 0$.

Mathematically the definition of the prefix function can be written as follows:

$$\pi[i] = \max_{k=0 \dots i} \{k : s[0 \dots k-1] = s[i-(k-1) \dots i]\}$$

For example, prefix function of string “abcabed” is $[0, 0, 0, 1, 2, 3, 0]$, and prefix function of string “aabaaab” is $[0, 1, 0, 1, 2, 2, 3]$.

1.3.2 Trivial Algorithm

An algorithm which follows the definition of prefix function exactly is the following:

```
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;
}
```

It is easy to see that its complexity is $O(n^3)$, which has room for improvement.

1.3.3 Efficient Algorithm

This algorithm was proposed by Knuth and Pratt and independently from them by Morris in 1977. It was used as the main function of a substring search algorithm.

First optimization

The first important observation is, that the values of the prefix function can only increase by at most one.

Indeed, otherwise, if $\pi[i+1] > \pi[i] + 1$, then we can take this suffix ending in position $i+1$ with the length $\pi[i+1]$ and remove the last character from it. We end up with a suffix ending in position i with the length $\pi[i+1] - 1$, which is better than $\pi[i]$, i.e. we get a contradiction.

The following illustration shows this contradiction. The longest proper suffix at position i that also is a prefix is of length 2, and at position $i+1$ it is of length 4. Therefore the string $s_0 s_1 s_2 s_3$ is equal to the string $s_{i-2} s_{i-1} s_i s_{i+1}$, which

means that also the strings $s_0 s_1 s_2$ and $s_{i-2} s_{i-1} s_i$ are equal, therefore $\pi[i]$ has to be 3.

$$\begin{array}{ccccccc} \overbrace{s_0 s_1}^{\pi[i]=2} & s_2 & s_3 & \dots & s_{i-2} & \overbrace{s_{i-1} s_i}^{\pi[i]=2} & s_{i+1} \\ \underbrace{\hspace{1.5cm}}_{\pi[i+1]=4} & & & & & \underbrace{\hspace{1.5cm}}_{\pi[i+1]=4} & \end{array}$$

Thus when moving to the next position, the value of the prefix function can either increase by one, stay the same, or decrease by some amount. This fact already allows us to reduce the complexity of the algorithm to $O(n^2)$, because in one step the prefix function can grow at most by one. In total the function can grow at most n steps, and therefore also only can decrease a total of n steps. This means we only have to perform $O(n)$ string comparisons, and reach the complexity $O(n^2)$.

Second optimization

Let's go further, we want to get rid of the string comparisons. To accomplish this, we have to use all the information computed in the previous steps.

So let us compute the value of the prefix function π for $i+1$. If $s[i+1] = s[\pi[i]]$, then we can say with certainty that $\pi[i+1] = \pi[i] + 1$, since we already know that the suffix at position i of length $\pi[i]$ is equal to the prefix of length $\pi[i]$. This is illustrated again with an example.

$$\begin{array}{ccccccc} \overbrace{s_0 s_1 s_2}^{\pi[i]} & \overbrace{s_3}^{s_3=s_{i+1}} & \dots & \overbrace{s_{i-2} s_{i-1} s_i}^{\pi[i]} & \overbrace{s_{i+1}}^{s_3=s_{i+1}} \\ \underbrace{\hspace{1.5cm}}_{\pi[i+1]=\pi[i]+1} & & & \underbrace{\hspace{1.5cm}}_{\pi[i+1]=\pi[i]+1} & \end{array}$$

If this is not the case, $s[i+1] \neq s[\pi[i]]$, then we need to try a shorter string. In order to speed things up, we would like to immediately move to the longest length $j < \pi[i]$, such that the prefix property in the position i holds, i.e. $s[0 \dots j-1] = s[i-j+1 \dots i]$:

$$\begin{array}{ccccccc} \overbrace{s_0 s_1 \dots j}^{\pi[i]} & s_2 & s_3 & \dots & s_{i-3} & s_{i-2} & \overbrace{s_{i-1} s_i \dots j}^{\pi[i]} & s_{i+1} \\ \underbrace{\hspace{1.5cm}} & & & & & & \underbrace{\hspace{1.5cm}} & \end{array}$$

Indeed, if we find such a length j , then we again only need to compare the characters $s[i+1]$ and $s[j]$. If they are equal, then we can assign $\pi[i+1] = j+1$. Otherwise we will need to find the largest value smaller than j , for which the prefix property holds, and so on. It can happen that this goes until $j = 0$. If then $s[i+1] = s[0]$, we assign $\pi[i+1] = 1$, and $\pi[i+1] = 0$ otherwise.

So we already have a general scheme of the algorithm. The only question left is how do we effectively find the lengths for j . Let's recap: for the current length j at the position i for which the prefix property holds, i.e. $s[0 \dots j-1] = s[i-j+1 \dots i]$, we want to find the greatest $k < j$, for which the prefix property holds.

$$\begin{array}{ccccccc} \overbrace{s_0 s_1 \dots k}^j & s_2 & s_3 & \dots & s_{i-3} & s_{i-2} & \overbrace{s_{i-1} s_i \dots k}^j & s_{i+1} \\ \underbrace{\hspace{1.5cm}} & & & & & & \underbrace{\hspace{1.5cm}} & \end{array}$$

The illustration shows, that this has to be the value of $\pi[j-1]$, which we already calculated earlier.

Final algorithm

So we finally can build an algorithm that doesn't perform any string comparisons and only performs $O(n)$ actions.

Here is the final procedure:

- We compute the prefix values $\pi[i]$ in a loop by iterating from $i = 1$ to $i = n - 1$ ($\pi[0]$ just gets assigned with 0).
- To calculate the current value $\pi[i]$ we set the variable j denoting the length of the best suffix for $i - 1$. Initially $j = \pi[i - 1]$.
- Test if the suffix of length $j + 1$ is also a prefix by comparing $s[j]$ and $s[i]$. If they are equal then we assign $\pi[i] = j + 1$, otherwise we reduce j to $\pi[j - 1]$ and repeat this step.
- If we have reached the length $j = 0$ and still don't have a match, then we assign $\pi[i] = 0$ and go to the next index $i + 1$.

Implementation

The implementation ends up being surprisingly short and expressive.

```
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;
}
```

This is an **online** algorithm, i.e. it processes the data as it arrives - for example, you can read the string characters one by one and process them immediately, finding the value of prefix function for each next character. The algorithm still requires storing the string itself and the previously calculated values of prefix function, but if we know beforehand the maximum value M the prefix function can take on the string, we can store only $M + 1$ first characters of the string and the same number of values of the prefix function.

1.3.4 Applications

Search for a substring in a string. The Knuth-Morris-Pratt algorithm

The task is the classical application of the prefix function.

Given a text t and a string s , we want to find and display the positions of all occurrences of the string s in the text t .

For convenience we denote with n the length of the string s and with m the length of the text t .

We generate the string $s + \# + t$, where $\#$ is a separator that appear in s and neither in t . Let us calculate the prefix function for this string. Now think about the meaning of the values of the prefix function, except for the first $n + 1$ entries (which belong to the string s and the separator). By definition the value $\pi[i]$ shows the longest length of a substring ending in position i that coincides with the prefix. But in our case this is nothing more than the largest block that coincides with s and ends at position i . This length cannot be bigger than n due to the separator. But if equality $\pi[i] = n$ is achieved, then it means that the string s appears completely in at this position, i.e. it ends at position i . Just do not forget that the positions are indexed in the string $s + \# + t$.

Thus if at some position i we have $\pi[i] = n$, then at the position $i - (n + 1) - n + 1 = i - 2n$ in the string t the string s appears.

As already mentioned in the description of the prefix function computation, if we know that the prefix values never exceed a certain value, then we do not need to store the entire string and the entire function, but only its beginning. In our case this means that we only need to store the string $s + \#$ and the values of the prefix function for it. We can read one character at a time of the string t and calculate the current value of the prefix function.

Thus the Knuth-Morris-Pratt algorithm solves the problem in $O(n + m)$ time and $O(n)$ memory.

Counting the number of occurrences of each prefix

Here we discuss two problems at once. Given a string s of length n . In the first variation of the problem we want to count the number of appearances of each prefix $s[0 \dots i]$ in the same string. In the second variation of the problem another string t is given and we want to count the number of appearances of each prefix $s[0 \dots i]$ in t .

First we solve the first problem. Consider the value of the prefix function $\pi[i]$ at a position i . By definition it means that in position i the prefix of length $\pi[i]$ of the string s appears and ends at position i , and there doesn't exists a longer prefix that follows this definition. At the same time shorter prefixes can end at this position. It is not difficult to see, that we have the same question that we already answered when we computed the prefix function itself: Given a prefix of length j that is a suffix ending at position i , what is the next smaller prefix $< j$ that is also a suffix ending at position i . Thus at the position i ends the prefix of length $\pi[i]$, the prefix of length $\pi[\pi[i] - 1]$, the prefix $\pi[\pi[\pi[i] - 1] - 1]$, and so on, until the index becomes zero. Thus we can compute the answer in the following way.

```
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];
```

```
for (int i = 0; i <= n; i++)
    ans[i]++;
```

Here for each value of the prefix function we first count how many times it occurs in the array π , and then compute the final answers: if we know that the length prefix i appears exactly $\text{ans}[i]$ times, then this number must be added to the number of occurrences of its longest suffix that is also a prefix. At the end we need to add 1 to each result, since we also need to count the original prefixes also.

Now let us consider the second problem. We apply the trick from Knuth-Morris-Pratt: we create the string $s + \# + t$ and compute its prefix function. The only differences to the first task is, that we are only interested in the prefix values that relate to the string t , i.e. $\pi[i]$ for $i \geq n + 1$. With those values we can perform the exact same computations as in the first task.

The number of different substring in a string

Given a string s of length n . We want to compute the number of different substrings appearing in it.

We will solve this problem iteratively. Namely we will learn, knowing the current number of different substrings, how to recompute this count by adding a character to the end.

So let k be the current number of different substrings in s , and we add the character c to the end of s . Obviously some new substrings ending in c will appear. We want to count these new substrings that didn't appear before.

We take the string $t = s + c$ and reverse it. Now the task is transformed into computing how many prefixes there are that don't appear anywhere else. If we compute the maximal value of the prefix function π_{\max} of the reversed string t , then the longest prefix that appears in s is π_{\max} long. Clearly also all prefixes of smaller length appear in it.

Therefore the number of new substrings appearing when we add a new character c is $|s| + 1 - \pi_{\max}$.

So for each character appended we can compute the number of new substrings in $O(n)$ times, which gives a time complexity of $O(n^2)$ in total.

It is worth noting, that we can also compute the number of different substrings by appending the characters at the beginning, or by deleting characters from the beginning or the end.

Compressing a string

Given a string s of length n . We want to find the shortest “compressed” representation of the string, i.e. we want to find a string t of smallest length such that s can be represented as a concatenation of one or more copies of t .

It is clear, that we only need to find the length of t . Knowing the length, the answer to the problem will be the prefix of s with this length.

Let us compute the prefix function for s . Using the last value of it we define the value $k = n - \pi[n - 1]$. We will show, that if k divides n , then k will be the

answer, otherwise there doesn't exist an effective compression and the answer is n .

Let n be divisible by k . Then the string can be partitioned into blocks of the length k . By definition of the prefix function, the prefix of length $n - k$ will be equal with its suffix. But this means that the last block is equal to the block before. And the block before has to be equal to the block before it. And so on. As a result, it turns out that all blocks are equal, therefore we can compress the string s to length k .

Of course we still need to show that this is actually the optimum. Indeed, if there was a smaller compression than k , then the prefix function at the end would be greater than $n - k$. Therefore k has really the answer.

Now let us assume that n is not divisible by k . We show that this implies that the length of the answer is n . We prove it by contradiction. Assuming there exists an answer, and the compression has length p (p divides n). Then the last value of the prefix function has to be greater than $n - p$, i.e. the suffix will partially cover the first block. Now consider the second block of the string. Since the prefix is equal with the suffix, and both the prefix and the suffix cover this block and their displacement relative to each other k does not divide the block length p (otherwise k divides n), then all the characters of the block have to be identical. But then the string consists of only one character repeated over and over, hence we can compress it to a string of size 1, which gives $k = 1$, and k divides n . Contradiction.

$$\begin{array}{ccccccc}
 & \overbrace{s_0 \ s_1 \ s_2 \ s_3}^p & & \overbrace{s_4 \ s_5 \ s_6 \ s_7}^p & & & \\
 & s_0 \ s_1 \ s_2 & & \overbrace{s_3 \ s_4 \ s_5 \ s_6 \ s_7}^p & & & \\
 & & & \pi[7]=5 & & & \\
 s_4 = s_3, \ s_5 = s_4, \ s_6 = s_5, \ s_7 = s_6 & \Rightarrow & s_0 = s_1 = s_2 = s_3
 \end{array}$$

Building an automaton according to the prefix function

Let's return to the concatenation of the two strings through a separator, i.e. for the strings s and t we compute the prefix function for the string $s + \# + t$. Obviously, since $\#$ is a separator, the value of the prefix function will never exceed $|s|$. It follows, that it is sufficient to only store the string $s + \#$ and the values of the prefix function for it, and we can compute the prefix function for all subsequent character on the fly:

$$\underbrace{s_0 \ s_1 \ \dots \ s_{n-1} \ \#}_{\text{need to store}} \quad \underbrace{t_0 \ t_1 \ \dots \ t_{m-1}}_{\text{do not need to store}}$$

Indeed, in such a situation, knowing the next character $c \in t$ and the value of the prefix function of the previous position is enough information to compute the next value of the prefix function, without using any previous characters of the string t and the value of the prefix function in them.

In other words, we can construct an **automaton** (a finite state machine): the state in it is the current value of the prefix function, and the transition from one state to another will be performed via the next character.

Thus, even without having the string t , we can construct such a transition table ($\text{old_}\pi, c \rightarrow \text{new_}\pi$) using the same algorithm as for calculating the transition table:

```
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            int j = i;
            while (j > 0 && 'a' + c != s[j])
                j = pi[j-1];
            if ('a' + c == s[j])
                j++;
            aut[i][c] = j;
        }
    }
}
```

However in this form the algorithm runs in $O(n^2 26)$ time for the lowercase letters of the alphabet. Note that we can apply dynamic programming and use the already calculated parts of the table. Whenever we go from the value j to the value $\pi[j-1]$, we actually mean that the transition (j, c) leads to the same state as the transition as $(\pi[j-1], c)$, and this answer is already accurately computed.

```
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            if (i > 0 && 'a' + c != s[i])
                aut[i][c] = aut[pi[i-1]][c];
            else
                aut[i][c] = i + ('a' + c == s[i]);
        }
    }
}
```

As a result we construct the automaton in $O(n 26)$ time.

When is such a automaton useful? To begin with, remember that we use the prefix function for the string $s + \# + t$ and its values mostly for a single purpose: find all occurrences of the string s in the string t .

Therefore the most obvious benefit of this automaton is the **acceleration of calculating the prefix function** for the string $s + \# + t$. By building the

automaton for $s + \#$, we no longer need to store the string s or the values of the prefix function in it. All transitions are already computed in the table.

But there is a second, less obvious, application. We can use the automaton when the string t is a **gigantic string constructed using some rules**. This can for instance be the Gray strings, or a string formed by a recursive combination of several short strings from the input.

For completeness we will solve such a problem: given a number $k \leq 10^5$ and a string s of length $\leq 10^5$. We have to compute the number of occurrences of s in the k -th Gray string. Recall that Gray's strings are defined in the following way:

$$\begin{aligned} g_1 &= "a" \\ g_2 &= "aba" \\ g_3 &= "abacaba" \\ g_4 &= "abacabadabacaba" \end{aligned}$$

In such cases even constructing the string t will be impossible, because of its astronomical length. The k -th Gray string is $2^k - 1$ characters long. However we can calculate the value of the prefix function at the end of the string effectively, by only knowing the value of the prefix function at the start.

In addition to the automaton itself, we also compute values $G[i][j]$ - the value of the automaton after processing the string g_i starting with the state j . And additionally we compute values $K[i][j]$ - the number of occurrences of s in g_i , before during the processing of g_i starting with the state j . Actually $K[i][j]$ is the number of times that the prefix function took the value $|s|$ while performing the operations. The answer to the problem will then be $K[k][0]$.

How can we compute these values? First the basic values are $G[0][j] = j$ and $K[0][j] = 0$. And all subsequent values can be calculated from the previous values and using the automaton. To calculate the value for some i we remember that the string g_i consists of g_{i-1} , the i character of the alphabet, and g_{i-1} . Thus the automaton will go into the state:

$$\text{mid} = \text{aut}[G[i-1][j]][i]$$

$$G[i][j] = G[i-1][\text{mid}]$$

The values for $K[i][j]$ can also be easily counted.

$$K[i][j] = K[i-1][j] + (\text{mid} == |s|) + K[i-1][\text{mid}]$$

So we can solve the problem for Gray strings, and similarly also a huge number of other similar problems. For example the exact same method also solves the following problem: we are given a string s and some patterns t_i , each of which is specified as follows: it is a string of ordinary characters, and there might be some recursive insertions of the previous strings of the form t_k^{cnt} , which means that at

this place we have to insert the string t_k cnt times. An example of such patterns:

$$\begin{aligned}t_1 &= "abdeca" \\t_2 &= "abc" + t_1^{30} + "abd" \\t_3 &= t_2^{50} + t_1^{100} \\t_4 &= t_2^{10} + t_3^{100}\end{aligned}$$

The recursive substitutions blow the string up, so that their lengths can reach the order of 100^{100} .

We have to find the number of times the string s appears in each of the strings.

The problem can be solved in the same way by constructing the automaton of the prefix function, and then we calculate the transitions in for each pattern by using the previous results.

1.3.5 Practice Problems

- [UVA # 455 "Periodic Strings"](#)
- [UVA # 11022 "String Factoring"](#)
- [UVA # 11452 "Dancing the Cheeky-Cheeky"](#)
- [UVA 12604 - Caesar Cipher](#)
- [UVA 12467 - Secret Word](#)
- [UVA 11019 - Matrix Matcher](#)
- [SPOJ - Pattern Find](#)
- [Codeforces - Anthem of Berland](#)
- [Codeforces - MUH and Cube Walls](#)

1.4 Z-function and its calculation

Suppose we are given a string s of length n . The **Z-function** for this string is an array of length n where the i -th element is equal to the greatest number of characters starting from the position i that coincide with the first characters of s .

In other words, $z[i]$ is the length of the longest common prefix between s and the suffix of s starting at i .

Note. In this article, to avoid ambiguity, we assume 0-based indexes; that is: the first character of s has index 0 and the last one has index $n - 1$.

The first element of Z-function, $z[0]$, is generally not well defined. In this article we will assume it is zero (although it doesn't change anything in the algorithm implementation).

This article presents an algorithm for calculating the Z-function in $O(n)$ time, as well as various of its applications.

1.4.1 Examples

For example, here are the values of the Z-function computed for different strings:

- “aaaaa” - [0, 4, 3, 2, 1]
- “aaabaab” - [0, 2, 1, 0, 2, 1, 0]
- “abacaba” - [0, 0, 1, 0, 3, 0, 1]

1.4.2 Trivial algorithm

Formal definition can be represented in the following elementary $O(n^2)$ implementation.

```
vector<int> z_function_trivial(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1; i < n; ++i)
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
    return z;
}
```

We just iterate through every position i and update $z[i]$ for each one of them, starting from $z[i] = 0$ and incrementing it as long as we don't find a mismatch (and as long as we don't reach the end of the line).

Of course, this is not an efficient implementation. We will now show the construction of an efficient implementation.

1.4.3 Efficient algorithm to compute the Z-function

To obtain an efficient algorithm we will compute the values of $z[i]$ in turn from $i = 1$ to $n - 1$ but at the same time, when computing a new value, we'll try to make the best use possible of the previously computed values.

For the sake of brevity, let's call **segment matches** those substrings that coincide with a prefix of s . For example, the value of the desired Z-function $z[i]$ is the length of the segment match starting at position i (and that ends at position $i + z[i] - 1$).

To do this, we will keep the $[l; r]$ **indices of the rightmost segment match**. That is, among all detected segments we will keep the one that ends rightmost. In a way, the index r can be seen as the “boundary” to which our string s has been scanned by the algorithm; everything beyond that point is not yet known.

Then, if the current index (for which we have to compute the next value of the Z-function) is i , we have one of two options:

- $i > r$ – the current position is **outside** of what we have already processed. We will then compute $z[i]$ with the **trivial algorithm** (that is, just comparing values one by one). Note that in the end, if $z[i] > 0$, we'll have to update the indices of the rightmost segment, because it's guaranteed that the new $r = i + z[i] - 1$ is better than the previous r .
- $i \leq r$ – the current position is inside the current segment match $[l; r]$.

Then we can use the already calculated Z-values to “initialize” the value of $z[i]$ to something (it sure is better than “starting from zero”), maybe even some big number.

For this, we observe that the substrings $s[l \dots r]$ and $s[0 \dots r - l]$ **match**. This means that as an initial approximation for $z[i]$ we can take the value already computed for the corresponding segment $s[0 \dots r - l]$, and that is $z[i - l]$.

However, the value $z[i - l]$ could be too large: when applied to position i it could exceed the index r . This is not allowed because we know nothing about the characters to the right of r : they may differ from those required.

Here is **an example** of a similar scenario:

$$s = \text{"aaaabaa"}$$

When we get to the last position ($i = 6$), the current match segment will be $[5; 6]$. Position 6 will then match position $6 - 5 = 1$, for which the value of the Z-function is $z[1] = 3$. Obviously, we cannot initialize $z[6]$ to 3, it would be completely incorrect. The maximum value we could initialize it to is 1 – because it's the largest value that doesn't bring us beyond the index r of the match segment $[l; r]$.

Thus, as an **initial approximation** for $z[i]$ we can safely take:

$$z_0[i] = \min(r - i + 1, z[i - l])$$

After having $z[i]$ initialized to $z_0[i]$, we try to increment $z[i]$ by running the **trivial algorithm** – because in general, after the border r , we cannot know if the segment will continue to match or not.

Thus, the whole algorithm is split in two cases, which differ only in **the initial value** of $z[i]$: in the first case it's assumed to be zero, in the second case it is determined by the previously computed values (using the above formula). After that, both branches of this algorithm can be reduced to the implementation of **the trivial algorithm**, which starts immediately after we specify the initial value.

The algorithm turns out to be very simple. Despite the fact that on each iteration the trivial algorithm is run, we have made significant progress, having an algorithm that runs in linear time. Later on we will prove that the running time is linear.

1.4.4 Implementation

Implementation turns out to be rather laconic:

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        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;
}
```

Comments on this implementation

The whole solution is given as a function which returns an array of length n – the Z-function of s .

Array z is initially filled with zeros. The current rightmost match segment is assumed to be $[0; 0]$ (that is, a deliberately small segment which doesn't contain any i).

Inside the loop for $i = 1 \dots n - 1$ we first determine the initial value $z[i]$ – it will either remain zero or be computed using the above formula.

Thereafter, the trivial algorithm attempts to increase the value of $z[i]$ as much as possible.

In the end, if it's required (that is, if $i + z[i] - 1 > r$), we update the rightmost match segment $[l; r]$.

1.4.5 Asymptotic behavior of the algorithm

We will prove that the above algorithm has a running time that is linear in the length of the string – thus, it's $O(n)$.

The proof is very simple.

We are interested in the nested **while** loop, since everything else is just a bunch of constant operations which sums up to $O(n)$.

We will show that **each iteration** of the **while** loop will increase the right border r of the match segment.

To do that, we will consider both branches of the algorithm:

- $i > r$

In this case, either the **while** loop won't make any iteration (if $s[0] \neq s[i]$), or it will take a few iterations, starting at position i , each time moving one character to the right. After that, the right border r will necessarily be updated.

So we have found that, when $i > r$, each iteration of the **while** loop increases the value of the new r index.

- $i \leq r$

In this case, we initialize $z[i]$ to a certain value z_0 given by the above formula. Let's compare this initial value z_0 to the value $r - i + 1$. We will have three cases:

- $z_0 < r - i + 1$

We prove that in this case no iteration of the **while** loop will take place.

It's easy to prove, for example, by contradiction: if the **while** loop made at least one iteration, it would mean that initial approximation $z[i] = z_0$ was inaccurate (less than the match's actual length). But since $s[l \dots r]$ and $s[0 \dots r - l]$ are the same, this would imply that $z[i - l]$ holds the wrong value (less than it should be).

Thus, since $z[i - l]$ is correct and it is less than $r - i + 1$, it follows that this value coincides with the required value $z[i]$.

- $z_0 = r - i + 1$

In this case, the **while** loop can make a few iterations, but each of them will lead to an increase in the value of the r index because we will start comparing from $s[r + 1]$, which will climb beyond the $[l; r]$ interval.

- $z_0 > r - i + 1$

This option is impossible, by definition of z_0 .

So, we have proved that each iteration of the inner loop make the r pointer advance to the right. Since r can't be more than $n - 1$, this means that the inner loop won't make more than $n - 1$ iterations.

As the rest of the algorithm obviously works in $O(n)$, we have proved that the whole algorithm for computing Z-functions runs in linear time.

1.4.6 Applications

We will now consider some uses of Z-functions for specific tasks.

These applications will be largely similar to applications of [prefix function](#).

Search the substring

To avoid confusion, we call t the **string of text**, and p the **pattern**. The problem is: find all occurrences of the pattern p inside the text t .

To solve this problem, we create a new string $s = p + \diamond + t$, that is, we apply string concatenation to p and t but we also put a separator character \diamond in the middle (we'll choose \diamond so that it will certainly not be present anywhere in the strings p or t).

Compute the Z-function for s . Then, for any i in the interval $[0; \text{length}(t) - 1]$, we will consider the corresponding value $k = z[i + \text{length}(p) + 1]$. If k is equal to $\text{length}(p)$ then we know there is one occurrence of p in the i -th position of t , otherwise there is no occurrence of p in the i -th position of t .

The running time (and memory consumption) is $O(\text{length}(t) + \text{length}(p))$.

Number of distinct substrings in a string

Given a string s of length n , count the number of distinct substrings of s .

We'll solve this problem iteratively. That is: knowing the current number of different substrings, recalculate this amount after adding to the end of s one character.

So, let k be the current number of distinct substrings of s . We append a new character c to s . Obviously, there can be some new substrings ending in this new character c (namely, all those strings that end with this symbol and that we haven't encountered yet).

Take a string $t = s + c$ and invert it (write its characters in reverse order). Our task is now to count how many prefixes of t are not found anywhere else in t . Let's compute the Z-function of t and find its maximum value z_{max} . Obviously, t 's prefix of length z_{max} occurs also somewhere in the middle of t . Clearly, shorter prefixes also occur.

So, we have found that the number of new substrings that appear when symbol c is appended to s is equal to $\text{length}(t) - z_{max}$.

Consequently, the running time of this solution is $O(n^2)$ for a string of length n .

It's worth noting that in exactly the same way we can recalculate, still in $O(n)$ time, the number of distinct substrings when appending a character in the beginning of the string, as well as when removing it (from the end or the beginning).

String compression

Given a string s of length n . Find its shortest "compressed" representation, that is: find a string t of shortest length such that s can be represented as a concatenation of one or more copies of t .

A solution is: compute the Z-function of s , loop through all i such that i divides n . Stop at the first i such that $i + z[i] = n$. Then, the string s can be compressed to the length i .

The proof for this fact is the same as the solution which uses the [prefix function](#).

1.4.7 Practice Problems

- [Codeforces - Password](#) [Difficulty: Easy]
- [UVA # 455 “Periodic Strings”](#) [Difficulty: Medium]
- [UVA # 11022 “String Factoring”](#) [Difficulty: Medium]
- [UVa 11475 - Extend to Palindrome](#)
- [LA 6439 - Pasti Pas!](#)
- [Codechef - Chef and Strings](#)
- [Codeforces - Prefixes and Suffixes](#)

1.5 Suffix Array

1.5.1 Definition

Let s be a string of length n . The i -th suffix of s is the substring $s[i \dots n - 1]$.

A **suffix array** will contain integers that represent the **starting indexes** of the all the suffixes of a given string, after the aforementioned suffixes are sorted.

As an example look at the string $s = abaab$. All suffixes are as follows

0. *abaab*
1. *baab*
2. *aab*
3. *ab*
4. *b*

After sorting these strings:

2. *aab*
3. *ab*
0. *abaab*
4. *b*
1. *baab*

Therefore the suffix array for s will be (2, 3, 0, 4, 1).

As a data structure it is widely used in areas such as data compression, bioinformatics and, in general, in any area that deals with strings and string matching problems.

1.5.2 Construction

$O(n^2 \log n)$ approach

This is the most naive approach. Get all the suffixes and sort them using quicksort or mergesort and simultaneously retain their original indices. Sorting uses $O(n \log n)$ comparisons, and since comparing two strings will additionally take $O(n)$ time, we get the final complexity of $O(n^2 \log n)$.

$O(n \log n)$ approach

Strictly speaking the following algorithm will not sort the suffixes, but rather the cyclic shifts of a string. However we can very easily derive an algorithm for sorting suffixes from it: it is enough to append an arbitrary character to the end of the string which is smaller than any character from the string. It is common to use the symbol \$. Then the order of the sorted cyclic shifts is equivalent to the order of the sorted suffixes, as demonstrated here with the string *dabbb*.

1. *abbb\$d* *abbb*
4. *b\$dabb* *b*
3. *bb\$dab* *bb*
2. *bbb\$da* *bbb*
0. *dabbb\$* *dabbb*

Since we are going to sort cyclic shifts, we will consider **cyclic substrings**. We will use the notation $s[i \dots j]$ for the substring of s even if $i > j$. In this case we actually mean the string $s[i \dots n-1] + s[0 \dots j]$. In addition we will take all indices modulo the length of s , and will omit the modulo operation for simplicity.

The algorithm we discuss will perform $\lceil \log n \rceil + 1$ iterations. In the k -th iteration ($k = 0 \dots \lceil \log n \rceil$) we sort the n cyclic substrings of s of length 2^k . After the $\lceil \log n \rceil$ -th iteration the substrings of length $2^{\lceil \log n \rceil} \geq n$ will be sorted, so this is equivalent to sorting the cyclic shifts altogether.

In each iteration of the algorithm, in addition to the permutation $p[0 \dots n-1]$, where $p[i]$ is the index of the i -th substring (starting at i and with length 2^k) in the sorted order, we will also maintain an array $c[0 \dots n-1]$, where $c[i]$ corresponds to the **equivalence class** to which the substring belongs. Because some of the substrings will be identical, and the algorithm needs to treat them equally. For convenience the classes will be labeled by numbers started from zero. In addition the numbers $c[i]$ will be assigned in such a way that they preserve information about the order: if one substring is smaller than the other, then it should also have a smaller class label. The number of equivalence classes will be stored in a variable `classes`.

Let's look at an example. Consider the string $s = aaba$. The cyclic substrings and the corresponding arrays $p[]$ and $c[]$ are given for each iteration:

0 :	(a, a, b, a)	$p = (0, 1, 3, 2)$	$c = (0, 0, 1, 0)$
1 :	(aa, ab, ba, aa)	$p = (0, 3, 1, 2)$	$c = (0, 1, 2, 0)$
2 :	(aaba, abaa, baaa, aaab)	$p = (3, 0, 1, 2)$	$c = (1, 2, 3, 0)$

It is worth noting that the values of $p[]$ can be different. For example in the 0-th iteration the array could also be $p = (3, 1, 0, 2)$ or $p = (3, 0, 1, 2)$. All these options permutation the substrings into a sorted order. So they are all valid. At the same time the array $c[]$ is fixed, there can be no ambiguities.

Let us now focus on the implementation of the algorithm. We will write a function that takes a string s and returns the permutations of the sorted cyclic shifts.

```
vector<int> sort_cyclic_shifts(string const& s) {
    int n = s.size();
    const int alphabet = 256;
```

At the beginning (in the **0-th iteration**) we must sort the cyclic substrings of length 1, that is we have to sort all characters of the string and divide them into equivalence classes (same symbols get assigned to the same class). This can be done trivially, for example, by using **counting sort**. For each character we count how many times it appears in the string, and then use this information to create the array $p[]$. After that we go through the array $p[]$ and construct $c[]$ by comparing adjacent characters.

```
vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
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;
}

```

Now we have to talk about the iteration step. Let's assume we have already performed the $k-1$ -th step and computed the values of the arrays $p[]$ and $c[]$ for it. We want to compute the values for the k -th step in $O(n)$ time. Since we perform this step $O(\log n)$ times, the complete algorithm will have a time complexity of $O(n \log n)$.

To do this, note that the cyclic substrings of length 2^k consists of two substrings of length 2^{k-1} which we can compare with each other in $O(1)$ using the information from the previous phase - the values of the equivalence classes $c[]$. Thus, for two substrings of length 2^k starting at position i and j , all necessary information to compare them is contained in the pairs $(c[i], c[i + 2^{k-1}])$ and $(c[j], c[j + 2^{k-1}])$.

$$\begin{array}{ccccccc}
 \dots & \underbrace{s_i \dots s_{i+2^{k-1}-1}}_{\text{length}=2^{k-1}, \text{ class}=c[i]} & \underbrace{s_{i+2^{k-1}} \dots s_{i+2^k-1}}_{\text{length}=2^{k-1}, \text{ class}=c[i+2^{k-1}]} & \dots & \underbrace{s_j \dots s_{j+2^{k-1}-1}}_{\text{length}=2^{k-1}, \text{ class}=c[j]} & \underbrace{s_{j+2^{k-1}} \dots s_{j+2^k-1}}_{\text{length}=2^{k-1}, \text{ class}=c[j+2^{k-1}]} & \dots
 \end{array}$$

This gives us a very simple solution: **sort** the substrings of length 2^k **by these pairs of numbers**. This will give us the required order $p[]$. However a normal sort runs in $O(n \log n)$ time, with which we are not satisfied. This will only give us an algorithm for constructing a suffix array in $O(n \log^2 n)$ times.

How do we quickly perform such a sorting of the pairs? Since the elements of the pairs do not exceed n , we can use counting sort again. However sorting pairs with counting sort is not the most efficient. To achieve a better hidden constant in the complexity, we will use another trick.

We use here the technique on which **radix sort** is based: to sort the pairs we first sort them by the second element, and then by the first element (with a stable sort, i.e. sorting without breaking the relative order of equal elements). However the second elements were already sorted in the previous iteration. Thus, in order to sort the pairs by the second elements, we just need to subtract 2^{k-1} from the indices in $p[]$ (e.g. if the smallest substring of length 2^{k-1} starts at position i , then the substring of length 2^k with the smallest second half starts at $i - 2^{k-1}$).

So only by simple subtractions we can sort the second elements of the pairs in $p[]$. Now we need to perform a stable sort by the first elements. As already mentioned, this can be accomplished with counting sort.

The only thing left is to compute the equivalence classes $c[]$, but as before this can be done by simply iterating over the sorted permutation $p[]$ and comparing neighboring pairs.

Here is the remaining implementation. We use temporary arrays $pn[]$ and $cn[]$ to store the permutation by the second elements and the new equivalent class indices.

```
vector<int> pn(n), cn(n);
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;
    }
    fill(cnt.begin(), cnt.begin() + classes, 0);
    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++) {
        pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
        pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << h)) % n]};
        if (cur != prev)
            ++classes;
        cn[p[i]] = classes - 1;
    }
    c.swap(cn);
}
return p;
}
```

The algorithm requires $O(n \log n)$ time and $O(n)$ memory. However if we take the size of the alphabet k into account, then it uses $O((n + k) \log n)$ time and $O(n + k)$ memory.

For simplicity we used the complete ASCII range as alphabet. If we know that the string only contains a subset of characters, e.g. only lowercase letters, then this implementation can obviously be optimized. However not by much, since the alphabet size only appears with a factor of $O(\log n)$ in the complexity.

Also note, that this algorithm only sorts the cycle shifts. As mentioned at the beginning of this section we can generate the sorted order of the suffixes by appending a character that is smaller than all other characters of the string, and sorting this resulting string by cycle shifts, e.g. by sorting the cycle shifts of $s + \$$. This will obviously give the suffix array of s , however prepended with $|s|$.

```
vector<int> suffix_array_construction(string s) {
    s += "$";
    vector<int> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
```

1.5.3 Applications

Finding the smallest cyclic shift

The algorithm above sorts all cyclic shifts (without appending a character to the string), and therefore $p[0]$ gives the position of the smallest cyclic shift.

Finding a substring in a string

The task is to find a string s inside some text t online - we know the text t beforehand, but not the string s . We can create the suffix array for the text t in $O(|t| \log |t|)$ time. Now we can look for the substring s in the following way. The occurrence of s must be a prefix of some suffix from t . Since we sorted all the suffixes we can perform a binary search for s in p . Comparing the current suffix and the substring s within the binary search can be done in $O(|s|)$ time, therefore the complexity for finding the substring is $O(|s| \log |t|)$. Also notice that if the substring occurs multiple times in t , then all occurrences will be next to each other in p . Therefore the number of occurrences can be found with a second binary search, and all occurrences can be printed easily.

Comparing two substrings of a string

We want to be able to compare two substrings of the same length of a given string s in $O(1)$ time, i.e. checking if the first substring is smaller than the second one.

For this we construct the suffix array in $O(|s| \log |s|)$ time and store all the intermediate results of the equivalence classes $c[]$.

Using this information we can compare any two substring whose length is equal to a power of two in $O(1)$: for this it is sufficient to compare the equivalence classes of both substrings. Now we want to generalize this method to substrings of arbitrary length.

Let's compare two substrings of length l with the starting indices i and j . We find the largest length of a block that is placed inside a substring of this length: the greatest k such that $2^k \leq l$. Then comparing the two substrings can be replaced by comparing two overlapping blocks of length 2^k : first you need to compare the two blocks starting at i and j , and if these are equal then compare the two blocks ending in positions $i + l - 1$ and $j + l - 1$:

$$\begin{array}{ccc}
 \text{first} & & \text{second} \\
 \dots \underbrace{s_i \dots s_{i+l-2^k} \dots s_{i+2^k-1} \dots s_{i+l-1}}_{2^k} \dots & & \dots \underbrace{s_j \dots s_{j+l-2^k} \dots s_{j+2^k-1} \dots s_{j+l-1}}_{2^k} \dots \\
 \\
 \text{first} & & \text{second} \\
 \dots \underbrace{s_i \dots s_{i+l-2^k} \dots s_{i+2^k-1} \dots s_{i+l-1}}_{2^k} \dots & & \dots \underbrace{s_j \dots s_{j+l-2^k} \dots s_{j+2^k-1} \dots s_{j+l-1}}_{2^k} \dots
 \end{array}$$

Here is the implementation of the comparison. Note that it is assumed that the function gets called with the already calculated k . k can be computed with $\lfloor \log l \rfloor$, but it is more efficient to precompute all k values for every l . See

for instance the article about the [Sparse Table](#), which uses a similar idea and computes all log values.

```
int compare(int i, int j, int l, int k) {
    pair<int, int> a = {c[k][i], c[k][(i+1-(1 << k))%n]};
    pair<int, int> b = {c[k][j], c[k][(j+1-(1 << k))%n]};
    return a == b ? 0 : a < b ? -1 : 1;
}
```

Longest common prefix of two substrings with additional memory

For a given string s we want to compute the longest common prefix (**LCP**) of two arbitrary suffixes with position i and j .

The method described here uses $O(|s| \log |s|)$ additional memory. A completely different approach that will only use a linear amount of memory is described in the next section.

We construct the suffix array in $O(|s| \log |s|)$ time, and remember the intermediate results of the arrays $c[]$ from each iteration.

Let's compute the LCP for two suffixes starting at i and j . We can compare any two substrings with a length equal to a power of two in $O(1)$. To do this, we compare the strings by power of twos (from highest to lowest power) and if the substrings of this length are the same, then we add the equal length to the answer and continue checking for the LCP to the right of the equal part, i.e. i and j get added by the current power of two.

```
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 \log_n denotes a constant that is equal to the logarithm of n in base 2 rounded down.

Longest common prefix of two substrings without additional memory

We have the same task as in the previous section. We have compute the longest common prefix (**LCP**) for two suffixes of a string s .

Unlike the previous method this one will only use $O(|s|)$ memory. The result of the preprocessing will be an array (which itself is an important source of information about the string, and therefore also used to solve other tasks). LCP queries can be answered by performing RMQ queries (range minimum queries) in this array, so for different implementations it is possible to achieve logarithmic and even constant query time.

The basis for this algorithm is the following idea: we will compute the longest common prefix for each **pair of adjacent suffixes in the sorted order**. In other words we construct an array $\text{lcp}[0 \dots n-2]$, where $\text{lcp}[i]$ is equal to the length of the longest common prefix of the suffixes starting at $p[i]$ and $p[i+1]$. This array will give us an answer for any two adjacent suffixes of the string. Then the answer for arbitrary two suffixes, not necessarily neighboring ones, can be obtained from this array. In fact, let the request be to compute the LCP of the suffixes $p[i]$ and $p[j]$. Then the answer to this query will be $\min(\text{lcp}[i], \text{lcp}[i+1], \dots, \text{lcp}[j-1])$.

Thus if we have such an array lcp , then the problem is reduced to the **RMQ**, which has many wide number of different solutions with different complexities.

So the main task is to **build** this array lcp . We will use **Kasai's algorithm**, which can compute this array in $O(n)$ time.

Let's look at two adjacent suffixes in the sorted order (order of the suffix array). Let their starting positions be i and j and their lcp equal to $k > 0$. If we remove the first letter of both suffixes - i.e. we take the suffixes $i+1$ and $j+1$ - then it should be obvious that the lcp of these two is $k-1$. However we cannot use this value and write it in the lcp array, because these two suffixes might not be next to each other in the sorted order. The suffix $i+1$ will of course be smaller than the suffix $j+1$, but there might be some suffixes between them. However, since we know that the LCP between two suffixes is the minimum value of all transitions, we also know that the LCP between any two pairs in that interval has to be at least $k-1$, especially also between $i+1$ and the next suffix. And possibly it can be bigger.

Now we already can implement the algorithm. We will iterate over the suffixes in order of their length. This way we can reuse the last value k , since going from suffix i to the suffix $i+1$ is exactly the same as removing the first letter. Will need an additional array rank , which will give us the position of a suffix in the sorted list of suffixes.

```
vector<int> lcp_construction(string const& s, vector<int> const& p) {
    int n = s.size();
    vector<int> rank(n, 0);
    for (int i = 0; i < n; i++)
        rank[p[i]] = i;

    int k = 0;
    vector<int> lcp(n-1, 0);
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = p[rank[i] + 1];
        while (i + k < n && j + k < n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
        if (k)
            k--;
    }
}
```



```

    return lcp;
}

```

It is easy to see, that we decrease k at most $O(n)$ times (each iteration at most once, except for $\text{rank}[i] == n - 1$, where we directly reset it to 0), and the LCP between two strings is at most $n - 1$, we will also increase k only $O(n)$ times. Therefore the algorithm runs in $O(n)$ time.

Number of different substrings

We preprocess the string s by computing the suffix array and the LCP array. Using this information we can compute the number of different substrings in the string.

To do this, we will think about which **new** substrings begin at position $p[0]$, then at $p[1]$, etc. In fact we take the suffixes in sorted order and see what prefixes give new substrings. Thus we will not overlook any by accident.

Because the suffixes are sorted, it is clear that the current suffix $p[i]$ will give new substrings for all its prefixes, except for the prefixes that coincide with the suffix $p[i - 1]$. Thus, all its prefixes except the first $\text{lcp}[i - 1]$ one. Since the length of the current suffix is $n - p[i]$, $n - p[i] - \text{lcp}[i - 1]$ new suffixes start at $p[i]$. Summing over all the suffixes, we get the final answer:

$$\sum_{i=0}^{n-1} (n - p[i]) - \sum_{i=0}^{n-2} \text{lcp}[i] = \frac{n^2 + n}{2} - \sum_{i=0}^{n-2} \text{lcp}[i]$$

1.5.4 Practice Problems

- [Uva 760 - DNA Sequencing](#)
- [Uva 1223 - Editor](#)
- [Codechef - Tandem](#)
- [Codechef - Substrings and Repetitions](#)
- [Codechef - Entangled Strings](#)
- [Codeforces - Martian Strings](#)
- [Codeforces - Little Elephant and Strings](#)
- [SPOJ - Ada and Terramorphing](#)
- [SPOJ - Ada and Substring](#)
- [UVA - 1227 - The longest constant gene](#)
- [SPOJ - Longest Common Substring](#)
- [UVA 11512 - GATTACA](#)
- [LA 7502 - Suffixes and Palindromes](#)
- [GYM - Por Costel and the Censorship Committee](#)
- [UVA 1254 - Top 10](#)
- [UVA 12191 - File Recover](#)
- [UVA 12206 - Stammering Aliens](#)
- [Codechef - Jarvis and LCP](#)
- [LA 3943 - Liking's Letter](#)
- [UVA 11107 - Life Forms](#)

- [UVA 12974 - Exquisite Strings](#)
- [UVA 10526 - Intellectual Property](#)
- [UVA 12338 - Anti-Rhyme Pairs](#)
- [DevSkills Reconstructing Blue Print of Life](#)
- [UVA 12191 - File Recover](#)
- [SPOJ - Suffix Array](#)
- [LA 4513 - Stammering Aliens](#)
- [SPOJ - LCS2](#)
- [Codeforces - Fake News \(hard\)](#)
- [SPOJ - Longest Common Substring](#)
- [SPOJ - Lexicographical Substring Search](#)
- [Codeforces - Forbidden Indices](#)
- [Codeforces - Tricky and Clever Password](#)
- [LA 6856 - Circle of digits](#)

1.6 Aho-Corasick algorithm

Let there be a set of strings with the total length m (sum of all lengths). The Aho-Corasick algorithm constructs a data structure similar to a trie with some additional links, and then constructs a finite state machine (automaton) in $O(mk)$ time, where k is the size of the used alphabet.

The algorithm was proposed by Alfred Aho and Margaret Corasick in 1975.

1.6.1 Construction of the trie

Formally a trie is a rooted tree, where each edge of the tree is labeled by some letter. All outgoing edge from one vertex must have different labels.

Consider any path in the trie from the root to any vertex. If we write out the labels of all edges on the path, we get a string that corresponds to this path. For any vertex in the trie we will associate the string from the root to the vertex.

Each vertex will also have a flag leaf which will be true, if any string from the given set corresponds to this vertex.

Accordingly to build a trie for a set of strings means to build a trie such that each leaf vertex will correspond to one string from the set, and conversely that each string of the set corresponds to one leaf vertex.

We now describe how to construct a trie for a given set of strings in linear time with respect to their total length.

We introduce a structure for the vertices of the tree.

```
const int K = 26;

struct Vertex {
    int next[K];
    bool leaf = false;

    Vertex() {
        fill(begin(next), end(next), -1);
    }
};

vector<Vertex> trie(1);
```

Here we store the trie as an array of Vertex. Each Vertex contains the flag leaf, and the edges in the form of an array next[], where next[i] is the index to the vertex that we reach by following the character i , or -1 , if there is no such edge. Initially the trie consists of only one vertex - the root - with the index 0.

Now we implement a function that will add a string s to the trie. The implementation is extremely simple: we start at the root node, and as long as there are edges corresponding to the characters of s we follow them. If there is no edge for one character, we simply generate a new vertex and connect it via an edge. At the end of the process we mark the last vertex with flag leaf.

```
void add_string(string const& s) {
    int v = 0;
```

```

for (char ch : s) {
    int c = ch - 'a';
    if (trie[v].next[c] == -1) {
        trie[v].next[c] = trie.size();
        trie.emplace_back();
    }
    v = trie[v].next[c];
}
trie[v].leaf = true;
}

```

The implementation obviously runs in linear time. And since every vertex store k links, it will use $O(mk)$ memory.

It is possible to decrease the memory consumption to $O(m)$ by using a map instead of an array in each vertex. However this will increase the complexity to $O(n \log k)$.

1.6.2 Construction of an automaton

Suppose we have built a trie for the given set of strings. Now let's look at it from a different side. If we look at any vertex. The string that corresponds to it is a prefix of one or more strings in the set, thus each vertex of the trie can be interpreted as a position in one or more strings from the set.

In fact the trie vertices can be interpreted as states in a **finite deterministic automaton**. From any state we can transition - using some input letter - to other states, i.e. to another position in the set of strings. For example, if there is only one string in the trie abc , and we are standing at vertex 2 (which corresponds to the string ab), then using the letter c we can transition to the state 3.

Thus we can understand the edges of the trie as transitions in an automaton according to the corresponding letter. However for an automaton we cannot restrict the possible transitions for each state. If we try to perform a transition using a letter, and there is no corresponding edge in the trie, then we nevertheless must go into some state.

More strictly, let us be in a state p corresponding to the string t , and we want to transition to a different state with the character c . If there is an edge labeled with this letter c , then we can simply go over this edge, and get the vertex corresponding to $t + c$. If there is no such edge, then we must find the state corresponding to the longest proper suffix of the string t (the longest available in the trie), and try to perform a transition via c from there.

For example let the trie be constructed by the strings ab and bc , and we are currently at the vertex corresponding to ab , which is a leaf. For a transition with the letter c , we are forced to go to the state corresponding to the string b , and from there follow the edge with the letter c .

A **suffix link** for a vertex p is a edge that points to the longest proper suffix of the string corresponding to the vertex p . The only special case is the root of the trie, the suffix link will point to itself. Now we can reformulate the statement about the transitions in the automaton like this: while from the current vertex

of the trie there is no transition using the current letter (or until we reach the root), we follow the suffix link.

Thus we reduced the problem of constructing an automaton to the problem of finding suffix links for all vertices of the trie. However we will build these suffix links, oddly enough, using the transitions constructed in the automaton.

Note that if we want to find a suffix link for some vertex v , then we can go to the ancestor p of the current vertex (let c be the letter of the edge from p to v), then follow its suffix link, and perform from there the transition with the letter c .

Thus the problem of finding the transitions has been reduced to the problem of finding suffix links, and the problem of finding suffix links has been reduced to the problem of finding a suffix link and a transition, but for vertices closer to the root. So we have a recursive dependence that we can resolve in linear time.

Let's move to the implementation. Note that we now will store the ancestor p and the character pch of the edge from p to v for each vertex v . Also at each vertex we will store the suffix link `link` (or -1 if it hasn't been calculated yet), and in the array `go[k]` the transitions in the machine for each symbol (again -1 if it hasn't been calculated yet).

```
const int K = 26;

struct Vertex {
    int next[K];
    bool leaf = false;
    int p = -1;
    char pch;
    int link = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}

int go(int v, char ch);
```

```

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 ch) {
    int c = ch - 'a';
    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), ch);
    }
    return t[v].go[c];
}

```

It is easy to see, that due to the memorization of the found suffix links and transitions the total time for finding all suffix links and transitions will be linear.

1.6.3 Applications

Find all strings from a given set in a text

Given a set of strings and a text. We have to print all occurrences of all strings from the set in the given text in $O(\text{len} + \text{ans})$, where len is the length of the text and ans is the size of the answer.

We construct an automaton for this set of strings. We will now process the text letter by letter, transitioning during the different states. Initially we are at the root of the trie. If we are at any time at state v , and the next letter is c , then we transition to the next state with $\text{go}(v, c)$, thereby either increasing the length of the current match substring by 1, or decreasing it by following a suffix link.

How can we find out for a state v , if there are any matches with strings for the set? First, it is clear that if we stand on a leaf vertex, then then string corresponding to the vertex ends at this position in the text. However this is by no means the only possible case of achieving a match: if we can reach one or more leaf vertices by moving along the suffix links, then there will be also a match corresponding to each found leaf vertex. A simple example demonstrating this situation can be creating using the set of strings $\{dabce, abc, bc\}$ and the text $dabc$.

Thus if we store in each leaf vertex the index of the string corresponding to it (or the list of indices if duplicate strings appear in the set), then we can find in $O(n)$ time the indices of all strings which match the current state, by simply following the suffix links from the current vertex to the root. However this is

not the most efficient solution, since this gives us $O(n \text{ len})$ complexity in total. However this can be optimized by computing and storing the nearest leaf vertex that is reachable using suffix links (this is sometimes called the **exit link**). This value we can compute lazily in linear time. Thus for each vertex we can advance in $O(1)$ time to the next marked vertex in the suffix link path, i.e. to the next match. Thus for each match we spend $O(1)$ time, and therefore we reach the complexity $O(\text{len} + \text{ans})$.

If you only want to count the occurrences and not find the indices themselves, you can calculate the number of marked vertices in the suffix link path for each vertex v . This can be calculated in $O(n)$ time in total. Thus we can sum up all matches in $O(\text{len})$.

Finding the lexicographical smallest string of a given length that doesn't match any given strings

A set of strings and a length L is given. We have to find a string of length L , which does not contain any of the string, and derive the lexicographical smallest of such strings.

We can construct the automaton for the set of strings. Let's remember, that the vertices from which we can reach a leaf vertex are the states, at which we have a match with a string from the set. Since in this task we have to avoid matches, we are not allowed to enter such states. On the other hand we can enter all other vertices. Thus we delete all "bad" vertices from the machine, and in the remaining graph of the automaton we find the lexicographical smallest path of length L . This task can be solved in $O(L)$ for example by [depth first search](#).

Finding the shortest string containing all given strings

Here we use the same ideas. For each vertex we store a mask that denotes the strings which match at this state. Then the problem can be reformulated as follows: initially being in the state ($v = \text{root}$, $\text{mask} = 0$), we want to reach the state (v , $\text{mask} = 2^n - 1$), where n is the number of strings in the set. When we transition from one state to another using a letter, we update the mask accordingly. By running a [breath first search](#) we can find a path to the state (v , $\text{mask} = 2^n - 1$) with the smallest length.

Finding the lexicographical smallest string of length L containing k strings

As in the previous problem, we calculate for each vertex the number of matches that correspond to it (that is the number of marked vertices reachable using suffix links). We reformulate the problem: the current state is determined by a triple of numbers (v , len , cnt), and we want to reach from the state (root , 0 , 0) the state (v , L , k), where v can be any vertex. Thus we can find such a path using depth first search (and if the search looks at the edges in their natural order, then the found path will automatically be the lexicographical smallest).

1.6.4 Problems

- [UVA #11590 - Prefix Lookup](#)
- [UVA #11171 - SMS](#)
- [UVA #10679 - I Love Strings!!](#)
- [Codeforces - Frequency of String](#)

Chapter 2

Advanced

2.1 Suffix Tree. Ukkonen's Algorithm

This article is a stub and doesn't contain any descriptions. For a description of the algorithm, refer to other sources, such as [Algorithms on Strings, Trees, and Sequences](#) by Dan Gusfield.

This algorithm builds a suffix tree for a given string s of length n in $O(n \log(k))$ time, where k is the size of the alphabet (if k is considered to be a constant, the asymptotic behavior is linear).

The input to the algorithm are the string s and its length n , which are passed as global variables.

The main function `build_tree` builds a suffix tree. It is stored as an array of structures `node`, where `node[0]` is the root of the tree.

In order to simplify the code, the edges are stored in the same structures: for each vertex its structure `node` stores the information about the edge between it and its parent. Overall each `node` stores the following information:

- `(l, r)` - left and right boundaries of the substring `s[l..r-1]` which correspond to the edge to this node,
- `par` - the parent node,
- `link` - the suffix link,
- `next` - the list of edges going out from this node.

```
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);
}

```

2.1.1 Compressed Implementation

This compressed implementation was proposed by [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++]=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');
}

```

Same code with comments:

```

const int N=1000000,      // maximum possible number of nodes in suffix tree
        INF=1000000000; // infinity constant
string a;                // input string for which the suffix tree is being built
int t[N][26],            // array of transitions (state, letter)
    l[N],                // left...
    r[N],                // ...and right boundaries of the substring of a which correspond to incoming edge
    p[N],                // parent of the node
    s[N],                // suffix link
    tv,                 // the node of the current suffix (if we're mid-edge, the lower node of the edge)
    tp,                 // position in the string which corresponds to the position on the edge (between
    ts,                 // the number of nodes
    la;                 // the current character in the string

void ukkadd(int c) { // add character s to the tree
    suff:;            // we'll return here after each transition to the suffix (and will add charac
    if (r[tv]<tp) { // check whether we're still within the boundaries of the current edge
        // if we're not, find the next edge. If it doesn't exist, create a leaf and add it to
        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];
    } // otherwise just proceed to the next edge
    if (tp==-1 || c==a[tp]-'a')
        tp++; // if the letter on the edge equal c, go down that edge
    else {
        // otherwise split the edge in two with middle in node ts
        l[ts]=l[tv];r[ts]=tp-1;p[ts]=p[tv];t[ts][a[tp]-'a']=tv;
        // add leaf ts+1. It corresponds to transition through c.
        t[ts][c]=ts+1;l[ts+1]=la;p[ts+1]=ts;
        // update info for the current node - remember to mark ts as parent of tv
        l[tv]=tp;p[tv]=ts;t[p[ts]][a[l[ts]]-'a']=ts;ts+=2;
        // prepare for descent
        // tp will mark where are we in the current suffix
        tv=s[p[ts-2]];tp=l[ts-2];
        // while the current suffix is not over, descend
        while (tp<=r[ts-2]) {tv=t[tv][a[tp]-'a'];tp+=r[tv]-l[tv]+1;}
        // if we're in a node, add a suffix link to it, otherwise add the link to ts
        // (we'll create ts on next iteration).
        if (tp==r[ts-2]+1) s[ts-2]=tv; else s[ts-2]=ts;
    }
}

```

```
        // add tp to the new edge and return to add letter to suffix
        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);
    // initialize data for the root of the tree
    s[0]=1;
    l[0]=-1;
    r[0]=-1;
    l[1]=-1;
    r[1]=-1;
    memset (t, -1, sizeof t);
    fill(t[1],t[1]+26,0);
    // add the text to the tree, letter by letter
    for (la=0; la<(int)a.size(); ++la)
        ukkadd (a[la]-'a');
}
```

2.1.2 Practice Problems

- [UVA 10679 - I Love Strings!!!](#)

2.2 Suffix Automaton

A **suffix automaton** is a powerful data structure that allows solving many string-related problems.

For example, you can search for all occurrences of one string in another, or count the amount of different substrings of a given string. Both tasks can be solved in linear time with the help of a suffix automaton.

Intuitively a suffix automaton can be understood as compressed form of **all substrings** of a given string. An impressive fact is, that the suffix automaton contains all this information in a highly compressed form. For a string of length n it only requires $O(n)$ memory. Moreover, it can also be build in $O(n)$ time (if we consider the size k of the alphabet as a constant), otherwise both the memory and the time complexity will be $O(n \log k)$.

The linearity of the size of the suffix automaton was first discovered in 1983 by Blumer et al., and in 1985 the first linear algorithms for the construction was presented by Crochemore and Blumer.

2.2.1 Definition of a suffix automaton

A suffix automaton for a given string s is a minimal **DFA** (deterministic finite automaton / deterministic finite state machine) that accepts all the suffixes of the string s .

In other words:

- A suffix automaton is an oriented acyclic graph. The vertices are called **states**, and the edges are called **transitions** between states.
- One of the states t_0 is the **initial state**, and it must be the source of the graph (all other states are reachable from t_0).
- Each **transition** is labeled with some character. All transitions originating from a state must have **different** labels.
- One or multiple states are marked as **terminal states**. If we start from the initial state t_0 and move along transitions to a terminal state, then the labels of the passed transitions must spell one of the suffixes of the string s . Each of the suffixes of s must be spellable using a path from t_0 to a terminal state.
- The suffix automaton contains the minimum number of vertices among all automata satisfying the conditions described above.

Substring property

The simplest and most important property of a suffix automaton is, that it contains information about all substrings of the string s . Any path starting at the initial state t_0 , if we write down the labels of the transitions, forms a **substring** of s . And conversely every substring of s corresponds to a certain path starting at t_0 .

In order to simplify the explanations, we will say that the substrings **corresponds** to that path (starting at t_0 and the labels spell the substring). And

conversely we say that any path **corresponds** to the string spelled by its labels.

One or multiple paths can lead to a state. Thus, we will say that a state **corresponds** to the set of strings, which correspond to these paths.

Examples of constructed suffix automata

Here we will show some examples of suffix automata for several simple strings.

We will denote the initial state with blue and the terminal states with green.

For the string $s = ""$:



Figure 2.1: Suffix automaton for ""

For the string $s = "a"$:

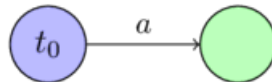


Figure 2.2: Suffix automaton for "a"

For the string $s = "aa"$:

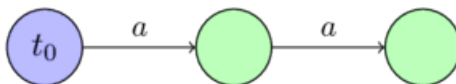


Figure 2.3: Suffix automaton for "aa"

For the string $s = "ab"$:

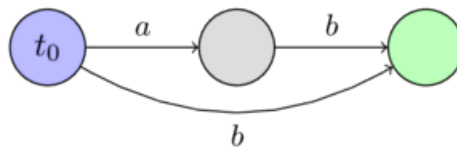


Figure 2.4: Suffix automaton for “ab”

For the string $s = \text{"aba"}$:

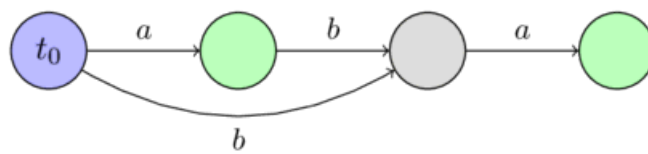


Figure 2.5: Suffix automaton for “aba”

For the string $s = \text{"abb"}$:

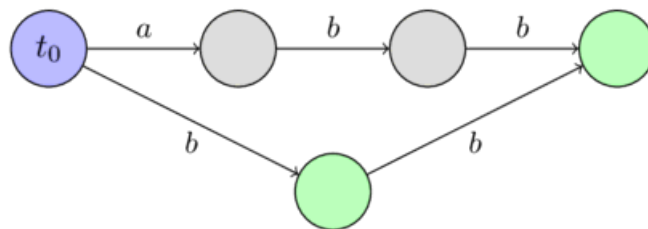


Figure 2.6: Suffix automaton for “abb”

For the string $s = \text{"abbb"}$:

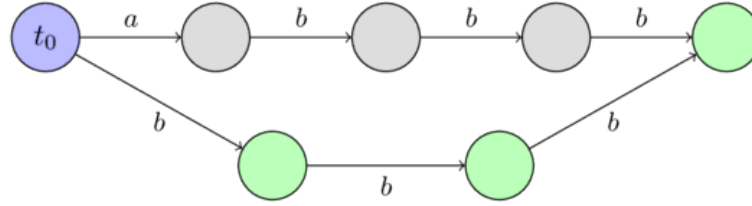


Figure 2.7: Suffix automaton for “abbb”

2.2.2 Construction in linear time

Before we describe the algorithm to construct a suffix automaton in linear time, we need to introduce several new concepts and simple proofs, which will be very important in understanding the construction.

End positions *endpos*

Consider any non-empty substring t of the string s . We will denote with $endpos(t)$ the set of all positions in the string s , in which the occurrences of t end. For instance, we have $endpos("bc") = \{2, 4\}$ for the string "abcbc".

We will call two substrings t_1 and t_2 *endpos*-equivalent, if their ending sets coincide: $endpos(t_1) = endpos(t_2)$. Thus all non-empty substrings of the string s can be decomposed into several **equivalence classes** according to their sets *endpos*.

It turns out, that in a suffix machine *endpos*-equivalent substrings **correspond to the same state**. In other words the number of states in a suffix automaton is equal to the number of equivalence classes among all substrings, plus the initial state. Each state of a suffix automaton corresponds to one or more substrings having the same value *endpos*.

We will later describe the construction algorithm using this assumption. We will then see, that all the required properties of a suffix automaton, except for the minimality, are fulfilled. And the minimality follows from Nerode's theorem (which will not be proven in this article).

We can make some important observations concerning the values *endpos*:

Lemma 1: Two non-empty substrings u and w (with $length(u) \leq length(w)$) are *endpos*-equivalent, if and only if the string u occurs in s only in the form of a suffix of w .

The proof is obvious. If u and w have the same *endpos* values, then u is a suffix of w and appears only in the form of a suffix of w in s . And if u is a suffix of w and appears only in the form as a suffix in s , then the values *endpos* are equal by definition.

Lemma 2: Consider two non-empty substrings u and w (with $length(u) \leq length(w)$). Then their sets *endpos* either don't intersect at all, or $endpos(w)$ is a subset of $endpos(u)$. And it depends on if u is a suffix of w or not.

$$\begin{cases} \text{endpos}(w) \subseteq \text{endpos}(u) & \text{if } u \text{ is a suffix of } w \\ \text{endpos}(w) \cap \text{endpos}(u) = \emptyset & \text{otherwise} \end{cases}$$

Proof: If the sets $\text{endpos}(u)$ and $\text{endpos}(w)$ have at least one common element, then the strings u and w both end in that position, i.e. u is a suffix of w . But then at every occurrence of w also appears the substring u , which means that $\text{endpos}(w)$ is a subset of $\text{endpos}(u)$.

Lemma 3: Consider an endpos -equivalence class. Sort all the substrings in this class by non-increasing length. Then in the resulting sequence each substring will be one shorter than the previous one, and at the same time will be a suffix of the previous one. In other words the substrings in the same equivalence class are actually each others suffixes, and take all possible lengths in a certain interval $[x; y]$.

Proof: Fix some endpos -equivalent class. If it only contains one string, then the lemma is obviously true. Now let's say that the number of strings in the class is greater than one.

According to Lemma 1, two different endpos -equivalent strings are always in such a way, that the shorter one is a proper suffix of the longer one. Consequently, there cannot be two strings of the same length in the equivalence class.

Let's denote by w the longest, and through u the shortest string in the equivalence class. According to Lemma 1, the string u is a proper suffix of the string w . Consider now any suffix of w with a length in the interval $[\text{length}(u); \text{length}(w)]$. It is easy to see, that this suffix is also contained in the same equivalence class. Because this suffix can only appear in the form of a suffix of w in the string s (since also the shorter suffix u occurs in s only in the form of a suffix of w). Consequently, according to Lemma 1, this suffix is endpos -equivalent to the string w .

Suffix links *link*

Consider some state $v \neq t_0$ in the automaton. As we know, the state v corresponds to the class of strings with the same endpos values. And if we denote by w the longest of these strings, then all the other strings are suffixes of w .

We also know the first few suffixes of a string w (if we consider suffixes in descending order of their length) are all contained in this equivalent class, and all other suffixes (at least one other - the empty suffix) are in some other classes. We denote by t the biggest such suffix, and make a suffix link to it.

In other words, a **suffix link** $\text{link}(v)$ leads to the state that corresponds to the **longest suffix** of w that is another endpos -equivalent class.

Here we assume that the initial state t_0 corresponds to its own equivalence class (containing only the empty string), and for convenience we set $\text{endpos}(t) = \{-1, 0, \dots, \text{length}(s) - 1\}$.

Lemma 4: Suffix links form a **tree** with the root t_0 .

Proof: Consider an arbitrary state $v \neq t_0$. A suffix link $\text{link}(v)$ leads to a state corresponding to strings with strictly smaller length (this follows from the definition of the suffix links and from Lemma 3). Therefore, by moving along the

suffix links, we will sooner or later come to the initial state t_0 , which corresponds to the empty string.

Lemma 5: If we construct a tree using the sets $endpos$ (by the rule that the set of a parent node contains the sets of all children as subsets), then the structure will coincide with the tree of suffix links.

Proof: The fact that we can construct a tree using the sets $endpos$ follows directly from Lemma 2 (that any two sets either do not intersect or one is contained in the other).

Let us now consider an arbitrary state $v \neq t_0$, and its suffix link $link(v)$. From the definition of the suffix link and from Lemma 2 it follows that

$$endpos(v) \subseteq endpos(link(v)),$$

which together with the previous lemma proves the assertion: the tree of suffix links is essentially a tree of sets $endpos$.

Here is an **example** of a tree of suffix links in the suffix automaton build for the string "abcabc". The nodes are labeled with the longest substring from the corresponding equivalence class.

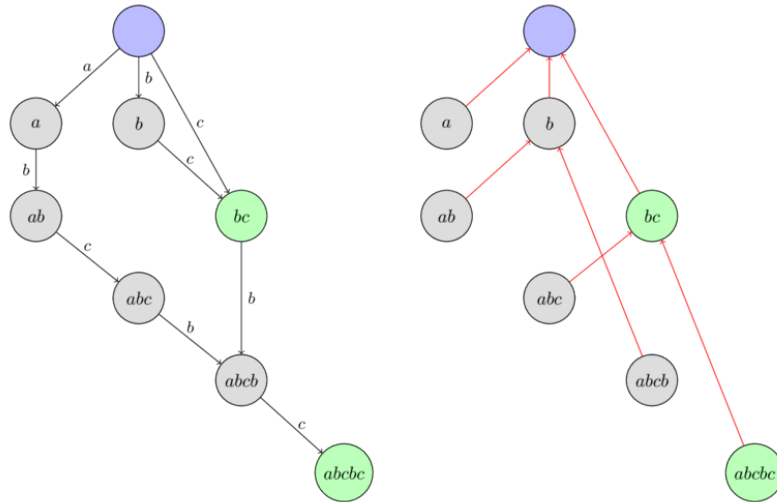


Figure 2.8: Suffix automaton for "abcabc" with suffix links

Recap

Before proceeding to the algorithm itself, we recap the accumulated knowledge, and introduce a few auxiliary notations.

- The substrings of the string s can be decomposed into equivalence classes according to their end positions $endpos$.
- The suffix automaton consists of the initial state t_0 , as well as of one state for each $endpos$ -equivalence class.

- For each state v one or multiple substrings match. We denote by $longest(v)$ the longest such string, and through $len(v)$ its length. We denote by $shortest(v)$ the shortest such substring, and its length with $minlen(v)$. Then all the strings corresponding to this state are different suffixes of the string $longest(v)$ and have all possible lengths in the interval $[minlength(v); len(v)]$.
- For each state $v \neq t_0$ a suffix link is defined as a link, that leads to a state that corresponds to the suffix of the string $longest(v)$ of length $minlen(v) - 1$. The suffix links form a tree with the root in t_0 , and at the same time this tree forms an inclusion relationship between the sets $endpos$.
- We can express $minlen(v)$ for $v \neq t_0$ using the suffix link $link(v)$ as:

$$minlen(v) = len(link(v)) + 1$$

- If we start from an arbitrary state v_0 and follow the suffix links, then sooner or later we will reach the initial state t_0 . In this case we obtain a sequence of disjoint intervals $[minlen(v_i); len(v_i)]$, which in union forms the continuous interval $[0; len(v_0)]$.

Algorithm

Now we can proceed to the algorithm itself. The algorithm will be **online**, i.e. we will add the characters of the string one by one, and modify the automaton accordingly in each step.

To achieve linear memory consumption, we will only store the values len , $link$ and a list of transitions in each state. We will not label terminal states (but we will later show how to arrange these labels after constructing the suffix automaton).

Initially the automaton consists of a single state t_0 , which will be the index 0 (the remaining states will receive the indices $1, 2, \dots$). We assign it $len = 0$ and $link = -1$ for convenience (-1 will be a fictional, non-existing state).

Now the whole task boils down to implementing the process of **adding one character** c to the end of the current string. Let us describe this process:

- Let $last$ be the state corresponding to the entire string before adding the character c . (Initially we set $last = 0$, and we will change $last$ in the last step of the algorithm accordingly.)
- Create a new state cur , and assign it with $len(cur) = len(last) + 1$. The value $link(cur)$ is not known at the time.
- Now we do the following procedure: We start at the state $last$. While there isn't a transition through the letter c , we will add a transition to the state cur , and follow the suffix link. If at some point there already exists a transition through the letter c , then we will stop and denote this state with p .

- If it haven't found such a state p , then we reached the fictitious state -1 , then we can just assign $link(cur) = 0$ and leave.
- Suppose now that we have found a state p , from which there exists a transition through the letter c . We will denote the state, to which the transition leads, with q .
- Now we have two cases. Either $len(p) + 1 = len(q)$, or not.
- If $len(p) + 1 = len(q)$, then we can simply assign $link(cur) = q$ and leave.
- Otherwise it is a bit more complicated. It is necessary to **clone** the state q : we create a new state $clone$, copy all the data from q (suffix link and transition) except the value len . We will assign $len(clone) = len(p) + 1$.

After cloning we direct the suffix link from cur to $clone$, and also from q to $clone$.

Finally we need to walk from the state p back using suffix links as long as there is a transition through c to the state q , and redirect all those to the state $clone$.

- In any of the three cases, after completing the procedure, we update the value $last$ with the state cur .

If we also want to know which states are **terminal** and which are not, the we can find all terminal states after constructing the complete suffix automaton for the entire string s . To do this, we take the state corresponding to the entire string (stored in the variable $last$), and follow its suffix links until we reach the initial state. We will mark all visited states as terminal. It is easy to understand that by doing so we will mark exactly the states corresponding to all the suffixes of the string s , which are exactly the terminal states.

In the next section we will look in detail at each step and show its **correctness**.

Here we only note that, since we only create one or two new states for each character of s , the suffix automaton contains a **linear number of states**.

The linearity of the number of transitions, and in general the linearity of the runtime of the algorithm is less clear, and they will be proven after we proved the correctness.

Correctness

- We will call a transition (p, q) **continuous** if $len(p) + 1 = len(q)$. Otherwise, i.e. when $len(p) + 1 < len(q)$, the transition will be called **non-continuous**.

As we can see from the description of the algorithm, continuous and non-continuous transitions will lead to different cases of the algorithm. Continuous transitions are fixed, and will never change again. In contrast non-continuous transition may change, when new letters are added to the string (the end of the transition edge may change).

- To avoid ambiguity we will denote the string, for which the suffix automaton was built before adding the current character c , with s .
- The algorithm begins with creating a new state cur , which will correspond to the entire string $s + c$. It is clear why we have to create a new state. Together with the new character a new equivalence class is created.
- After creating a new state we traverse by suffix links starting from the state corresponding to the entire string s . For each state we try to add a transition with the character c to the new state cur . Thus we append to each suffix of s the character c . However we can only add these new transitions, if they don't conflict with an already existing one. Therefore as soon as we find an already existing transition with c we have to stop.
- In the simplest case we reached the fictitious state -1 . This means we added the transition with c to all suffixes of s . This also means, that the character c hasn't been part of the string s before. Therefore the suffix link of cur has to lead to the state 0.
- In the second case we came across an existing transition (p, q) . This means that we tried to add a string $x + c$ (where x is a suffix of s) to the machine that **already exists** in the machine (the string $x + c$ already appears as a substring of s). Since we assume that the automaton for the string s is build correctly, we should not add a new transition here.

However there is a difficulty. To which state should the suffix link from the state cur lead? We have to make a suffix link to a state, in which the longest string is exactly $x + c$, i.e. the len of this state should be $len(p) + 1$. However it is possible, that such a state doesn't yet exists, i.e. $len(q) > len(p) + 1$. In this case we have to create such a state, by **splitting** the state q .

- If the transition (p, q) turns out to be continuous, then $len(q) = len(p) + 1$. In this case everything is simple. We direct the suffix link from cur to the state q .
- Otherwise the transition is non-continuous, i.e. $len(q) > len(p) + 1$. This means that the state q corresponds to not only the suffix of $s + c$ with length $len(p) + 1$, but also to longer substrings of s . We can do nothing other than **splitting** the state q into two sub-states, so that the first one has length $len(p) + 1$.

How can we split a state? We **clone** the state q , which gives us the state $clone$, and we set $len(clone) = len(p) + 1$. We copy all the transitions from q to $clone$, because we don't want to change the paths that traverse through q . Also we set the suffix link from $clone$ to the target of the suffix link of q , and set the suffix link of q to $clone$.

And after splitting the state, we set the suffix link from cur to $clone$.

In the last step we change some of the transitions to q , we redirect them to $clone$. Which transitions do we have to change? It is enough to redirect

only the transitions corresponding to all the suffixes of the string $w + c$ (where w is the longest string of p), i.e. we need to continue to move along the suffix links, starting from the vertex p until we reach the fictitious state -1 or a transition that leads to a different state than q .

Linear number of operations

First we immediately make the assumption that the size of the alphabet is **constant**. If this is not the case, then it will not be possible to talk about the linear time complexity. The list of transitions from one vertex will be stored in a balanced tree, which allows you to quickly perform key search operations and adding keys. Therefore if we denote with k the size of the alphabet, then the asymptotic behavior of the algorithm will be $O(n \log k)$ with $O(n)$ memory. However if the alphabet is small enough, then you can sacrifice memory by avoiding balanced trees, and store the transitions at each vertex as an array of length k (for quick searching by key) and a dynamic list (to quickly traverse all available keys). Thus we reach the $O(n)$ time complexity for the algorithm, but at a cost of $O(nk)$ memory complexity.

So we will consider the size of the alphabet to be constant, i.e. each operation of searching for a transition on a character, adding a transition, searching for the next transition - all these operations can be done in $O(1)$.

If we consider all parts of the algorithm, then it contains three places in the algorithm in which the linear complexity is not obvious:

- The first place is the traversal through the suffix links from the state *last*, adding transitions with the character c .
- The second place is the copying of transitions when the state q is cloned into a new state *clone*.
- Third place is changing the transition leading to q , redirecting them to *clone*.

We use the fact that the size of the suffix automaton (both in number of states and in the number of transitions) is **linear**. (The proof of the linearity of the number of states is the algorithm itself, and the proof of linearity of the number of states is given below, after the implementation of the algorithm).

Thus the total complexity of the **first and second places** is obvious, after all each operation adds only one amortized new transition to the automaton.

It remains to estimate the total complexity of the **third place**, in which we redirect transitions, that pointed originally to q , to *clone*. We denote $v = \text{longest}(p)$. This is a suffix of the string s , and with each iteration its length decreases - and therefore the position v as the suffix of the string s increases monotonically with each iteration. In this case, if before the first iteration of the loop, the corresponding string v was at the depth k ($k \geq 2$) from *last* (by counting the depth as the number of suffix links), then after the last iteration the string $v + c$ will be a 2-th suffix link on the path from *cur* (which will become the new value *last*).

Thus, each iteration of this loop leads to the fact that the position of the string $longest(link(link(last)))$ as suffix of the current string will monotonically increase. Therefore this cycle cannot be executed more than n iterations, which was required to prove.

Implementation

First we describe a data structure that will store all information about a specific transition (len , $link$ and the list of transitions). If necessary you can add a terminal flag here, as well as other information. We will store the list of transitions in the form of a *map*, which allows us to achieve total $O(n)$ memory and $O(n \log k)$ time for processing the entire string.

```
struct state {
    int len, link;
    map<char, int> next;
};
```

The suffix automaton itself will be stored in an array of these structures *state*. We store the current size *sz* and also the variable *last*, the state corresponding to the entire string at the moment.

```
const int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last;
```

We give a function that initializes a suffix automaton (creating a suffix automaton with a single state).

```
void sa_init() {
    st[0].len = 0;
    st[0].link = -1;
    sz++;
    last = 0;
}
```

And finally we give the implementation of the main function - which adds the next character to the end of the current line, rebuilding the machine accordingly.

```
void sa_extend(char c) {
    int cur = sz++;
    st[cur].len = st[last].len + 1;
    int p = last;
    while (p != -1 && !st[p].next.count(c)) {
        st[p].next[c] = cur;
        p = st[p].link;
    }
    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;
        while (p != -1 && st[p].next[c] == q) {
            st[p].next[c] = clone;
            p = st[p].link;
        }
        st[q].link = st[cur].link = clone;
    }
}
last = cur;
}

```

As mentioned above, if you sacrifice memory ($O(nk)$, where k is the size of the alphabet), then you can achieve the build time of the machine in $O(n)$, even for any alphabet size k . But for this you will have to store an array of size k in each state (for quickly jumping to the transition of the letter), and additional a list of all transitions (to quickly iterate over the transitions them).

2.2.3 Additional properties

Number of states

The number of states in a suffix automaton of the string s of length n **doesn't exceed** $2n - 1$ (for $n \geq 2$).

The proof is the construction algorithm itself, since initially the automaton consists of one state, and in the first and second iteration only a single state will be created, and in the remaining $n - 2$ steps at most 2 states will be created each.

However we can also **show** this estimation **without knowing the algorithm**. Let us recall that the number of states is equal to the number of different sets *endpos*. In addition these sets *endpos* form a tree (a parent vertex contains all children sets in his set). Consider this tree and transform it a little bit: as long as it has an internal vertex with only one child (which means that the set of the child misses at least one position from the parent set), we create a new child with the set of the missing positions. In the end we have a tree in which each inner vertex has a degree greater than one, and the number of leaves does not exceed n . Therefore there are no more than $2n - 1$ vertices in such a tree.

This bound of the number of states can actually be achieved for each n . A possible string is:

"abbb...bbb"

In each iteration, starting at the third one, the algorithm will split a state, resulting in exactly $2n - 1$ states.

Number of transitions

The number of transitions in a suffix automaton of a string s of length n **doesn't exceed** $3n - 4$ (for $n \geq 3$).

Let us prove this:

Let us first estimate the number of continuous transitions. Consider a spanning tree of the longest paths in the automaton starting in the state t_0 . This skeleton will consist of only the continuous edges, and therefore their number is less than the number of states, i.e. it does not exceed $2n - 2$.

Now let us estimate the number of non-continuous transitions. Let the current non-continuous transition be (p, q) with the character c . We take the correspondent string $u + c + w$, where the string u corresponds to the longest path from the initial state to p , and w to the longest path from q to any terminal state. On one hand, each such string $u + c + w$ for each incomplete strings will be different (since the strings u and w are formed only by complete transitions). On the other hand each such string $u + c + w$, by the definition of the terminal states, will be a suffix of the entire string s . Since there are only n non-empty suffixes of s , and non of the strings $u + c + w$ can contain s (because the entire string only contains complete transitions), the total number of incomplete transitions does not exceed $n - 1$.

Combining these two estimates gives us the bound $3n - 3$. However, since the maximum number of states can only be achieved with the test case "*abbb...bbb*" and this case has clearly less than $3n - 3$ transitions, we get the tighter bound of $3n - 4$ for the number of transitions in a suffix automaton.

This bound can also be achieved with the string:

"abbb...bbbc"

2.2.4 Applications

Here we look at some tasks that can be solved using the suffix automaton. For the simplicity we assume that the alphabet size k is constant, which allows us to consider the complexity of appending a character and the traversal as constant.

Check for occurrence

Given a text T , and multiple patters P . We have to check whether or not the strings P appear as a substring of T .

We build a suffix automaton of the text T in $O(\text{length}(T))$ time. To check if a pattern P appears in T , we follow the transitions, starting from t_0 , according to the characters of P . If at some point there doesn't exists a transition, then the pattern P doesn't appear as a substring of T . If we can process the entire string P this way, then the string appears in T .

It is clear that this will take $O(\text{length}(P))$ time for each string P . Moreover the algorithm actually finds the length of the longest prefix of P that appears in the text.

Number of different substrings

Given a string S . You want to compute the number of different substrings.

Let us build a suffix automaton for the string S .

Each substring of S corresponds to some path in the automaton. Therefore the number of different substrings is equal to the number of different paths in the automaton starting at t_0 .

Given that the suffix automaton is a directed acyclic graph, the number of different ways can be computed using dynamic programming.

Namely, let $d[v]$ be the number of ways, starting at the state v (including the path of length zero). Then we have the recursion:

$$d[v] = 1 + \sum_{w:(v,w,c) \in SA} d[w]$$

I.e. $d[v]$ can be expressed as the sum of answers for all ends of the transitions of v .

The number of different substrings is the value $d[t_0] - 1$ (since we don't count the empty substring).

Total time complexity: $O(\text{length}(S))$

Total length of all different substrings

Given a string S . We want to compute the total length of all its various substrings.

The solution is similar to the previous one, only now it is necessary to consider two quantities for the dynamic programming part: the number of different substrings $d[v]$ and their total length $ans[v]$.

We already described how to compute $d[v]$ in the previous task. The value $ans[v]$ can be computed using the recursion:

$$ans[v] = \sum_{w:(v,w,c) \in DAWG} d[w] + ans[w]$$

We take the answer of each adjacent vertex w , and add to it $d[w]$ (since every substrings is one character longer when starting from the state v).

Again this task can be computed in $O(\text{length}(S))$ time.

Lexicographically k -th substring

Given a string S . We have to answer multiple queries. For each given number K_i we have to find the K_i -th string in the lexicographically ordered list of all substrings.

The solution of this problem is based on the idea of the previous two problems. The lexicographically k -th substring corresponds to the lexicographically k -th path in the suffix automaton. Therefore after counting the number of paths from each state, we can easily search for the k -th path starting from the root of the automaton.

This takes $O(\text{length}(S))$ time for preprocessing and then $O(\text{length}(ans) \cdot k)$ for each query (where ans is the answer for the query and k is the size of the alphabet).

Smallest cyclic shift

Given a string S . We want to find the lexicographically smallest cyclic shift.

We construct a suffix automaton for the string $S + S$. Then the automaton will contain in itself as paths all the cyclic shifts of the string S .

Consequently the problem is reduced to finding the lexicographically smallest path of length $\text{length}(S)$, which can be done in a trivial way: we start in the initial state and greedily pass through the transitions with the minimal character.

Total time complexity is $O(\text{length}(S))$.

Number of occurrences

For a given text T . We have to answer multiple queries. For each given pattern P we have to find out how many times the string P appears in the string T as substring.

We construct the suffix automaton for the text T .

Next we do the following preprocessing: for each state v in the automaton we calculate the number $\text{cnt}[v]$ that is equal to the size of the set $\text{endpos}(v)$. In fact all strings corresponding to the same state v appear in the text T an equal amount of times, which is equal to the number of positions in the set endpos .

However we cannot construct the sets endpos explicitly, therefore we only consider their sizes cnt .

To compute them we proceed as follows. For each state, if it was not created by cloning (and if it is not the initial state t_0), we initialize it with $\text{cnt} = 1$. Then we will go through all states in decreasing order of their length len , and add the current value $\text{cnt}[v]$ to the suffix links:

$$\text{cnt}[\text{link}(v)] += \text{cnt}[v]$$

This gives the correct value for each state.

Why is this correct? The total stats obtained not obtained by cloning are exactly $\text{length}(T)$, and the first i of them appeared when we added the first i characters. Consequently for each of these states we count the corresponding position at which it was processed. Therefore initially we have $\text{cnt} = 1$ for each such state, and $\text{cnt} = 0$ for all other.

Then we apply the following operation for each v : $\text{cnt}[\text{link}(v)] += \text{cnt}[v]$. The meaning behind this is, that if a string v appears $\text{cnt}[v]$ times, then also all its suffixes appear at the exact same end positions, therefore also $\text{cnt}[v]$ times.

Why don't we overcount in this procedure (i.e. don't count some position twice)? Because we add the positions of a state to only one other state, so it can not happen that one state directs its positions to another state twice in two different ways.

Thus we can compute the quantities cnt for all states in the automaton in $O(\text{length}(T))$ time.

After that answering a query by just looking up the value $\text{cnt}[t]$, where t is the state corresponding to the pattern, if such a state exists. Otherwise answer with 0. Answering a query takes $O(\text{length}(P))$ time.

First occurrence position

Given a text T and multiple queries. For each query string P we want to find the position of the first occurrence of P in the string T (the position of the beginning of P).

We again construct a suffix automaton. Additionally we precompute the position $firstpos$ for all states in the automaton, i.e. for each state v we want to find the position $firstpos[v]$ of the end of the first occurrence. In other words, we want to find in advance the minimal element of each set $endpos$ (since obviously cannot maintain all sets $endpos$ explicitly).

To maintain these positions $firstpos$ we extend the function `sa_extend()`. When we create a new state cur , we set:

$$firstpos(cur) = len(cur) - 1$$

And when we clone a vertex q as $clone$, we set:

$$firstpos(clone) = firstpos(q)$$

(since the only other option for a value would be $firstpos(cur)$ which is definitely too big)

Thus the answer for a query is simply $firstpos(t) - length(P) + 1$, where t is the state corresponding to the string P . Answering a query again takes only $O(length(P))$ time.

All occurrence positions

This time we have to display all positions of the occurrences in the string T .

Again we construct a suffix automaton for the text T . Similar as in the previous task we compute the position $firstpos$ for all states.

Clearly $firstpos(t)$ is part of the answer, if t is the state corresponding to a query string P . So we took into account the state of the automaton containing P . What other states do we need to take into account? All states that correspond to strings for which P is a suffix. In other words we need to find all the states that can reach the state t via suffix links.

Therefore to solve the problem we need to save for each state a list of suffix references leading to it. The answer to the query then will then contain all $firstpos$ for each state that we can find on a DFS / BFS starting from the state t using only the suffix references.

This workaround will work in time $O(answer(P))$, because we will not visit a state twice (because only one suffix link leaves each state, so there cannot be two different paths leading to the same state).

We only must take into account that two different states can have the same $firstpos$ value. This happens if one state was obtained by cloning another. However, this doesn't ruin the complexity, since each state can only have at most one clone.

Moreover, we can also get rid of the duplicate positions, if we don't output the positions from the cloned states. In fact a state, that a cloned state can

reach, is also reachable from the original state. Thus if we remember the flag `is_cloned` for each state, we can simply ignore the cloned states and only output *firstpos* for all other states.

Here are some implementation sketches:

```
struct state {
    ...
    bool is_clone;
    int first_pos;
    vector<int> inv_link;
};

// after constructing the automaton
for (int v = 1; v < sz; v++) {
    st[st[v].link].inv_link.push_back(v);
}

// output all positions of occurrences
void output_all_occurrences(int v, int P_length) {
    if (!st[v].is_clone)
        cout << st[v].first_pos - P_length + 1 << endl;
    for (int u : st[v].inv_link)
        output_all_occurrences(u, P_length);
}
```

Shortest non-appearing string

Given a string S and a certain alphabet. We have to find a string of smallest length, that doesn't appear in S .

We will apply dynamic programming on the suffix automaton built for the string S .

Let $d[v]$ be the answer for the node v , i.e. we already processed part of the substring, are currently in the state v , and want to find the smallest number of characters that have to be added to find a non-existent transition. Computing $d[v]$ is very simple. If there is not transition using at least one character of the alphabet, then $d[v] = 1$. Otherwise one character is not enough, and so we need to take the minimum of all answers of all transitions:

$$d[v] = 1 + \min_{w:(v,w,c) \in SA} d[w].$$

The answer to the problem will be $d[t_0]$, and the actual string can be restored using the computed array $d[]$.

Longest common substring of two strings

Given two strings S and T . We have to find the longest common substring, i.e. such a string X that appears as substring in S and also in T .

We construct a suffix automaton for the string S .

We will now take the string T , and for each prefix look for the longest suffix of this prefix in S . In other words, for each position in the string T , we want to find the longest common substring of S and T ending in that position.

For this we will use two variables, the **current state** v , and the **current length** l . These two variables will describe the current matching part: its length and the state that corresponds to it.

Initially $v = t_0$ and $l = 0$, i.e. the match is empty.

Now let us describe how we can add a character $T[i]$ and recalculate the answer for it.

- If there is a transition from v with the character $T[i]$, then we simply follow the transition and increase l by one.
- If there is no such transition, we have to shorten the current matching part, which means that we need to follow the suffix link:

$$v = \text{link}(v)$$

At the same time, the current length has to be shortened. Obviously we need to assign $l = \text{len}(v)$, since after passing through the suffix link we end up in state whose corresponding longest string is a substring.

- If there is still no transition using the required character, we repeat and again go through the suffix link and decrease l , until we find a transition or we reach the fictional state -1 (which means that the symbol $T[i]$ doesn't appear at all in S , so we assign $v = l = 0$).

The answer to the task will be the maximum of all the values l .

The complexity of this part is $O(\text{length}(T))$, since in one move we can either increase l by one, or make several passes through the suffix links, each one ends up reducing the value l .

Implementation:

```
string lcs (string S, string T) {
    sa_init();
    for (int i = 0; i < S.size(); i++)
        sa_extend(S[i]);

    int v = 0, l = 0, best = 0, bestpos = 0;
    for (int i = 0; i < T.size(); 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);
}

```

Largest common substring of multiple strings

There are k strings S_i given. We have to find the longest common substring, i.e. such a string X that appears as substring in each string S_i .

We join all strings into one large string T , separating the strings by a special characters D_i (one for each string):

$$T = S_1 + D_1 + S_2 + D_2 + \dots + S_k + D_k.$$

Then we construct the suffix automaton for the string T .

Now we need to find a string in the machine, which is contained in all the strings S_i , and this can be done by using the special added characters. Note that if a substring is included in some string S_j , then in the suffix automaton exists a path starting from this substring containing the character D_j and not containing the other characters $D_1, \dots, D_{j-1}, D_{j+1}, \dots, D_k$.

Thus we need to calculate the attainability, which tells us for each state of the machine and each symbol D_i if there exists such a path. This can easily be computed by DFS or BFS and dynamic programming. After that, the answer to the problem will be the string $longest(v)$ for the state v , from which the paths were exists for all special characters.

2.2.5 Practice Problems

- [SPOJ - SUBLEX](#)

2.3 Lyndon factorization

2.3.1 Lyndon factorization

First let us define the notion of the Lyndon factorization.

A string is called **simple** (or a Lyndon word), if it is strictly **smaller than** any of its own nontrivial **suffixes**. Examples of simple strings are: a , b , ab , aab , abb , $ababb$, $abcd$. It can be shown that a string is simple, if and only if it is strictly **smaller than** all its nontrivial **cyclic shifts**.

Next, let there be a given string s . The **Lyndon factorization** of the string s is a factorization $s = w_1 w_2 \dots w_k$, where all strings w_i are simple, and they are in non-increasing order $w_1 \geq w_2 \geq \dots \geq w_k$.

It can be shown, that for any string such a factorization exists and that it is unique.

2.3.2 Duval algorithm

The Duval algorithm constructs the Lyndon factorization in $O(n)$ time using $O(1)$ additional memory.

First let us introduce another notion: a string t is called **pre-simple**, if it has the form $t = ww \dots w\bar{w}$, where w is a simple string and \bar{w} is a prefix of w (possibly empty). A simple string is also pre-simple.

The Duval algorithm is greedy. At any point during its execution, the string s will actually be divided into three strings $s = s_1 s_2 s_3$, where the Lyndon factorization for s_1 is already found and finalized, the string s_2 is pre-simple (and we know the length of the simple string in it), and s_3 is completely untouched. In each iteration the Duval algorithm takes the first character of the string s_3 and tries to append it to the string s_2 . If s_2 is no longer pre-simple, then the Lyndon factorization for some part of s_2 becomes known, and this part goes to s_1 .

Let's describe the algorithm in more detail. The pointer i will always point to the beginning of the string s_2 . The outer loop will be executed as long as $i < n$. Inside the loop we use two additional pointers, j which points to the beginning of s_3 , and k which points to the current character that we are currently comparing to. We want to add the character $s[j]$ to the string s_2 , which requires a comparison with the character $s[k]$. There can be three different cases:

- $s[j] = s[k]$: if this is the case, then adding the symbol $s[j]$ to s_2 doesn't violate its pre-simplicity. So we simply increment the pointers j and k .
- $s[j] > s[k]$: here, the string $s_2 + s[j]$ becomes simple. We can increment j and reset k back to the beginning of s_2 , so that the next character can be compared with the beginning of the simple word.
- $s[j] < s[k]$: the string $s_2 + s[j]$ is no longer pre-simple. Therefore we will split the pre-simple string s_2 into its simple strings and the remainder, possibly empty. The simple string will have the length $j - k$. In the next iteration we start again with the remaining s_2 .

Implementation

Here we present the implementation of the Duval algorithm, which will return the desired Lyndon factorization of a given string s .

```
vector<string> duval(string const& s) {
    int n = s.size();
    int i = 0;
    vector<string> factorization;
    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) {
            factorization.push_back(s.substr(i, j - k));
            i += j - k;
        }
    }
    return factorization;
}
```

Complexity

Let us estimate the running time of this algorithm.

The **outer while loop** does not exceed n iterations, since at the end of each iteration i increases. Also the second inner while loop runs in $O(n)$, since it only outputs the final factorization.

So we are only interested in the **first inner while loop**. How many iterations does it perform in the worst case? It's easy to see that the simple words that we identify in each iteration of the outer loop are longer than the remainder that we additionally compared. Therefore also the sum of the remainders will be smaller than n , which means that we only perform at most $O(n)$ iterations of the first inner while loop. In fact the total number of character comparisons will not exceed $4n - 3$.

2.3.3 Finding the smallest cyclic shift

Let there be a string s . We construct the Lyndon factorization for the string $s + s$ (in $O(n)$ time). We will look for a simple string in the factorization, which starts at a position less than n (i.e. it starts in the first instance of s), and ends in a position greater than or equal to n (i.e. in the second instance) of s). It is stated, that the position of the start of this simple string will be the beginning of the desired smallest cyclic shift. This can be easily verified using the definition of the Lyndon decomposition.

The beginning of the simple block can be found easily - just remember the pointer i at the beginning of each iteration of the outer loop, which indicated the beginning of the current pre-simple string.

So we get the following implementation:

```
string min_cyclic_string(string s) {
    s += s;
    int n = s.size();
    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);
}
```

2.3.4 Problems

- [UVA #719 - Glass Beads](#)

Chapter 3

Tasks

3.1 Expression parsing

A string containing a mathematical expression containing numbers and various operators is given. We have to compute the value of it in $O(n)$, where n is the length of the string.

The algorithm discussed here translates an expression into the so-called **reverse Polish notation** (explicitly or implicitly), and evaluates this expression.

3.1.1 Reverse Polish notation

The reverse Polish notation is a form of writing mathematical expressions, in which the operators are located after their operands. For example the following expression

$$a + b * c * d + (e - f) * (g * h + i)$$

can be written in reverse Polish notation in the following way:

$$abc * d * + ef - gh * i + * +$$

The reverse Polish notation was developed by the Australian philosopher and computer science specialist Charles Hamblin in the mid 1950s on the basis of the Polish notation, which was proposed in 1920 by the Polish mathematician Jan Łukasiewicz.

The convenience of the reverse Polish notation is, that expressions in this form are very **easy to evaluate** in linear time. We use a stack, which is initially empty. We will iterate over the operands and operators of the expression in reverse Polish notation. If the current element is a number, then we put the value on top of the stack, if the current element is an operator, then we get the top two elements from the stack, perform the operation, and put the result back on top of the stack. In the end there will be exactly one element left in the stack, which will be the value of the expression.

Obviously this simple evaluation runs in $O(n)$ time.

3.1.2 Parsing of simple expressions

For the time being we only consider a simplified problem: we assume that all operators are **binary** (i.e. they take two arguments), and all are **left-associative** (if the priorities are equal, they get executed from left to right). Parentheses are allowed.

We will set up two stacks: one for numbers, and one for operators and parentheses. Initially both stacks are empty. For the second stack we will maintain the condition that all operations are ordered by strict descending priority. If there are parenthesis on the stack, than each block of operators (corresponding to one pair of parenthesis) is ordered, and the entire stack is not necessarily ordered.

We will iterate over the characters of the expression from left to right. If the current character is a digit, then we put the value of this number on the stack. If the current character is an opening parenthesis, then we put it on the stack. If the current character is a closing parenthesis, then we execute all operators on the stack until we reach the opening bracket (in other words we perform all operations inside the parenthesis). Finally if the current character is an operator, then while the top of the stack has an operator with the same or higher priority, we will execute this operation, and put the new operation on the stack.

After we processed the entire string, some operators might still be in the stack, so we execute them.

Here is the implementation of this method for the four operators $+$ $-$ $*$ $/$:

```
bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

int priority (char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    int r = st.top(); st.pop();
    int l = st.top(); st.pop();
    switch (op) {
        case '+': st.push(l + r); break;
        case '-': st.push(l - r); break;
        case '*': st.push(l * r); break;
        case '/': st.push(l / r); break;
    }
}
```

```

int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))
                number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
        }
    }

    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}

```

Thus we learned how to calculate the value of an expression in $O(n)$, at the same time we implicitly used the reverse Polish notation. By slightly modifying the above implementation it is also possible to obtain the expression in reverse Polish notation in an explicit form.

3.1.3 Unary operators

Now suppose that the expression also contains **unary** operators (operators that take one argument). The unary plus and unary minus are common common examples for such operators.

One of the differences in this case, is that we need to determine whether the current operator is a unary or a binary one.

You can notice, that before an unary operator, there always is another operator or an opening parenthesis, or nothing at all (if it is at the very beginning of the expression). On the contrary before a binary operator there will always be an operand (number) or a closing parenthesis. Thus it is easy to flag whether the next operator can be unary or not.

Additionally we need to execute a unary and a binary operator differently. And we need to chose the priority of a binary operator higher than all of the binary operations.

In addition it should be noted, that some unary operators (e.g. unary plus and unary minus) are actually **right-associative**.

3.1.4 Right-associativity

Right-associative means, that whenever the priorities are equal, the operators must be evaluated from right to left.

As noted above, unary operators are usually right-associative. Another example for an right-associative operator is the exponentiation operator ($a \wedge b \wedge c$ is usually perceived as a^{b^c} and not as $(a^b)^c$).

What difference do we need to make in order to correctly handle right-associative operators? It turns out that the changes are very minimal. The only difference will be, if the priorities are equal we will postpone the execution of the right-associative operation.

The only line that needs to be replaced is

```
while (!op.empty() && priority(op.top()) >= priority(cur_op))
```

with

```
while (!op.empty() && (
    (left_assoc(cur_op) && priority(op.top()) >= priority(cur_op)) ||
    (!left_assoc(cur_op) && priority(op.top()) > priority(cur_op))
))
```

where `left_assoc` is a function that decides if an operator is left_associative or not.

Here is an implementation for the binary operators $+$ $-$ $*$ $/$ and the unary operators $+$ and $-$.

```
bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

bool is_unary(char c) {
    return c == '+' || c == '-';
}
```

```

int priority (char op) {
    if (op < 0) // unary operator
        return 3;
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    if (op < 0) {
        int l = st.top(); st.pop();
        switch (-op) {
            case '+': st.push(l); break;
            case '-': st.push(-l); break;
        }
    } else {
        int r = st.top(); st.pop();
        int l = st.top(); st.pop();
        switch (op) {
            case '+': st.push(l + r); break;
            case '-': st.push(l - r); break;
            case '*': st.push(l * r); break;
            case '/': st.push(l / r); break;
        }
    }
}

int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    bool may_be_unary = true;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
            may_be_unary = true;
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
            may_be_unary = false;
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            if (may_be_unary && is_unary(cur_op))
                cur_op = -cur_op;
        }
    }
}

```



```
        while (!op.empty() && (
            (cur_op >= 0 && priority(op.top()) >= priority(cur_op)) ||
            (cur_op < 0 && priority(op.top()) > priority(cur_op))
        )) {
            process_op(st, op.top());
            op.pop();
        }
        op.push(cur_op);
        may_be_unary = true;
    } else {
        int number = 0;
        while (i < (int)s.size() && isalnum(s[i]))
            number = number * 10 + s[i++] - '0';
        --i;
        st.push(number);
        may_be_unary = false;
    }
}

while (!op.empty()) {
    process_op(st, op.top());
    op.pop();
}
return st.top();
}
```

3.2 Finding all sub-palindromes in $O(N)$

3.2.1 Statement

Given string s with length n . Find all the pairs (i, j) such that substring $s[i \dots j]$ is a palindrome. String t is a palindrome when $t = t_{rev}$ (t_{rev} is a reversed string for t).

3.2.2 More precise statement

It's clear that in the worst case we can have $O(n^2)$ palindrome strings, and at the first glance it seems that there is no linear algorithm for this problem.

But the information about the palindromes can be kept **in a more compact way**: for each position $i = 0 \dots n - 1$ we'll find the values $d_1[i]$ and $d_2[i]$, denoting the number of palindromes accordingly with odd and even lengths with centers in the position i .

For instance, string $s = abababc$ has three palindromes with odd length with centers in the position $s[3] = b$, i. e. $d_1[3] = 3$:

$$\begin{array}{ccccccc} & & d_1[3]=3 & & & & \\ & & \overbrace{\hspace{1.5cm}} & & & & \\ a & b & a & b & a & b & c \\ & & \underbrace{\hspace{1.5cm}}_{s_3} & & & & \end{array}$$

And string $s = cbaabd$ has two palindromes with even length with centers in the position $s[3] = a$, i. e. $d_2[3] = 2$:

$$\begin{array}{ccccccc} & & d_2[3]=2 & & & & \\ & & \overbrace{\hspace{1.5cm}} & & & & \\ c & b & a & a & b & d & \\ & & \underbrace{\hspace{1.5cm}}_{s_3} & & & & \end{array}$$

So the idea is that if we have a sub-palindrome with length l with center in some position i , we also have sub-palindromes with lengths $l - 2$, $l - 4$ etc. with centers in i . So these two arrays $d_1[i]$ and $d_2[i]$ are enough to keep the information about all the sub-palindromes in the string.

It's a surprising fact that there is an algorithm, which is simple enough, that calculates these "palindromity arrays" $d_1[]$ and $d_2[]$ in linear time. The algorithm is described in this article.

3.2.3 Solution

In general, this problem has many solutions: with [String Hashing](#) it can be solved in $O(n \cdot \log n)$, and with [Suffix Trees](#) and fast LCA this problem can be solved in $O(n)$.

But the method described here is **sufficiently** simpler and has less hidden constant in time and memory complexity. This algorithm was discovered by **Glenn K. Manacher** in 1975.

3.2.4 Trivial algorithm

To avoid ambiguities in the further description we denote what "trivial algorithm" is.

It's the algorithm that does the following. For each center position i it tries to increase the answer by one until it's possible, comparing a pair of corresponding characters each time.

Such algorithm is slow, it can calculate the answer only in $O(n^2)$.

The implementation of the trivial algorithm is:

```
vector<int> d1(n), d2(n);
for (int i = 0; i < n; i++) {
    d1[i] = 1;
    while (0 <= i - d1[i] && i + d1[i] < n && s[i - d1[i]] == s[i + d1[i]]) {
        d1[i]++;
    }

    d2[i] = 0;
    while (0 <= i - d2[i] - 1 && i + d2[i] < n && s[i - d2[i] - 1] == s[i + d2[i]]) {
        d2[i]++;
    }
}
```

3.2.5 Manacher's algorithm

We describe the algorithm to find all the sub-palindromes with odd length, i. e. to calculate $d_1[]$; the solution for all the sub-palindromes with even length (i. e. calculating the array $d_2[]$) will be a minor modification for this one.

For fast calculation we'll keep the **borders** (l, r) of the rightmost found sub-palindrome (i. e. the palindrome with maximal r). Initially we assume $l = 0, r = -1$.

So, we want to calculate $d_1[i]$ for the next i , and all the previous values in $d_1[]$ have been already calculated. We do the following:

- If i is outside the current sub-palindrome, i. e. $i > r$, we'll just launch the trivial algorithm.

So we'll increase $d_1[i]$ consecutively and check each time if the current substring $[i - d_1[i] \dots i + d_1[i]]$ is a palindrome. When we find first divergence or meet the boundaries of s , we'll stop. In this case we've finally calculated $d_1[i]$. After this, we must not forget to update (l, r) .

- Now consider the case when $i \leq r$. We'll try to extract some information from the already calculated values in $d_1[]$. So, let's flip the position i inside the sub-palindrome (l, r) , i. e. we'll get the position $j = l + (r - i)$, and we'll look on the value $d_1[j]$. Because j is the position symmetrical to i , we'll **almost always** can assign $d_1[i] = d_1[j]$. Illustration of this (palindrome around j is actually "copied" into the palindrome around i):

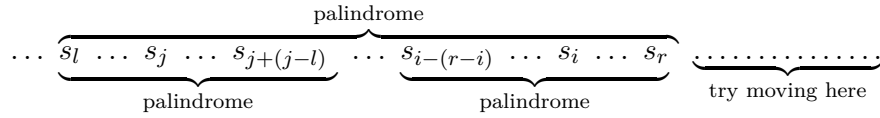
$$\dots \overbrace{s_l \dots s_{j-d_1[j]+1} \dots s_j \dots s_{j+d_1[j]-1} \dots s_{i-d_1[j]+1} \dots s_i \dots s_{i+d_1[j]-1} \dots s_r}^{\text{palindrome}} \dots$$

$$\underbrace{s_{j-d_1[j]+1} \dots s_j \dots s_{j+d_1[j]-1}}_{\text{palindrome}} \quad \underbrace{s_{i-d_1[j]+1} \dots s_i \dots s_{i+d_1[j]-1}}_{\text{palindrome}}$$

But there is a **tricky case** to be handled correctly: when the “inner” palindrome reaches the borders of the “outer” one, i. e. $j - d_1[j] + 1 \leq l$ (or, which is the same, $i + d_1[j] - 1 \geq r$). Because the symmetry outside the “outer” palindrome is not guaranteed, just assigning $d_1[i] = d_1[j]$ will be incorrect: we have not enough data to state that the palindrome in the position i has the same length.

Actually, we should “cut” the length of our palindrome, i. e. assign $d_1[i] = r - i$, to handle such situations correctly. After this we’ll run the trivial algorithm which will try to increase $d_1[i]$ while it’s possible.

Illustration of this case (the palindrome with center j is already “cut” to fit the “outer” palindrome):



It is shown on the illustration that, though the palindrome with center j could be larger and go outside the “outer” palindrome, in the position i we can use only the part that entirely fits into the “outer” palindrome. But the answer for the position i can be much longer than this part, so next we’ll run our trivial algorithm that will try to grow it outside our “outer” palindrome, i. e. to the region “try moving here”.

At the end, it’s necessary to remind that we should not forget to update the values (l, r) after calculating each $d_1[i]$.

Also we’ll repeat that the algorithm was described to calculate the array for odd palindromes $d_1[]$, the algorithm is similar for the array of even palindromes $d_2[]$.

3.2.6 Complexity of Manacher’s algorithm

At the first glance it’s not obvious that this algorithm has linear time complexity, because we often run the naive algorithm while searching the answer for a particular position.

But more careful analysis shows that the algorithm is linear however. We need to mention [Z-function building algorithm](#) which looks similar to this algorithm and also works in linear time.

Actually, we can notice that every iteration of trivial algorithm makes r increase by one. Also r cannot be decreased during the algorithm. So, trivial algorithm will make $O(n)$ iterations in total.

Also, other parts of Manacher’s algorithm work obviously in linear time, we get $O(n)$ time complexity.

3.2.7 Implementation of Manacher’s algorithm

For calculating $d_1[]$, we get the following code:

```

vector<int> d1(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 1 : min(d1[l + r - i], r - i);
    while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
        k++;
    }
    d1[i] = k--;
    if (i + k > r) {
        l = i - k;
        r = i + k;
    }
}

```

For calculating $d_2[]$, the code looks similar, but with minor changes in arithmetical expressions:

```

vector<int> d2(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
    while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
        k++;
    }
    d2[i] = k--;
    if (i + k > r) {
        l = i - k - 1;
        r = i + k;
    }
}

```

3.2.8 Problems

[UVA #11475 “Extend to Palindrome”](#)

3.3 Finding repetitions

Given a string s of length n .

A **repetition** is two occurrences of a string in a row. In other words a repetition can be described by a pair of indices $i < j$ such that the substring $s[i \dots j]$ consists of two identical strings written after each other.

The challenge is to **find all repetitions** in a given string s . Or a simplified task: find **any** repetition or find the **longest** repetition.

The algorithm described here was published in 1982 by Main and Lorentz.

3.3.1 Example

Consider the repetitions in the following example string:

acababae

The string contains the following three repetitions:

- $s[2 \dots 5] = abab$
- $s[3 \dots 6] = baba$
- $s[7 \dots 7] = ee$

Another example:

abaaba

Here there are only two repetitions

- $s[0 \dots 5] = abaaba$
- $s[2 \dots 3] = aa$

3.3.2 Number of repetitions

In general there can be up to $O(n^2)$ repetitions in a string of length n . An obvious example is a string consisting of n times the same letter, in this case any substring of even length is a repetition. In general any periodic string with a short period will contain a lot of repetitions.

On the other hand this fact does not prevent computing the number of repetitions in $O(n \log n)$ time, because the algorithm can give the repetitions in compressed form, in groups of several pieces at once.

There is even the concept, that describes groups of periodic substrings with tuples of size four. It has been proven that the number of such groups is at most linear with respect to the string length.

Also, here are some more interesting results related to the number of repetitions:

- The number of primitive repetitions (those whose halves are not repetitions) is at most $O(n \log n)$.

- If we encode repetitions with tuples of numbers (called Crochemore triples) (i, p, r) (where i is the position of the beginning, p the length of the repeating substring, and r the number of repetitions), then all repetitions can be described with $O(n \log n)$ such triples.
- Fibonacci strings, defined as

$$\begin{aligned} t_0 &= a, \\ t_1 &= b, \\ t_i &= t_{i-1} + t_{i-2}, \end{aligned}$$

are “strongly” periodic. The number of repetitions in the Fibonacci string f_i , even in the compressed with Crochemore triples, is $O(f_n \log f_n)$. The number of primitive repetitions is also $O(f_n \log f_n)$.

3.3.3 Main-Lorentz algorithm

The idea behind the Main-Lorentz algorithm is **divide-and-conquer**.

It splits the initial string into halves, and computes the number of repetitions that lie completely in each half by two recursive calls. Then comes the difficult part. The algorithm finds all repetitions starting in the first half and ending in the second half (which we will call **crossing repetitions**). This is the essential part of the Main-Lorentz algorithm, and we will discuss it in detail here.

The complexity of divide-and-conquer algorithms is well researched. The master theorem says, that we will end up with an $O(n \log n)$ algorithm, if we can compute the crossing repetitions in $O(n)$ time.

Search for crossing repetitions

So we want to find all such repetitions that start in the first half of the string, let's call it u , and end in the second half, let's call it v :

$$s = u + v$$

Their lengths are approximately equal to the length of s divided by two.

Consider an arbitrary repetition and look at the middle character (more precisely the first character of the second half of the repetition). I.e. if the repetition is a substring $s[i \dots j]$, then the middle character is $(i + j + 1)/2$.

We call a repetition **left** or **right** depending on which string this character is located - in the string u or in the string v . In other words a string is called left, if the majority of it lies in u , otherwise we call it right.

We will now discuss how to find **all left repetitions**. Finding all right repetitions can be done in the same way.

Let us denote the length of the left repetition by $2l$ (i.e. each half of the repetition has length l). Consider the first character of the repetition falling into the string v (it is at position $|u|$ in the string s). It coincides with the character l positions before it, let's denote this position *cntr*.

We will fixate this position *cntr*, and **look for all repetitions at this position *cntr***.

For example:

$$\begin{array}{c} c \quad a \quad c \mid a \quad d \quad a \\ \text{ctr} \end{array}$$

The vertical line divides the two halves. Here we fixated the position $ctr = 1$, and at this position we find the repetition $caca$.

It is clear, that if we fixate the position ctr , we simultaneously fixate the length of the possible repetitions: $l = |u| - ctr$. Once we know how to find these repetitions, we will iterate over all possible values for ctr from 0 to $|u| - 1$, and find all left crossover repetitions of length $l = |u|, |u| - 1, \dots, 1$.

Criterion for left crossing repetitions

Now, how can we find all such repetitions for a fixated ctr ? Keep in mind that there still can be multiple such repetitions.

Let's again look at a visualization, this time for the repetition $abcabc$:

$$\begin{array}{c} l_1 \quad l_2 \quad l_1 \quad l_2 \\ \underbrace{a} \quad \underbrace{b \quad c} \quad \underbrace{a} \mid \underbrace{b \quad c} \\ \text{ctr} \end{array}$$

Here we denoted the lengths of the two pieces of the repetition with l_1 and l_2 : l_1 is the length of the repetition up to the position $ctr - 1$, and l_2 is the length of the repetition from ctr to the end of the half of the repetition. We have $2l = l_1 + l_2 + l_1 + l_2$ as the total length of the repetition.

Let us generate **necessary and sufficient** conditions for such a repetition at position ctr of length $2l = 2(l_1 + l_2) = 2(|u| - ctr)$:

- Let k_1 be the largest number such that the first k_1 characters before the position ctr coincide with the last k_1 characters in the string u :

$$u[ctr - k_1 \dots ctr - 1] = u[|u| - k_1 \dots |u| - 1]$$

- Let k_2 be the largest number such that the k_2 characters starting at position ctr coincide with the first k_2 characters in the string v :

$$u[ctr \dots ctr + k_2 - 1] = v[0 \dots k_2 - 1]$$

- Then we have a repetition exactly for any pair (l_1, l_2) with

$$\begin{aligned} l_1 &\leq k_1, \\ l_2 &\leq k_2. \end{aligned}$$

To summarize:

- We fixate a specific position ctr .
- All repetition which we will find now have length $2l = 2(|u| - ctr)$. There might be multiple such repetitions, they depend on the lengths l_1 and $l_2 = l - l_1$.

- We find k_1 and k_2 as described above.
- Then all suitable repetitions are the ones for which the lengths of the pieces l_1 and l_2 satisfy the conditions:

$$\begin{aligned} l_1 + l_2 &= l = |u| - cntr \\ l_1 &\leq k_1, \\ l_2 &\leq k_2. \end{aligned}$$

Therefore the only remaining part is how we can compute the values k_1 and k_2 quickly for every position $cntr$. Luckily we can compute them in $O(1)$ using the [Z-function](#):

- To find the value k_1 for each position by calculating the Z-function for the string \bar{u} (i.e. the reversed string u). Then the value k_1 for a particular $cntr$ will be equal to the corresponding value of the array of the Z-function.
- To precompute all values k_2 , we calculate the Z-function for the string $v + \# + u$ (i.e. the string u concatenated with the separator character $\#$ and the string v). Again we just need to look up the corresponding value in the Z-function to get the k_2 value.

So this is enough to find all left crossing repetitions.

Right crossing repetitions

For computing the right crossing repetitions we act similarly: we define the center $cntr$ as the character corresponding to the last character in the string u .

Then the length k_1 will be defined as the largest number of characters before the position $cntr$ (inclusive) that coincide with the last characters of the string u . And the length k_2 will be defined as the largest number of characters starting at $cntr + 1$ that coincide with the characters of the string v .

Thus we can find the values k_1 and k_2 by computing the Z-function for the strings $\bar{u} + \# + \bar{v}$ and v .

After that we can find the repetitions by looking at all positions $cntr$, and use the same criterion as we had for left crossing repetitions.

Implementation

The implementation of the Main-Lorentz algorithm finds all repetitions in form of peculiar tuples of size four: $(cntr, l, k_1, k_2)$ in $O(n \log n)$ time. If you only want to find the number of repetitions in a string, or only want to find the longest repetition in a string, this information is enough and the runtime will still be $O(n \log n)$.

Notice that if you want to expand these tuples to get the starting and end position of each repetition, then the runtime will be the runtime will be $O(n^2)$ (remember that there can be $O(n^2)$ repetitions). In this implementation we will do so, and store all found repetition in a vector of pairs of start and end indices.

```

vector<int> z_function(string const& s) {
    int n = s.size();
    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-l]);
        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;
}

int get_z(vector<int> const& z, int i) {
    if (0 <= i && i < (int)z.size())
        return z[i];
    else
        return 0;
}

vector<pair<int, int>> repetitions;

void convert_to_repetitions(int shift, bool left, int cntr, int l, int k1, int k2) {
    for (int l1 = max(1, l - k2); l1 <= min(l, k1); l1++) {
        if (left && l1 == 1) break;
        int l2 = l - l1;
        int pos = shift + (left ? cntr - l1 : cntr - l - l1 + 1);
        repetitions.emplace_back(pos, pos + 2*l1 - 1);
    }
}

void find_repetitions(string s, int shift = 0) {
    int n = s.size();
    if (n == 1)
        return;

    int nu = n / 2;
    int nv = n - nu;
    string u = s.substr(0, nu);
    string v = s.substr(nu);
    string ru(u.rbegin(), u.rend());
    string rv(v.rbegin(), v.rend());

    find_repetitions(u, shift);
    find_repetitions(v, shift + nu);

    vector<int> z1 = z_function(ru);
    vector<int> z2 = z_function(v + '#' + u);
    vector<int> z3 = z_function(ru + '#' + rv);
}

```

```
vector<int> 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 + 1 + 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)
        convert_to_repetitions(shift, cntr < nu, cntr, l, k1, k2);
}
```

Part V

Linear Algebra

Chapter 1

Matrices

1.1 Gauss method for solving system of linear equations

Given a system of n linear algebraic equations (SLAE) with m unknowns. You are asked to solve the system: to determine if it has no solution, exactly one solution or infinite number of solutions. And in case it has at least one solution, find any of them.

Formally, the problem is formulated as follows: solve the system:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m &= b_2 \\ &\dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m &= b_n \end{aligned}$$

where the coefficients a_{ij} (for i from 1 to n , j from 1 to m) and b_i (i from 1 to n) are known and variables x_i (i from 1 to m) are unknowns.

This problem also has a simple matrix representation:

$$Ax = b$$

, where A is a matrix of size $n \times m$ of coefficients a_{ij} and b is the column vector of size n .

It is worth noting that the method presented in this article can also be used to solve the equation modulo any number p , i.e.:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m &\equiv b_1 \pmod{p} \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m &\equiv b_2 \pmod{p} \\ &\dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m &\equiv b_n \pmod{p} \end{aligned}$$

1.1.1 Gauss

Strictly speaking, the method described below should be called “Gauss-Jordan”, or Gauss-Jordan elimination, because it is a variation of the Gauss method, described by Jordan in 1887.

1.1.2 Overview

The algorithm is a **sequential elimination** of the variables in each equation, until each equation will have only one remaining variable. If $n = m$, you can think of it as transforming the matrix A to identity matrix, and solve the equation in this obvious case, where solution is unique and is equal to coefficient b_i .

Gaussian elimination is based on two simple transformation: * It is possible to exchange two equations * Any equation can be replaced by a linear combination of the that row (with non-zero coefficient), and some other rows (with arbitrary coefficients).

In the first step, Gauss-Jordan algorithm divides the first row by a_{11} . Then, the algorithm adds the first row to the remaining rows such that the coefficients in the first column becomes all zeros. To achieve this, on the i -th row, we must add the first row multiplied by $-a_{i1}$. Note that, this operation must also be performed on vector b . In a sense, it behaves as if vector b was the $m + 1$ -th column of matrix A .

As a result, after the first step, the first column of matrix A will consists of 1 on the first row, and 0 in other rows.

Similarly, we perform the second step of the algorithm, where we consider the second column of second row. First, the row is divided by a_{22} , then it is subtracted from other rows so that all the second column becomes 0 (except for the second row).

We continue this process for all columns of matrix A . If $n = m$, then A will become identity matrix.

1.1.3 Search for the pivoting element

The described scheme left out many details. At the i th step, if a_{ii} is zero, we cannot apply directly the described method. Instead, we must first **select a pivoting row**: find one row of the matrix where the i th column is non-zero, and then swap the two rows.

Note that, here we swap rows but not columns. This is because if you swap columns, then when you find a solution, you must remember to swap back to correct places. Thus, swapping rows is much easier to do.

In many implementations, when $a_{ii} \neq 0$, you can see people still swap the i th row with some pivoting row, using some heuristics such as choosing the pivoting row with maximum absolute value of a_{ji} . This heuristic is used to reduce the value range of the matrix in later step. Without this heuristic, even for matrices of size about 20, the error will be too big and can cause overflow for floating points data types of C++.

1.1.4 Degenerate cases

In the case where $m = n$ and the system is degenerate (i.e. it has non-zero determinant, and has unique solution), the algorithm described above will transform A into identity matrix.

Now we consider the **general case**, where n and m are not necessarily equal, and the system can be non-degenerate. In these cases, the pivoting element in i th step may not be found. This means that on the i th column, starting from the current line, all contains zeros. In this case, either there is no possible value of variable x_i (meaning the SLAE has no solution), or x_i is an independent variable and can take arbitrary value. When implementing Gauss-Jordan, you should continue the work for subsequent variables and just skip the i th column (this is equivalent to removing the i th column of the matrix).

So, some of the variables in the process can be found to be independent. When the number of variables, m is greater than the number of equations, n , then at least $m - n$ independent variables will be found.

In general, if you find at least on independent variable, it can take any arbitrary value, while the other (dependent) variables are expressed through it. This means that when we work in the field of real numbers, the system potentially has infinitely many solutions. But you should remember that when there are independent variables, SLAE can have no solution at all. This happens when the remaining untreated equations has at least one non-zero constant term. You can check this by assigning zeros to all independent variables, calculate other variables, and then plug in to the original SLAE to check if they satisfies.

1.1.5 Implementation

Following is an implementation of Gauss-Jordan. Choosing the pivot row is done with heuristic: choosing maximum value in the current column.

The input to the function `gauss` is the system matrix a . The last column of this matrix is vector b .

The function returns the number of solutions of the system (0, 1, or ∞). If at least one solution exists, then it is returned in 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;
}

```

Implementation notes:

- The function uses two pointers - the current column *col* and the current row *row*.
- For each variable x_i , the value $where(i)$ is the line where this column is not zero. This vector is needed because some variables can be independent.
- In this implementation, the current *i*th line is not divided by a_{ii} as described above, so in the end the matrix is not identity matrix (though apparently dividing the *i*th line can help reducing errors).
- After finding a solution, it is inserted back into the matrix - to check whether the system has at least one solution or not. If the test solution is successful, then the function returns 1 or inf, depending on whether there is at least one independent variable.

1.1.6 Complexity

Now we should estimate the complexity of this algorithm. The algorithm consists of m phases, in each phase:

- Search and reshuffle the pivoting row. This takes $O(n + m)$ when using heuristic mentioned above.
- If there reference element in the current column is found - then we must add this equation to all other equations, which takes time $O(nm)$.

So, the final complexity of the algorithm is $O(\min(n, m).nm)$. In case $n = m$, the complexity is simply $O(n^3)$.

Note that when the SLAE is not on real numbers, but is in the modulo two, then the system can be solved much faster, which is described below.

1.1.7 Acceleration of the algorithm

The previous implementation can be speed up by two times, by dividing the algorithm into two phases: forward and reverse:

- Forward phase: Similar to the previous implementation, but the current row is only added to the rows after it. As a result, we obtain a triangular matrix instead of diagonal.
- Reverse phase: When the matrix is triangular, we first calculate the value of the last variable. Then plug this value to find the value of next variable. Then plug these two values to find the next variables...

Reverse phase only takes $O(nm)$, which is much faster than forward phase. In forward phase, we reduce the number of operations by half, thus reducing the running time of the implementation.

1.1.8 Solving SLAE

For solving SLAE in some module, we can still use the described algorithm. However, in case the module is equal to two, we can perform Gauss-Jordan elimination much more effectively using bitwise operations and C++ bitset data types:

```
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;
    }
    // The rest of implementation is the same as above
}
```

Since we use bit compress, the implementation is not only shorter, but also 32 times faster.

1.1.9 A little note about different heuristics of choosing pivoting row

There is no general rule for what heuristics to use.

The heuristics used in previous implementation works quite well in practice. It also turns out to give almost the same answers as “full pivoting” - where the pivoting row is search amongst all elements of the whose submatrix (from the current row and current column).

Though, you should note that both heuristics is dependent on how much the original equations was scaled. For example, if one of the equation was multiplied by 10^6 , then this equation is almost certain to be chosen as pivot in first step. This seems rather strange, so it seems logical to change to a more complicated heuristics, called **implicit pivoting**.

Implicit pivoting compares elements as if both lines were normalized, so that the maximum element would be unity. To implement this technique, one need to maintain maximum in each row (or maintain each line so that maximum is unity, but this can lead to increase in the accumulated error).

1.1.10 Improve the solution

Despite various heuristics, Gauss-Jordan algorithm can still lead to large errors in special matrices even of size 50 – 100.

Therefore, the resulting Gauss-Jordan solution must sometimes be improved by applying a simple numerical method - for example, the method of simple iteration.

Thus, the solution turns into two-step: First, Gauss-Jordan algorithm is applied, and then a numerical method taking initial solution as solution in the first step.

1.1.11 Practice Problems

- [Spoj - Xor Maximization](#)
- [Codechef - Knight Moving](#)
- [Lightoj - Graph Coloring](#)
- [UVA 12910 - Snakes and Ladders](#)

1.2 Calculating the determinant of a matrix by Gauss

Problem: Given a matrix A of size $N \times N$. Compute its determinant.

1.2.1 Algorithm

We use the ideas of [Gauss method for solving systems of linear equations](#)

We will perform the same steps as in the solution of systems of linear equations, excluding only the division of the current line to a_{ij} . These operations will not change the absolute value of the determinant of the matrix. When we exchange two lines of the matrix, however, the sign of the determinant can change.

After applying Gauss on the matrix, we receive a diagonal matrix, whose determinant is just the product of the elements on the diagonal. The sign, as previously mentioned, can be determined by the number of exchanged rows (if odd, then the sign of the determinant should be reversed). Thus, we can use the Gauss algorithm to compute the determinant of the matrix in complexity $O(N^3)$.

It should be noted that if at some point, we do not find non-zero cell in current column, the algorithm should stop and returns 0.

1.2.2 Implementation

```
const double EPS = 1E-9;
int n;
vector < vector<double> > a (n, vector<double> (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;
```

1.2.3 Practice Problems

- [Codeforces - Wizards and Bets](#)

1.3 Calculating the determinant using Kraut method in $O(N^3)$

In this article, we'll describe how to find the determinant of the matrix using Kraut method, which works in $O(N^3)$.

The Kraut algorithm finds decomposition of matrix A as $A = LU$ where L is lower triangular and U is upper triangular matrix. Without loss of generality, we can assume that all the diagonal elements of L are equal to 1. Once we know these matrices, it is easy to calculate the determinant of A : it is equal to the product of all the elements on the main diagonal of the matrix U .

There is a theorem stating that any invertible matrix has a LU-decomposition, and it is unique, if and only if all its principle minors are non-zero. We consider only such decomposition in which the diagonal of matrix L consists of ones.

Let A be the matrix and N - its size. We will find the elements of the matrices L and U using the following steps:

1. Let $L_{ii} = 1$ for $i = 1, 2, \dots, N$.
2. For each $j = 1, 2, \dots, N$ perform:
 - For $i = 1, 2, \dots, j$ find values

$$U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} \cdot U_{kj}$$

- Next, for $i = j + 1, j + 2, \dots, N$ find values

$$L_{ij} = \frac{1}{U_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} \cdot U_{kj} \right)$$

1.3.1 Implementation

```
static BigInteger det (BigDecimal a [] [], int n) {
    try {

        for (int i=0; i<n; i++) {
            boolean nonzero = false;
            for (int j=0; j<n; j++)
                if (a[i][j].compareTo (new BigDecimal (BigInteger.ZERO)) > 0)
                    nonzero = true;
            if (!nonzero)
                return BigInteger.ZERO;
        }

        BigDecimal scaling [] = new BigDecimal [n];
        for (int i=0; i<n; i++) {
            BigDecimal big = new BigDecimal (BigInteger.ZERO);
            for (int j=0; j<n; j++)
```

```

        if (a[i][j].abs().compareTo (big) > 0)
            big = a[i][j].abs();
        scaling[i] = (new BigDecimal (BigInteger.ONE)) .divide
            (big, 100, BigDecimal.ROUND_HALF_EVEN);
    }

    int sign = 1;

    for (int j=0; j<n; j++) {
        for (int i=0; i<j; i++) {
            BigDecimal sum = a[i][j];
            for (int k=0; k<i; k++)
                sum = sum.subtract (a[i][k].multiply (a[k][j]));
            a[i][j] = sum;
        }

        BigDecimal big = new BigDecimal (BigInteger.ZERO);
        int imax = -1;
        for (int i=j; i<n; i++) {
            BigDecimal sum = a[i][j];
            for (int k=0; k<j; k++)
                sum = sum.subtract (a[i][k].multiply (a[k][j]));
            a[i][j] = sum;
            BigDecimal cur = sum.abs();
            cur = cur.multiply (scaling[i]);
            if (cur.compareTo (big) >= 0) {
                big = cur;
                imax = i;
            }
        }

        if (j != imax) {
            for (int k=0; k<n; k++) {
                BigDecimal t = a[j][k];
                a[j][k] = a[imax][k];
                a[imax][k] = t;
            }

            BigDecimal t = scaling[imax];
            scaling[imax] = scaling[j];
            scaling[j] = t;

            sign = -sign;
        }

        if (j != n-1)
            for (int i=j+1; i<n; i++)
                a[i][j] = a[i][j].divide
                    (a[j][j], 100, BigDecimal.ROUND_HALF_EVEN);
    }

```

```
        BigDecimal result = new BigDecimal (1);
        if (sign == -1)
            result = result.negate();
        for (int i=0; i<n; i++)
            result = result.multiply (a[i][i]);

        return result.divide
            (BigDecimal.valueOf(1), 0, BigDecimal.ROUND_HALF_EVEN).toBigInteger();
    }
    catch (Exception e) {
        return BigInteger.ZERO;
    }
}
```

1.4 Finding the rank of a matrix

The rank of a matrix is the largest number of linearly independent rows/columns of the matrix. The rank is not only defined for square matrices.

The rank of a matrix can also be defined as the largest order of any non-zero minor in the matrix.

Let the matrix be rectangular and have size $N \times M$. Note that if the matrix is square and its determinant is non-zero, then the rank is $N (= M)$; otherwise it will be less. Generally, the rank of a matrix does not exceed $\min(N, M)$.

1.4.1 Algorithm

You can search for the rank using [Gaussian elimination](#). We will perform the same operations as when solving the system or finding its determinant. But if at any step in the i -th column there are no rows with a non-empty entry among those that we didn't selected already, then we skip this step and decrease the rank by one (initially the rank is set equal to $\max(N, M)$). Otherwise, if we have found a row with a non-zero element in the i -th column during the i -th step, then we mark this row as a selected one and perform the usual operations of taking this row away from the rest.

1.4.2 Complexity

This algorithm runs in $\mathcal{O}(n^3)$.

1.4.3 Implementation

```
const double EPS = 1E-9;

int compute_rank(vector<vector<int>> A) {
    int n = A.size();
    int m = A[0].size();

    int rank = max(n, m);
    vector<bool> row_selected(n, false);
    for (int i = 0; i < m; ++i) {
        int j;
        for (j = 0; j < n; ++j) {
            if (!row_selected[j] && abs(A[j][i]) > EPS)
                break;
        }

        if (j == n) {
            --rank;
        } else {
            row_selected[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];
    }
}
return rank;
}
```

Part VI

Combinatorics

Chapter 1

Fundamentals

1.1 Finding Power of Factorial Divisor

You are given two numbers n and k . Find the largest power of k x such that $n!$ is divisible by k^x .

1.1.1 Prime k

Let's first consider the case of prime k . The explicit expression for factorial

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

Note that every k -th element of the product is divisible by k , i.e. adds +1 to the answer; the number of such elements is $\left\lfloor \frac{n}{k} \right\rfloor$.

Next, every k^2 -th element is divisible by k^2 , i.e. adds another +1 to the answer (the first power of k has already been counted in the previous paragraph). The number of such elements is $\left\lfloor \frac{n}{k^2} \right\rfloor$.

And so on, for every i each k^i -th element adds another +1 to the answer, and there are $\left\lfloor \frac{n}{k^i} \right\rfloor$ such elements.

The final answer is

$$\left\lfloor \frac{n}{k} \right\rfloor + \left\lfloor \frac{n}{k^2} \right\rfloor + \dots + \left\lfloor \frac{n}{k^i} \right\rfloor + \dots$$

The sum is of course finite, since only approximately the first $\log_k n$ elements are not zeros. Thus, the runtime of this 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;  
}
```

1.1.2 Composite k

The same idea can't be applied directly. Instead we can factor k , representing it as $k = k_1^{p_1} \cdot \dots \cdot k_m^{p_m}$. For each k_i , we find the number of times it is present in $n!$ using the algorithm described above - let's call this value a_i . The answer for composite k will be

$$\min_{i=1\dots m} \frac{a_i}{p_i}$$

1.2 Binomial Coefficients

Binomial coefficients $\binom{n}{k}$ are the number of ways to select a set of k elements from n different elements without taking into account the order of arrangement of these elements (i.e., the number of unordered sets).

Binomial coefficients are also the coefficients in the expansion of $(a + b)^n$ (so-called binomial theorem):

$$(a + b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{k}a^{n-k}b^k + \cdots + \binom{n}{n}b^n$$

It is believed that this formula, as well as the triangle which allows efficient calculation of the coefficients, was discovered by Blaise Pascal in the 17th century. Nevertheless, it was known to the Chinese mathematician Yang Hui, who lived in the 13th century. Perhaps it was discovered by a Persian scholar Omar Khayyam. Moreover, Indian mathematician Pingala, who lived earlier in the 3rd. BC, got similar results. The merit of the Newton is that he generalized this formula for exponents that are not natural.

1.2.1 Calculation

Analytic formula for the calculation:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This formula can be easily deduced from the problem of ordered arrangement (number of ways to select k different elements from n different elements). First, let's count the number of ordered selections of k elements. There are n ways to select the first element, $n - 1$ ways to select the second element, $n - 2$ ways to select the third element, and so on. As a result, we get the formula of the number of ordered arrangements: $n(n-1)(n-2) \cdots (n-k+1) = \frac{n!}{(n-k)!}$. We can easily move to unordered arrangements, noting that each unordered arrangement corresponds to exactly $k!$ ordered arrangements ($k!$ is the number of possible permutations of k elements). We get the final formula by dividing $\frac{n!}{(n-k)!}$ by $k!$.

Recurrence formula (which is associated with the famous “Pascal’s Triangle”):

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

It is easy to deduce this using the analytic formula.

Note that for $n < k$ the value of $\binom{n}{k}$ is assumed to be zero.

1.2.2 Properties

Binomial coefficients have many different properties. Here are the simplest of them:

- Symmetry rule:

$$\binom{n}{k} = \binom{n}{n-k}$$

- Factoring in:

$$\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}$$

- Sum over k :

$$\sum_{k=0}^n \binom{n}{k} = 2^n$$

- Sum over n :

$$\sum_{m=0}^n \binom{m}{k} = \binom{n+1}{k+1}$$

- Sum over n and k :

$$\sum_{k=0}^m \binom{n+k}{k} = \binom{n+m+1}{m}$$

- Sum of the squares:

$$\binom{n}{0}^2 + \binom{n}{1}^2 + \cdots + \binom{n}{n}^2 = \binom{2n}{n}$$

- Weighted sum:

$$1 \binom{n}{1} + 2 \binom{n}{2} + \cdots + n \binom{n}{n} = n 2^{n-1}$$

- Connection with the [Fibonacci numbers](#):

$$\binom{n}{0} + \binom{n-1}{1} + \cdots + \binom{n-k}{k} + \cdots + \binom{0}{n} = F_{n+1}$$

1.2.3 Calculation

Straightforward calculation using analytical formula

The first, straightforward formula is very easy to code, but this method is likely to overflow even for relatively small values of n and k (even if the answer completely fit into some datatype, the calculation of the intermediate factorials can lead to overflow). Therefore, this method often can only be used with [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

Note that in the above implementation numerator and denominator have the same number of factors (k), each of which is greater than or equal to 1. Therefore, we can replace our fraction with a product k fractions, each of which is real-valued. However, on each step after multiplying current answer by each of the next fractions the answer will still be integer (this follows from the property of factoring in). C++ implementation:

```
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 carefully cast the floating point number to an integer, taking into account that due to the accumulated errors, it may be slightly less than the true value (for example, 2.99999 instead of 3).

Pascal's Triangle

By using the recurrence relation we can construct a table of binomial coefficients (Pascal's triangle) and take the result from it. The advantage of this method is that intermediate results never exceed the answer and calculating each new table element requires only one addition. The flaw is slow execution for large n and k if you just need a single value and not the whole table (because in order to calculate $\binom{n}{k}$ you will need to build a table of all $\binom{i}{j}$, $1 \leq i \leq n$, $1 \leq j \leq n$, or at least to $1 \leq j \leq \min(i, 2k)$). The time complexity can be considered to be $\mathcal{O}(n^2)$. C++ implementation:

```
const int maxn = ...;
int C[maxn + 1][maxn + 1];
C[0][0] = 1;
for (int n = 1; 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, storing only two last rows of it is sufficient (current n -th row and the previous $n - 1$ -th).

Calculation in $\mathcal{O}(1)$

Finally, in some situations it is beneficial to precompute all the factorials in order to produce any necessary binomial coefficient with only two divisions later. This can be advantageous when using [long arithmetic](#), when the memory does not allow precomputation of the whole Pascal's triangle.

1.2.4 Computing binomial coefficients modulo m .

Quite often you come across the problem of computing binomial coefficients modulo some m .

Binomial coefficient for small n

The previously discussed approach of Pascal's triangle can be used to calculate all values of $\binom{n}{k} \bmod m$ for reasonably small n , since it requires time complexity $\mathcal{O}(n^2)$. This approach can handle any modulo, since only addition operations are used.

Binomial coefficient modulo large prime

The formula for the binomial coefficients is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

so if we want to compute it modulo some prime $m > n$ we get

$$\binom{n}{k} \equiv n! \cdot (k!)^{-1} \cdot ((n-k)!)^{-1} \pmod{m}.$$

First we precompute all factorials modulo m up to $\text{MAXN}!$ in $O(\text{MAXN})$ time.

```
factorial[0] = 1;
for (int i = 1; i <= MAXN; i++) {
    factorial[i] = factorial[i - 1] * i % m;
}
```

And afterwards we can compute the binomial coefficient in $O(\log m)$ time.

```
long long binomial_coefficient(int n, int k) {
    return factorial[n] * inverse(factorial[k]) % m * inverse(factorial[n - k]) % m;
}
```

We even can compute the binomial coefficient in $O(1)$ time if we precompute the inverses of all factorials in $O(\text{MAXN} \log m)$ using the regular method for computing the inverse, or even in $O(\text{MAXN})$ time using the congruence $(x!)^{-1} \equiv ((x-1)!)^{-1} \cdot x^{-1}$ and the method for [computing all inverses](#) in $O(n)$.

Binomial coefficient modulo prime power

Here we want to compute the binomial coefficient modulo some prime power, i.e. $m = p^b$ for some prime p . If $p > \max(k, n-k)$, then we can use the same method as described in the previous section. But if $p \leq \max(k, n-k)$, then at least one of $k!$ and $(n-k)!$ are not coprime with m , and therefore we cannot compute the inverses - they don't exist. Nevertheless we can compute the binomial coefficient.

The idea is the following: We compute for each $x!$ the biggest exponent c such that p^c divides $x!$, i.e. $p^c \mid x!$. Let $c(x)$ be that number. And let $g(x) := \frac{x!}{p^{c(x)}}$. Then we can write the binomial coefficient as:

$$\binom{n}{k} = \frac{g(n)p^{c(n)}}{g(k)p^{c(k)}g(n-k)p^{c(n-k)}} = \frac{g(n)}{g(k)g(n-k)}p^{c(n)-c(k)-c(n-k)}$$

The interesting thing is, that $g(x)$ is now free from the prime divisor p . Therefore $g(x)$ is coprime to m , and we can compute the modular inverses of $g(k)$ and $g(n-k)$.

After precomputing all values for g and c , which can be done efficiently using dynamic programming in $\mathcal{O}(n)$, we can compute the binomial coefficient in $\mathcal{O}(\log m)$ time. Or precompute all inverses and all powers of p , and then compute the binomial coefficient in $\mathcal{O}(1)$.

Notice, if $c(n) - c(k) - c(n-k) \geq b$, then $p^b \mid p^{c(n)-c(k)-c(n-k)}$, and the binomial coefficient is 0.

Binomial coefficient modulo an arbitrary number

Now we compute the binomial coefficient modulo some arbitrary modulus m .

Let the prime factorization of m be $m = p_1^{e_1} p_2^{e_2} \cdots p_h^{e_h}$. We can compute the binomial coefficient modulo $p_i^{e_i}$ for every i . This gives us h different congruences. Since all moduli $p_i^{e_i}$ are coprime, we can apply the [Chinese Remainder Theorem](#) to compute the binomial coefficient modulo the product of the moduli, which is the desired binomial coefficient modulo m .

Binomial coefficient for large n and small modulo

When n is too large, the $\mathcal{O}(n)$ algorithms discussed above become impractical. However, if the modulo m is small there are still ways to calculate $\binom{n}{k} \bmod m$.

When the modulo m is prime, there are 2 options:

- [Lucas's theorem](#) can be applied which breaks the problem of computing $\binom{n}{k} \bmod m$ into $\log_m n$ problems of the form $\binom{x_i}{y_i} \bmod m$ where $x_i, y_i < m$. If each reduced coefficient is calculated using precomputed factorials and inverse factorials, the complexity is $\mathcal{O}(m + \log_m n)$.
- The method of computing [factorial modulo P](#) can be used to get the required g and c values and use them as described in the section of [modulo prime power](#). This takes $\mathcal{O}(m \log_m n)$.

When m is not prime but square-free, the prime factors of m can be obtained and the coefficient modulo each prime factor can be calculated using either of the above methods, and the overall answer can be obtained by the Chinese Remainder Theorem.

When m is not square-free, a [generalization of Lucas's theorem for prime powers](#) can be applied instead of Lucas's theorem.

1.2.5 Practice Problems

- [Codechef - Number of ways](#)
- [Codeforces - Curious Array](#)
- [LightOj - Necklaces](#)
- [HACKEREARTH: Binomial Coefficient](#)
- [SPOJ - Ada and Teams](#)
- [DevSkill - Drive In Grid](#)
- [SPOJ - Greedy Walking](#)
- [UVa 13214 - The Robot's Grid](#)
- [SPOJ - Good Predictions](#)
- [SPOJ - Card Game](#)
- [SPOJ - Topper Rama Rao](#)
- [UVa 13184 - Counting Edges and Graphs](#)
- [Codeforces - Anton and School 2](#)
- [DevSkill - Parandthesis](#)
- [Codeforces - Bacterial Melee](#)
- [Codeforces - Points, Lines and Ready-made Titles](#)
- [SPOJ - The Ultimate Riddle](#)
- [CodeChef - Long Sandwich](#)

1.2.6 References

- [Blog fishi.devtail.io](#)
- [Question on Mathematics StackExchange](#)
- [Question on CodeChef Discuss](#)

1.3 Catalan Numbers

Catalan numbers is a number sequence, which is found useful in a number of combinatorial problems, often involving recursively-defined objects.

This sequence was named after the Belgian mathematician [Catalan](#), who lived in the 19th century. (In fact it was known before to Euler, who lived a century before Catalan).

The first few numbers Catalan numbers, C_n (starting from zero):

1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

Application in some combinatorial problems

The Catalan number C_n is the solution for

- Number of correct bracket sequence consisting of n opening and n closing brackets.
- The number of rooted full binary trees with $n + 1$ leaves (vertices are not numbered). A rooted binary tree is full if every vertex has either two children or no children.
- The number of ways to completely parenthesize $n + 1$ factors.
- The number of triangulations of a convex polygon with $n + 2$ sides (i.e. the number of partitions of polygon into disjoint triangles by using the diagonals).
- The number of ways to connect the $2n$ points on a circle to form n disjoint chords.
- The number of [non-isomorphic](#) full binary trees with n internal nodes (i.e. nodes having at least one son).
- The number of monotonic lattice paths from point $(0, 0)$ to point (n, n) in a square lattice of size $n \times n$, which do not pass above the main diagonal (i.e. connecting $(0, 0)$ to (n, n)).
- Number of permutations of length n that can be [stack sorted](#) (i.e. it can be shown that the rearrangement is stack sorted if and only if there is no such index $i < j < k$, such that $a_k < a_i < a_j$).
- The number of [non-crossing partitions](#) of a set of n elements.
- The number of ways to cover the ladder $1 \dots n$ using n rectangles (The ladder consists of n columns, where i^{th} column has a height i).

1.3.1 Calculations

There are two formulas for the Catalan numbers: **Recursive and Analytical**. Since, we believe that all the mentioned above problems are equivalent (have the same solution), for the proof of the formulas below we will choose the task which it is easiest to do.

Recursive formula

$$C_0 = C_1 = 1$$

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}, n \geq 2$$

The recurrence formula can be easily deduced from the problem of the correct bracket sequence.

The leftmost opening parenthesis l corresponds to certain closing bracket r , which divides the sequence into 2 parts which in turn should be a correct sequence of brackets. Thus formula is also divided into 2 parts. If we denote $k = r - l - 1$, then for fixed r , there will be exactly $C_k C_{n-1-k}$ such bracket sequences. Summing this over all admissible k 's, we get the recurrence relation on C_n .

You can also think it in this manner. By definition, C_n denotes number of correct bracket sequences. Now, the sequence may be divided into 2 parts of length k and $n - k$, each of which should be a correct bracket sequence. Example :

$()(())$ can be divided into $()$ and $((()))$, but cannot be divided into $()()$ and $((()))$. Again summing over all admissible k 's, we get the recurrence relation on C_n .

C++ implementation

```
const int MOD = ....
const int MAX = ....
int catalan[MAX];
void init() {
    catalan[0] = catalan[1] = 1;
    for (int i=2; i<=n; i++) {
        catalan[i] = 0;
        for (int j=0; j < i; j++) {
            catalan[i] += (catalan[j] * catalan[i-j-1]) % MOD;
            if (catalan[i] >= MOD) {
                catalan[i] -= MOD;
            }
        }
    }
}
```

Analytical formula

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

(here $\binom{n}{k}$ denotes the usual binomial coefficient, i.e. number of ways to select k objects from set of n objects).

The above formula can be easily concluded from the problem of the monotonic paths in square grid. The total number of monotonic paths in the lattice size of $n \times n$ is given by $\binom{2n}{n}$.

Now we count the number of monotonic paths which cross the main diagonal. Consider such paths crossing the main diagonal and find the first edge in it which is above the diagonal. Reflect the path about the diagonal all the way, going after

this edge. The result is always a monotonic path in the grid $(n-1) \times (n+1)$. On the other hand, any monotonic path in the lattice $(n-1) \times (n+1)$ must intersect the diagonal. Hence, we enumerated all monotonic paths crossing the main diagonal in the lattice $n \times n$.

The number of monotonic paths in the lattice $(n-1) \times (n+1)$ are $\binom{2n}{n-1}$. Let us call such paths as “bad” paths. As a result, to obtain the number of monotonic paths which do not cross the main diagonal, we subtract the above “bad” paths, obtaining the formula:

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1} \binom{2n}{n}, n \geq 0$$

1.3.2 Reference

- [Catalan Number by Tom Davis](#)

1.3.3 Practice Problems

- [Codechef - PANSTACK](#)
- [Spoj - Skyline](#)
- [UVA - Safe Salutations](#)
- [Codeforces - How many trees?](#)
- [SPOJ - FUNPROB](#)
- [LOJ - 1170 - Counting Perfect BST](#)
- [UVA - 12887 - The Soldier's Dilemma](#)

Chapter 2

Techniques

2.1 The Inclusion-Exclusion Principle

The inclusion-exclusion principle is an important combinatorial way to compute the size of a set or the probability of complex events. It relates the sizes of individual sets with their union.

2.1.1 Statement

The verbal formula

The inclusion-exclusion principle can be expressed as follows:

To compute the size of a union of multiple sets, it is necessary to sum the sizes of these sets **separately**, and then subtract the sizes of all **pairwise** intersections of the sets, then add back the size of the intersections of **triples** of the sets, subtract the size of **quadruples** of the sets, and so on, up to the intersection of **all** sets.

The formulation in terms of sets

The above definition can be expressed mathematically as follows:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{i=1}^n |A_i| - \sum_{1 \leq i < j \leq n} |A_i \cap A_j| + \sum_{1 \leq i < j < k \leq n} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n-1} |A_1 \cap \cdots \cap A_n|$$

And in a more compact way:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{\emptyset \neq J \subseteq \{1, 2, \dots, n\}} (-1)^{|J|-1} \left| \bigcap_{j \in J} A_j \right|$$

The formulation using Venn diagrams

Let the diagram show three sets A , B and C :

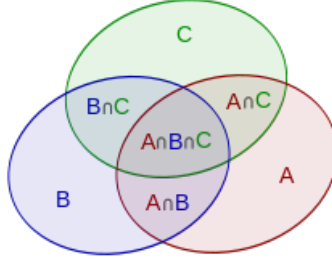


Figure 2.1: Venn diagram

Then the area of their union $A \cup B \cup C$ is equal to the sum of the areas A , B and C less double-covered areas $A \cap B$, $A \cap C$, $B \cap C$, but with the addition of the area covered by three sets $A \cap B \cap C$:

$$S(A \cup B \cup C) = S(A) + S(B) + S(C) - S(A \cap B) - S(A \cap C) - S(B \cap C) + S(A \cap B \cap C)$$

It can also be generalized for an association of n sets.

The formulation in terms of probability theory

If A_i ($i = 1, 2, \dots, n$) are events and $\mathcal{P}(A_i)$ the probability of an event from A_i to occur, then the probability of their union (i.e. the probability that at least one of the events occur) is equal to:

$$\begin{aligned} \mathcal{P}\left(\bigcup_{i=1}^n A_i\right) = & \sum_{i=1}^n \mathcal{P}(A_i) - \sum_{1 \leq i < j \leq n} \mathcal{P}(A_i \cap A_j) + \\ & + \sum_{1 \leq i < j < k \leq n} \mathcal{P}(A_i \cap A_j \cap A_k) - \dots + (-1)^{n-1} \mathcal{P}(A_1 \cap \dots \cap A_n) \end{aligned}$$

And in a more compact way:

$$\mathcal{P}\left(\bigcup_{i=1}^n A_i\right) = \sum_{\emptyset \neq J \subseteq \{1, 2, \dots, n\}} (-1)^{|J|-1} \mathcal{P}\left(\bigcap_{j \in J} A_j\right)$$

2.1.2 Proof

For the proof it is convenient to use the mathematical formulation in terms of set theory:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{\emptyset \neq J \subseteq \{1, 2, \dots, n\}} (-1)^{|J|-1} \left| \bigcap_{j \in J} A_j \right|$$

We want to prove that any element contained in at least one of the sets A_i will occur in the formula only once (note that elements which are not present in any of the sets A_i will never be considered on the right part of the formula).

Consider an element x occurring in $k \geq 1$ sets A_i . We will show it is counted only once in the formula. Note that:

- in terms which $|J| = 1$, the item x will be counted $+k$ times;
- in terms which $|J| = 2$, the item x will be counted $-\binom{k}{2}$ times - because it will be counted in those terms that include two of the k sets containing x ;
- in terms which $|J| = 3$, the item x will be counted $+\binom{k}{3}$ times;
- ...
- in terms which $|J| = k$, the item x will be counted $(-1)^{k-1} \cdot \binom{k}{k}$ times;
- in terms which $|J| > k$, the item x will be counted **zero** times;

This leads us to the following sum of [binomial coefficients](#):

$$T = \binom{k}{1} - \binom{k}{2} + \binom{k}{3} - \cdots + (-1)^{i-1} \cdot \binom{k}{i} + \cdots + (-1)^{k-1} \cdot \binom{k}{k}$$

This expression is very similar to the binomial expansion of $(1-x)^k$:

$$(1-x)^k = \binom{k}{0} - \binom{k}{1} \cdot x + \binom{k}{2} \cdot x^2 - \binom{k}{3} \cdot x^3 + \cdots + (-1)^k \cdot \binom{k}{k} \cdot x^k$$

When $x = 1$, $(1-x)^k$ looks a lot like T . However, the expression has an additional $\binom{k}{0} = 1$, and it is multiplied by -1 . That leads us to $(1-1)^k = 1-T$. Therefore $T = 1 - (1-1)^k = 1$, what was required to prove. The element is counted only once.

2.1.3 Generalization for calculating number of elements in exactly r sets

Inclusion-exclusion principle can be rewritten to calculate number of elements which are present in zero sets:

$$\left| \bigcap_{i=1}^n \overline{A_i} \right| = \sum_{m=0}^n (-1)^m \sum_{|X|=m} \left| \bigcap_{i \in X} A_i \right|$$

Consider its generalization to calculate number of elements which are present in exactly r sets:

$$\left| \bigcup_{|B|=r} \left[\bigcap_{i \in B} A_i \cap \bigcap_{j \notin B} \overline{A_j} \right] \right| = \sum_{m=r}^n (-1)^{m-r} \binom{m}{r} \sum_{|X|=m} \left| \bigcap_{i \in X} A_i \right|$$

To prove this formula, consider some particular B . Due to basic inclusion-exclusion principle we can say about it that:

$$\left| \bigcap_{i \in B} A_i \cap \bigcap_{j \notin B} \overline{A_j} \right| = \sum_{m=r}^n (-1)^{m-r} \sum_{|X|=m, B \subset X} \left| \bigcap_{i \in X} A_i \right|$$

The sets on the left side do not intersect for different B , thus we can sum them up directly. Also one should note that any set X will always have coefficient $(-1)^{m-r}$ if it occurs and it will occur for exactly $\binom{m}{r}$ sets B .

2.1.4 Usage when solving problems

The inclusion-exclusion principle is hard to understand without studying its applications.

First, we will look at three simplest tasks “at paper”, illustrating applications of the principle, and then consider more practical problems which are difficult to solve without inclusion-exclusion principle.

Tasks asking to “find the **number** of ways” are worth of note, as they sometimes lead to polynomial solutions, not necessarily exponential.

A simple task on permutations

Task: count how many permutations of numbers from 0 to 9 exist such that the first element is greater than 1 and the last one is less than 8.

Let’s count the number of “bad” permutations, that is, permutations in which the first element is ≤ 1 and/or the last is ≥ 8 .

We will denote by X the set of permutations in which the first element is ≤ 1 and Y the set of permutations in which the last element is ≥ 8 . Then the number of “bad” permutations, as on the inclusion-exclusion formula, will be:

$$|X \cup Y| = |X| + |Y| - |X \cap Y|$$

After a simple combinatorial calculation, we will get to:

$$2 \cdot 9! + 2 \cdot 9! - 2 \cdot 2 \cdot 8!$$

The only thing left is to subtract this number from the total of $10!$ to get the number of “good” permutations.

A simple task on (0, 1, 2) sequences

Task: count how many sequences of length n exist consisting only of numbers 0, 1, 2 such that each number occurs **at least once**.

Again let us turn to the inverse problem, i.e. we calculate the number of sequences which do **not** contain **at least one** of the numbers.

Let’s denote by $A_i (i = 0, 1, 2)$ the set of sequences in which **at least** i of the numbers do **not** occur. The formula of inclusion-exclusion on the number of “bad” sequences will be:

$$|A_0 \cup A_1 \cup A_2| = |A_0| + |A_1| + |A_2| - |A_0 \cap A_1| - |A_0 \cap A_2| - |A_1 \cap A_2| + |A_0 \cap A_1 \cap A_2|$$

- The size of each A_i is 2^n , as each sequence can only contain two of the digits.

- The size of each pairwise intersection $A_i \cap A_j$ is equal to 1, as there will be only one digit to build the sequence.
- The size of the intersection of all three sets is equal to 0, as there will be no digits to build the sequence.

As we solved the inverse problem, we subtract it from the total of 3^n sequences:

$$3^n - (3 \cdot 2^n - 3 \cdot 1 + 0)$$

The number of integer solutions to the equation

Consider the following equation:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 20$$

where $0 \leq x_i \leq 8 (i = 1, 2, \dots, 6)$.

Task: count the number of solutions to the equation.

Forget the restriction on x_i for a moment and just count the number of nonnegative solutions to this equation. This is easily done using [binomial coefficients](#): we want to break a sequence of 20 units into 6 groups, which is the same as distributing 5 “walls” over 25 slots:

$$N_0 = \binom{25}{5}$$

We will now calculate the number of “bad” solutions with the inclusion-exclusion principle. The “bad” solutions will be those in which one or more x_i are greater than 9.

Denote by $A_k (k = 1, 2, \dots, 6)$ the set of solutions where $x_k \geq 9$, and all other $x_i \geq 0 (i \neq k)$ (they may be ≥ 9 or not). To calculate the size of A_k , note that we have essentially the same combinatorial problem that was solved in the two paragraphs above, but now 9 of the units are excluded from the slots and definitely belong to the first group. Thus:

$$|A_k| = \binom{16}{5}$$

Similarly, the size of the intersection between sets A_k and A_p is equal to:

$$|A_k \cap A_p| = \binom{7}{5}$$

The size of each intersection of three sets is zero, since 20 units will not be enough for three or more variables greater than or equal to 9.

Combining all this into the formula of inclusions-exceptions and given that we solved the inverse problem, we finally get the answer:

$$\binom{25}{5} - \left(\binom{6}{1} \cdot \binom{16}{5} - \binom{6}{2} \cdot \binom{7}{5} \right)$$

The number of relatively primes in a given interval

Task: given two numbers n and r , count the number of integers in the interval $[1; r]$ that are relatively prime to n (their greatest common divisor is 1).

Let's solve the inverse problem - compute the number of not mutually primes with n .

We will denote the prime factors of n as $p_i (i = 1 \dots k)$.

How many numbers in the interval $[1; r]$ are divisible by p_i ? The answer to this question is:

$$\left\lfloor \frac{r}{p_i} \right\rfloor$$

However, if we simply sum these numbers, some numbers will be summarized several times (those that share multiple p_i as their factors). Therefore, it is necessary to use the inclusion-exclusion principle.

We will iterate over all 2^k subsets of p_i s, calculate their product and add or subtract the number of multiples of their product.

Here is a C++ implementation:

```
int solve (int n, int r) {
    vector<int> p;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            p.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        p.push_back (n);

    int sum = 0;
    for (int msk=1; msk<(1<<p.size()); ++msk) {
        int mult = 1,
            bits = 0;
        for (int i=0; i<(int)p.size(); ++i)
            if (msk & (1<<i)) {
                ++bits;
                mult *= p[i];
            }

        int cur = r / mult;
        if (bits % 2 == 1)
            sum += cur;
        else
            sum -= cur;
    }

    return r - sum;
}
```

Asymptotics of the solution is $O(\sqrt{n})$.

The number of integers in a given interval which are multiple of at least one of the given numbers

Given n numbers a_i and number r . You want to count the number of integers in the interval $[1; r]$ that are multiple of at least one of the a_i .

The solution algorithm is almost identical to the one for previous task — construct the formula of inclusion-exclusion on the numbers a_i , i.e. each term in this formula is the number of numbers divisible by a given subset of numbers a_i (in other words, divisible by their [least common multiple](#)).

So we will now iterate over all 2^n subsets of integers a_i with $O(n \log r)$ operations to find their least common multiple, adding or subtracting the number of multiples of it in the interval. Asymptotics is $O(2^n \cdot n \cdot \log r)$.

The number of strings that satisfy a given pattern

Consider n patterns of strings of the same length, consisting only of letters ($a...z$) or question marks. You're also given a number k . A string matches a pattern if it has the same length as the pattern, and at each position, either the corresponding characters are equal or the character in the pattern is a question mark. The task is to count the number of strings that match exactly k of the patterns (first problem) and at least k of the patterns (second problem).

Notice first that we can easily count the number of strings that satisfy at once all of the specified patterns. To do this, simply “cross” patterns: iterate through the positions (“slots”) and look at a position over all patterns. If all patterns have a question mark in this position, the character can be any letter from a to z . Otherwise, the character of this position is uniquely defined by the patterns that do not contain a question mark.

Learn now to solve the first version of the problem: when the string must satisfy exactly k of the patterns.

To solve it, iterate and fix a specific subset X from the set of patterns consisting of k patterns. Then we have to count the number of strings that satisfy this set of patterns, and only matches it, that is, they don't match any other pattern. We will use the inclusion-exclusion principle in a slightly different manner: we sum on all supersets Y (subsets from the original set of strings that contain X), and either add to the current answer or subtract it from the number of strings:

$$ans(X) = \sum_{Y \supseteq X} (-1)^{|Y|-k} \cdot f(Y)$$

Where $f(Y)$ is the number of strings that match Y (at least Y).

(If you have a hard time figuring out this, you can try drawing Venn Diagrams.)

If we sum up on all $ans(X)$, we will get the final answer:

$$ans = \sum_{X : |X|=k} ans(X)$$

However, asymptotics of this solution is $O(3^k \cdot k)$. To improve it, notice that different $ans(X)$ computations very often share Y sets.

We will reverse the formula of inclusion-exclusion and sum in terms of Y sets. Now it becomes clear that the same set Y would be taken into account in the computation of $ans(X)$ of $\binom{|Y|}{k}$ sets with the same sign $(-1)^{|Y|-k}$.

$$ans = \sum_{Y : |Y| \geq k} (-1)^{|Y|-k} \cdot \binom{|Y|}{k} \cdot f(Y)$$

Now our solution has asymptotics $O(2^k \cdot k)$.

We will now solve the second version of the problem: find the number of strings that match **at least** k of the patterns.

Of course, we can just use the solution to the first version of the problem and add the answers for sets with size greater than k . However, you may notice that in this problem, a set $|Y|$ is considered in the formula for all sets with size $\geq k$ which are contained in Y . That said, we can write the part of the expression that is being multiplied by $f(Y)$ as:

$$(-1)^{|Y|-k} \cdot \binom{|Y|}{k} + (-1)^{|Y|-k-1} \cdot \binom{|Y|}{k+1} + (-1)^{|Y|-k-2} \cdot \binom{|Y|}{k+2} + \dots + (-1)^{|Y|-|Y|} \cdot \binom{|Y|}{|Y|}$$

Looking at Graham's (Graham, Knuth, Patashnik. "Concrete mathematics" [1998]), we see a well-known formula for [binomial coefficients](#):

$$\sum_{k=0}^m (-1)^k \cdot \binom{n}{k} = (-1)^m \cdot \binom{n-1}{m}$$

Applying it here, we find that the entire sum of binomial coefficients is minimized:

$$(-1)^{|Y|-k} \cdot \binom{|Y|-1}{|Y|-k}$$

Thus, for this task, we also obtained a solution with the asymptotics $O(2^k \cdot k)$:

$$ans = \sum_{Y : |Y| \geq k} (-1)^{|Y|-k} \cdot \binom{|Y|-1}{|Y|-k} \cdot f(Y)$$

The number of ways of going from a cell to another

There is a field $n \times m$, and k of its cells are impassable walls. A robot is initially at the cell $(1, 1)$ (bottom left). The robot can only move right or up, and eventually it needs to get into the cell (n, m) , avoiding all obstacles. You need to count the number of ways he can do it.

Assume that the sizes n and m are very large (say, 10^9), and the number k is small (around 100).

For now, sort the obstacles by their coordinate x , and in case of equality — coordinate y .

Also just learn how to solve a problem without obstacles: i.e. learn how to count the number of ways to get from one cell to another. In one axis, we need to go through x cells, and on the other, y cells. From simple combinatorics, we get a formula using [binomial coefficients](#):

$$\binom{x+y}{x}$$

Now to count the number of ways to get from one cell to another, avoiding all obstacles, you can use inclusion-exclusion to solve the inverse problem: count the number of ways to walk through the board stepping at a subset of obstacles (and subtract it from the total number of ways).

When iterating over a subset of the obstacles that we'll step, to count the number of ways to do this simply multiply the number of all paths from starting cell to the first of the selected obstacles, a first obstacle to the second, and so on, and then add or subtract this number from the answer, in accordance with the standard formula of inclusion-exclusion.

However, this will again be non-polynomial in complexity $O(2^k \cdot k)$.

Here goes a polynomial solution:

We will use dynamic programming: let's compute the numbers $d[i][j]$ — the number of ways to get from the i -th point to j -th, without stepping on any other obstacle (except for i and j , of course). We will compute this number for all the obstacle cells, and also the starting and ending ones (all possible pairs of cells from these).

Let's forget for a second the obstacles and just count the number of paths from cell i to j . We need to consider some "bad" paths, the ones that pass through the obstacles, and subtract them from the total number of ways of going from i to j .

When considering an obstacle t between i and j ($i < t < j$), on which we can step, we see that the number of paths from i to j that pass through t which have t as the **first obstacle between i and j** . We can compute that as: $d[i][t]$ multiplied by the number of arbitrary paths from t to j . We can count the number of "bad" ways summing this for all t between i and j .

We can compute $d[i][j]$ in $O(k)$ for $O(k^2)$ pairs, so this solution has complexity $O(k^3)$.

The number of coprime quadruples

You're given n numbers: a_1, a_2, \dots, a_n . You are required to count the number of ways to choose four numbers so that their combined greatest common divisor is equal to one.

We will solve the inverse problem — compute the number of "bad" quadruples, i.e. quadruples in which all numbers are divisible by a number $d > 1$.

We will use the inclusion-exclusion principle while summing over all possible groups of four numbers divisible by a divisor d .

$$ans = \sum_{d \geq 2} (-1)^{\deg(d)-1} \cdot f(d)$$

where $\deg(d)$ is the number of primes in the factorization of the number d and $f(d)$ the number of quadruples divisible by d .

To calculate the function $f(d)$, you just have to count the number of multiples of d (as mentioned on a previous task) and use [binomial coefficients](#) to count the number of ways to choose four of them.

Thus, using the formula of inclusions-exclusions we sum the number of groups of four divisible by a prime number, then subtract the number of quadruples which are divisible by the product of two primes, add quadruples divisible by three primes, etc.

The number of harmonic triplets

You are given a number $n \leq 10^6$. You are required to count the number of triples $2 \leq a < b < c \leq n$ that satisfy one of the following conditions:

- 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 — i.e. count the number of non-harmonic triples.

Second, note that any non-harmonic triplet is made of a pair of coprimes and a third number that is not coprime with at least one from the pair.

Thus, the number of non-harmonic triples that contain i is equal the number of integers from 2 to n that are coprimes with i multiplied by the number of integers that are not coprime with i .

Either $\gcd(a, b) = 1 \wedge \gcd(a, c) > 1 \wedge \gcd(b, c) > 1$
or $\gcd(a, b) = 1 \wedge \gcd(a, c) = 1 \wedge \gcd(b, c) > 1$

In both of these cases, it will be counted twice. The first case will be counted when $i = a$ and when $i = b$. The second case will be counted when $i = b$ and when $i = c$. Therefore, to compute the number of non-harmonic triples, we sum this calculation through all i from 2 to n and divide it by 2.

Now all we have left to solve is to learn to count the number of coprimes to i in the interval $[2; n]$. Although this problem has already been mentioned, the above solution is not suitable here — it would require the factorization of each of the integers from 2 to n , and then iterating through all subsets of these primes.

A faster solution is possible with such modification of the sieve of Eratosthenes:

1. First, we find all numbers in the interval $[2; n]$ such that its simple factorization does not include a prime factor twice. We will also need to know, for these numbers, how many factors it includes.
 - To do this we will maintain an array $\deg[i]$ to store the number of primes in the factorization of i , and an array $good[i]$, to mark either if i contains each factor at most twice ($good[i] = 1$) or not ($good[i] = 0$).

When iterating from 2 to n , if we reach a number that has deg equal to 0, then it is a prime and its deg is 1.

- During the sieve of Eratosthenes, we will iterate i from 2 to n . When processing a prime number we go through all of its multiples and increase their $deg[]$. If one of these multiples is multiple of the square of i , then we can put $good$ as false.

2. Second, we need to calculate the answer for all i from 2 to n , i.e., the array $cnt[]$ — the number of integers not coprime with i .

- To do this, remember how the formula of inclusion-exclusion works — actually here we implement the same concept, but with inverted logic: we iterate over a component (a product of primes from the factorization) and add or subtract its term on the formula of inclusion-exclusion of each of its multiples.
- So, let's say we are processing a number i such that $good[i] = true$, i.e., it is involved in the formula of inclusion-exclusion. Iterate through all numbers that are multiples of i , and either add or subtract $\lfloor N/i \rfloor$ from their $cnt[]$ (the signal depends on $deg[i]$: if $deg[i]$ is odd, then we must add, otherwise subtract).

Here's a C++ 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;
}
```


The asymptotics of our solution is $O(n \log n)$, as for almost every number up to n we make n/i iterations on the nested loop.

The number of permutations without fixed points (derangements)

Prove that the number of permutations of length n without fixed points (i.e. no number i is in position i - also called a derangement) is equal to the following number:

$$n! - \binom{n}{1} \cdot (n-1)! + \binom{n}{2} \cdot (n-2)! - \binom{n}{3} \cdot (n-3)! + \cdots \pm \binom{n}{n} \cdot (n-n)!$$

and approximately equal to:

$$\frac{n!}{e}$$

(if you round this expression to the nearest whole number — you get exactly the number of permutations without fixed points)

Denote by A_k the set of permutations of length n with a fixed point at position k ($1 \leq k \leq n$) (i.e. element k is at position k).

We now use the formula of inclusion-exclusion to count the number of permutations with at least one fixed point. For this we need to learn to count sizes of an intersection of sets A_i , as follows:

$$\begin{aligned} |A_p| &= (n-1)! , \\ |A_p \cap A_q| &= (n-2)! , \\ |A_p \cap A_q \cap A_r| &= (n-3)! , \\ &\cdots , \end{aligned}$$

because if we know that the number of fixed points is equal x , then we know the position of x elements of the permutation, and all other $(n-x)$ elements can be placed anywhere.

Substituting this into the formula of inclusion-exclusion, and given that the number of ways to choose a subset of size x from the set of n elements is equal to $\binom{n}{x}$, we obtain a formula for the number of permutations with at least one fixed point:

$$\binom{n}{1} \cdot (n-1)! - \binom{n}{2} \cdot (n-2)! + \binom{n}{3} \cdot (n-3)! - \cdots \pm \binom{n}{n} \cdot (n-n)!$$

Then the number of permutations without fixed points is equal to:

$$n! - \binom{n}{1} \cdot (n-1)! + \binom{n}{2} \cdot (n-2)! - \binom{n}{3} \cdot (n-3)! + \cdots \pm \binom{n}{n} \cdot (n-n)!$$

Simplifying this expression, we obtain **exact and approximate expressions for the number of permutations without fixed points**:

$$n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots \pm \frac{1}{n!} \right) \approx \frac{n!}{e}$$

(because the sum in brackets are the first $n + 1$ terms of the expansion in Taylor series e^{-1})

It is worth noting that a similar problem can be solved this way: when you need the fixed points were not among the m first elements of permutations (and not among all, as we just solved). The formula obtained is as the given above accurate formula, but it will go up to the sum of k , instead of n .

2.1.5 Practice Problems

A list of tasks that can be solved using the principle of inclusions-exclusions:

- [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]
- [Test>>>thebest “HarmonicTriples”](#) (in Russian) [difficulty: medium]
- [SPOJ #4191 MSKYCODE “Sky Code”](#) [difficulty: medium]
- [SPOJ #4168 SQFREE “Square-free integers”](#) [difficulty: medium]
- [CodeChef “Count Relations”](#) [difficulty: medium]
- [SPOJ - Almost Prime Numbers Again](#)
- [SPOJ - Find number of Pair of Friends](#)
- [SPOJ - Balanced Cow Subsets](#)

2.2 Burnside's lemma / Pólya enumeration theorem

2.2.1 Burnside's lemma

Burnside's lemma was formulated and proven by **Burnside** in 1897, but historically it was already discovered in 1887 by **Frobenius**, and even earlier in 1845 by **Cauchy**. Therefore it is also sometimes named the **Cauchy-Frobenius lemma**.

Burnside's lemma allows us to count the number of equivalence classes in sets, based on internal symmetry.

Objects and representations

We have to clearly distinguish between the number of objects and the number of representations.

Different representations can correspond to the same objects, but of course any representation corresponds to exactly one object. Consequently the set of all representations is divided into equivalence classes. Our task is to compute the number of objects, or equivalently, the number of equivalence classes. The following example will make the difference between object and representation clearer.

Example: coloring of binary trees

Suppose we have the following problem. We have to count the number of ways to color a rooted binary tree with n vertices with two colors, where at each vertex we do not distinguish between the left and the right children.

Here the set of objects is the set of different colorings of the tree.

We now define the set of representations. A representation of a coloring is a function $f(v)$, which assigns each vertex a color (here we use the colors 0 and 1). The set of representations is the set containing all possible functions of this kind, and its size is obviously equal to 2^n .

At the same time we introduce a partition of this set into equivalence classes.

For example, suppose $n = 3$, and the tree consists of the root 1 and its two children 2 and 3. Then the following functions f_1 and f_2 are considered equivalent.

$$\begin{array}{ll} f_1(1) = 0 & f_2(1) = 0 \\ f_1(2) = 1 & f_2(2) = 0 \\ f_1(3) = 0 & f_2(3) = 1 \end{array}$$

Invariant permutations

Why do these two functions f_1 and f_2 belong to the same equivalence class? Intuitively this is understandable - we can rearrange the children of vertex 1, the vertices 2 and 3, and after such a transformation of the function f_1 it will coincide with f_2 .

But formally this means that there exists an **invariant permutation** π (i.e. a permutation which does not change the object itself, but only its representation), such that:

$$f_2\pi \equiv f_1$$

So starting from the definition of objects, we can find all the invariant permutations, i.e. all permutations which do not change the object when applying the permutation to the representation. Then we can check whether two functions f_1 and f_2 are equivalent (i.e. if they correspond to the same object) by checking the condition $f_2\pi \equiv f_1$ for each invariant permutation (or equivalently $f_1\pi \equiv f_2$). If at least one permutation is found for which the condition is satisfied, then f_1 and f_2 are equivalent, otherwise they are not equivalent.

Finding all such invariant permutations with respect to the object definition is a key step for the application of both Burnside's lemma and the Pólya enumeration theorem. It is clear that these invariant permutations depend on the specific problem, and their finding is a purely heuristic process based on intuitive considerations. However in most cases it is sufficient to manually find several "basic" permutations, with which all other permutations can be generated (and this part of the work can be shifted to a computer).

It is not difficult to understand that invariant permutations form a **group**, since the product (composition) of invariant permutations is again an invariant permutation. We denote the **group of invariant permutations** by G .

The statement of the lemma

For the formulation of the lemma we need one more definition from algebra. A **fixed point** f for a permutation π is an element that is invariant under this permutation: $f \equiv f\pi$. For example in our example the fixed points are those functions f , which correspond to colorings that do not change when the permutation π is applied to them (i.e. they do not change in the formal sense of the equality of functions). We denote by $I(\pi)$ the **number of fixed points** for the permutation π .

Then **Burnside's lemma** goes as follows: the number of equivalence classes is equal to the sum of the numbers of fixed points with respect to all permutations from the group G , divided by the size of this group:

$$|\text{Classes}| = \frac{1}{|G|} \sum_{\pi \in G} I(\pi)$$

Although Burnside's lemma itself is not so convenient to use in practice (it is unclear how to quickly look for the value $I(\pi)$), it most clearly reveals the mathematical essence on which the idea of calculating equivalence classes is based.

Proof of Burnside's lemma

The proof of Burnside's lemma described here is not important for the practical applications, so it can be skipped on the first reading.

The proof here is the simplest known, and does not use group theory. The proof was published by Kenneth P. Bogart in 1991.

We need to prove the following statement:

$$|\text{Classes}| \cdot |G| = \sum_{\pi \in G} I(\pi)$$

The value on the right side is nothing more than the number of “invariant pairs” (f, π) , i.e. pairs such that $f\pi \equiv f$. It is obvious that we can change the order of summation. We let the sum iterate over all elements f and sum over the values $J(f)$ - the number of permutations for which f is a fixed point.

$$|\text{Classes}| \cdot |G| = \sum_f J(f)$$

To prove this formula we will compose a table with columns labeled with all functions f_i and rows labeled with all permutations π_j . And we fill the cells with $f_i\pi_j$. If we look at the columns in this table as sets, then some of them will coincide, and this means that the corresponding functions f for these columns are also equivalent. Thus the numbers of different (as sets) columns is equal to the number of classes. Incidentally, from the standpoint of group theory, the column labeled with f_i is the orbit of this element. For equivalent elements the orbits coincide, and the number of orbits gives exactly the number of classes.

Thus the columns of the table decompose into equivalence classes. Let us fix a class, and look at the columns in it. First, note that these columns can only contain elements f_i of the equivalence class (otherwise some permutation π_j moved one of the functions into a different equivalence class, which is impossible since we only look at invariant permutations). Secondly each element f_i will occur the same number of times in each column (this also follows from the fact that the columns correspond to equivalent elements). From this we can conclude, that all the columns within the same equivalence class coincide with each other as multisets.

Now fix an arbitrary element f . On the one hand, it occurs in its column exactly $J(f)$ times (by definition). On the other hand, all columns within the same equivalence class are the same as multisets. Therefore within each column of a given equivalence class any element g occurs exactly $J(g)$ times.

Thus if we arbitrarily take one column from each equivalence class, and sum the number of elements in them, we obtain on one hand $|\text{Classes}| \cdot |G|$ (simply by multiplying the number of columns by the number of rows), and on the other hand the sum of the quantities $J(f)$ for all f (this follows from all the previous arguments):

$$|\text{Classes}| \cdot |G| = \sum_f J(f)$$

2.2.2 Pólya enumeration theorem

The Pólya enumeration theorem is a generalization of Burnside’s lemma, and it also provides a more convenient tool for finding the number of equivalence

classes. It should be noted that this theorem was already discovered before Pólya by Redfield in 1927, but his publication went unnoticed by mathematicians. Pólya independently came to the same results in 1937, and his publication was more successful.

Here we discuss only a special case of the Pólya enumeration theorem, which will turn out very useful in practice. The general formula of the theorem will not be discussed.

We denote by $C(\pi)$ the number of cycles in the permutation π . Then the following formula (a **special case of the Pólya enumeration theorem**) holds:

$$|\text{Classes}| = \frac{1}{|G|} \sum_{\pi \in G} k^{C(\pi)}$$

k is the number of values that each representation element can take, in the case of the coloring of a binary tree this would be $k = 2$.

Evidence

This formula is a direct consequence of Burnside's lemma. To get it, we just need to find an explicit expression for $I(\pi)$, which appears in the lemma. Recall, that $I(\pi)$ is the number of fixed points in the permutation π .

Thus we consider a permutation π and some element f . During the application of π , the elements in f move via the cycles in the permutation. Since the result should obtain $f \equiv f\pi$, the elements touched by one cycle must all be equal. At the same time different cycles are independent. Thus for each permutation cycle π we can choose one value (among k possible) and thus we get the number of fixed points:

$$I(\pi) = k^{C(\pi)}$$

2.2.3 Application: Coloring necklaces

The problem "Necklace" is one of the classical combinatorial problems. The task is to count the number of different necklaces from n beads, each of which can be painted in one of the k colors. When comparing two necklaces, they can be rotated, but not reversed (i.e. a cyclic shift is permitted).

In this problem we can immediately find the group of invariant permutations:

$$\begin{aligned}\pi_0 &= 123 \dots n \\ \pi_1 &= 23 \dots n1 \\ \pi_2 &= 3 \dots n12 \\ &\dots \\ \pi_{n-1} &= n123 \dots\end{aligned}$$

Let us find an explicit formula for calculating $C(\pi_i)$. First we note, that the permutation π_i has at the j -th position the value $i + j$ (taken modulo n). If we check the cycle structure for π_i . We see that 1 goes to $1 + i$, $1 + i$ goes to $1 + 2i$, which goes to $1 + 3i$, etc., until we come to a number of the form $1 + kn$.

Similar statements can be made for the remaining elements. Hence we see that all cycles have the same length, namely $\frac{\text{lcm}(i,n)}{i} = \frac{n}{\text{gcd}(i,n)}$. Thus the number of cycles in π_i will be equal to $\text{gcd}(i,n)$.

Substituting these values into the Pólya enumeration theorem, we obtain the solution:

$$\frac{1}{n} \sum_{i=1}^n k^{\text{gcd}(i,n)}$$

You can leave this formula in this form, or you can simplify it even more. Let transfer the sum so that it iterates over all divisors of n . In the original sum there will be many equivalent terms: if i is not a divisor of n , then such a divisor can be found after computing $\text{gcd}(i,n)$. Therefore for each divisor $d \mid n$ its term $k^{\text{gcd}(d,n)} = k^d$ will appear in the sum multiple times, i.e. the answer to the problem can be rewritten as

$$\frac{1}{n} \sum_{d \mid n} C_d k^d,$$

where C_d is the number of such numbers i with $\text{gcd}(i,n) = d$. We can find an explicit expression for this value. Any such number i has the form $i = dj$ with $\text{gcd}(j, n/d) = 1$ (otherwise $\text{gcd}(i,n) > d$). So we can count the number of j with this behavior. [Euler's phi function](#) gives us the result $C_d = \phi(n/d)$, and therefore we get the answer:

$$\frac{1}{n} \sum_{d \mid n} \phi\left(\frac{n}{d}\right) k^d$$

2.2.4 Application: Coloring a torus

Quite often we cannot obtain an explicit formula for the number of equivalence classes. In many problems the number of permutations in a group can be too large for manual calculations and it is not possible to compute analytically the number of cycles in them.

In that case we should manually find several “basic” permutations, so that they can generate the entire group G . Next we can write a program that will generate all permutations of the group G , count the number of cycles in them, and compute the answer with the formula.

Consider the example of the problem for coloring a torus. There is a checkered sheet of paper $n \times m$ ($n < m$), some of the cells are black. Then a cylinder is obtained from this sheet by gluing together the two sides with lengths m . Then a torus is obtained from the cylinder by gluing together the two circles (top and bottom) without twisting. The task is to compute the number of different colored tori, assuming that we cannot see the glued lines, and the torus can be turned and turned.

We again start with a piece of $n \times m$ paper. It is easy to see that the following types of transformations preserve the equivalence class: a cyclic shift of the rows, a cyclic shift of the columns, and a rotation of the sheet by 180 degrees. It is also easy to see, that these transformations can generate the entire group of invariant

transformations. If we somehow number the cells of the paper, then we can write three permutations p_1, p_2, p_3 corresponding to these types of transformation.

Next it only remains to generate all permutations obtained as a product. It is obvious that all such permutations have 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 implementations to this problem.

```
using Permutation = vector<int>;

void operator*=(Permutation& p, Permutation const& q) {
    Permutation copy = p;
    for (int i = 0; i < p.size(); i++)
        p[i] = copy[q[i]];
}

int count_cycles(Permutation p) {
    int cnt = 0;
    for (int i = 0; i < p.size(); i++) {
        if (p[i] != -1) {
            cnt++;
            for (int j = i; p[j] != -1;) {
                int next = p[j];
                p[j] = -1;
                j = next;
            }
        }
    }
    return cnt;
}

int solve(int n, int m) {
    Permutation 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);
    }

    set<Permutation> s;
    for (int i1 = 0; i1 < n; i1++) {
        for (int i2 = 0; i2 < m; i2++) {
            for (int i3 = 0; i3 < 2; i3++) {
                s.insert(p);
                p *= p3;
            }
            p *= p2;
        }
        p *= p1;
    }

    int sum = 0;
```



```
    for (Permutation const& p : s) {  
        sum += 1 << count_cycles(p);  
    }  
    return sum / s.size();  
}
```

2.3 Stars and bars

Stars and bars is a mathematical technique for solving certain combinatorial problems. It occurs whenever you want to count the number of ways to group identical objects.

2.3.1 Theorem

The number of ways to put n identical objects into k labeled boxes is

$$\binom{n+k-1}{n}.$$

The proof involves turning the objects into stars and separating the boxes using bars (therefore the name). E.g. we can represent with $\star|\star\star||\star\star$ the following situation: in the first box is one object, in the second box are two objects, the third one is empty and in the last box are two objects. This is one way of dividing 5 objects into 4 boxes.

It should be pretty obvious, that every partition can be represented using n stars and $k-1$ bars and every stars and bars permutation using n stars and $k-1$ bars represents one partition. Therefore the number of ways to divide n identical objects into k labeled boxes is the same number as there are permutations of n stars and $k-1$ bars. The [Binomial Coefficient](#) gives us the desired formula.

2.3.2 Number of non-negative integer sums

This problem is a direct application of the theorem.

You want to count the number of solution of the equation

$$x_1 + x_2 + \cdots + x_k = n$$

with $x_i \geq 0$.

Again we can represent a solution using stars and bars. E.g. the solution $1 + 3 + 0 = 4$ for $n = 4$, $k = 3$ can be represented using $\star|\star\star\star|$.

It is easy to see, that this is exactly the stars and bars theorem. Therefore the solution is $\binom{n+k-1}{n}$.

2.3.3 Number of lower-bound integer sums

This can easily be extended to integer sums with different lower bounds. I.e. we want to count the number of solutions for the equation

$$x_1 + x_2 + \cdots + x_k = n$$

with $x_i \geq a_i$.

After substituting $x'_i := x_i - a_i$ we receive the modified equation

$$\begin{aligned} (x'_1 + a_1) + (x'_2 + a_2) + \cdots + (x'_k + a_k) &= n \\ \Leftrightarrow x'_1 + x'_2 + \cdots + x'_k &= n - a_1 - a_2 - \cdots - a_k \end{aligned}$$

with $x'_i \geq 0$. So we have reduced the problem to the simpler case with $x'_i \geq 0$ and again can apply the stars and bars theorem.

2.4 Generating all K -combinations

In this article we will discuss the problem of generating all K -combinations. Given the natural numbers N and K , and considering a set of numbers from 1 to N . The task is to derive all **subsets of size K** .

2.4.1 Generate next lexicographical K -combination

First we will generate them in lexicographical order. The algorithm for this is simple. The first combination will be $1, 2, \dots, K$. Now let's see how to find the combination that immediately follows this, lexicographically. To do so, we consider our current combination, and find the rightmost element that has not yet reached its highest possible value. Once finding this element, we increment it by 1, and assign the lowest valid value to all subsequent elements.

```
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;
}
```

2.4.2 Generate all K -combinations such that adjacent combinations differ by one element

This time we want to generate all K -combinations in such an order, that adjacent combinations differ exactly by one element.

This can be solved using the [Gray Code](#): If we assign a bitmask to each subset, then by generating and iterating over these bitmasks with Gray codes, we can obtain our answer.

The task of generating K -combinations can also be solved using Gray Codes in a different way: Generate Gray Codes for the numbers from 0 to $2^N - 1$ and leave only those codes containing K 1s. The surprising fact is that in the resulting sequence of K set bits, any two neighboring masks (including the first and last mask - neighboring in a cyclic sense) - will differ exactly by two bits, which is our objective (remove a number, add a number).

Let us prove this:

For the proof, we recall the fact that the sequence $G(N)$ (representing the N th Gray Code) can be obtained as follows:

$$G(N) = 0G(N-1) \cup 1G(N-1)^R$$

That is, consider the Gray Code sequence for $N - 1$, and prefix 0 before every term. And consider the reversed Gray Code sequence for $N - 1$ and prefix a 1 before every mask, and concatenate these two sequences.

Now we may produce our proof.

First, we prove that the first and last masks differ exactly in two bits. To do this, it is sufficient to note that the first mask of the sequence $G(N)$, will be of the form $N - K$ 0s, followed by K 1s. As the first bit is set as 0, after which $(N - K - 1)$ 0s follow, after which K set bits follow and the last mask will be of the form 1, then $(N - K)$ 0s, then $K - 1$ 1s. Applying the principle of mathematical induction, and using the formula for $G(N)$, concludes the proof.

Now our task is to show that any two adjacent codes also differ exactly in two bits, we can do this by considering our recursive equation for the generation of Gray Codes. Let us assume the content of the two halves formed by $G(N - 1)$ is true. Now we need to prove that the new consecutive pair formed at the junction (by the concatenation of these two halves) is also valid, i.e. they differ by exactly two bits.

This can be done, as we know the last mask of the first half and the first mask of the second half. The last mask of the first half would be 1, then $(N - K - 1)$ 0s, then $K - 1$ 1s. And the first mask of the second half would be 0, then $(N - K - 2)$ 0s would follow, and then K 1s. Thus, comparing the two masks, we find exactly two bits that differ.

The following is a naive implementation working by generating all 2^n possible subsets, and finding subsets of size K .

```
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))
                    cout << j + 1;
            }
            cout << "\n";
        }
    }
}
```

It's worth mentioning that a more efficient implementation exists that only resorts to building valid combinations and thus works in $O\left(N \cdot \binom{N}{K}\right)$ however

it is recursive in nature and for smaller values of N it probably has a larger constant than the previous solution.

The implementation is derived from the formula:

$$G(N, K) = 0G(N - 1, K) \cup 1G(N - 1, K - 1)^R$$

This formula is obtained by modifying the general equation to determine the Gray code, and works by selecting the subsequence from appropriate elements.

Its implementation is as follows:

```
vector<int> ans;

void gen(int n, int k, int idx, bool rev) {
    if (k > n || k < 0)
        return;

    if (!n) {
        for (int i = 0; i < idx; ++i) {
            if (ans[i])
                cout << i + 1;
        }
        cout << "\n";
        return;
    }

    ans[idx] = rev;
    gen(n - 1, k - rev, idx + 1, false);
    ans[idx] = !rev;
    gen(n - 1, k - !rev, idx + 1, true);
}

void all_combinations(int n, int k) {
    ans.resize(n);
    gen(n, k, 0, false);
}
```

Chapter 3

Tasks

3.1 Placing Bishops on a Chessboard

Find the number of ways to place K bishops on an $N \times N$ chessboard so that no two bishops attack each other.

3.1.1 Algorithm

This problem can be solved using dynamic programming.

Let's enumerate the diagonals of the chessboard as follows: black diagonals have odd indices, white diagonals have even indices, and the diagonals are numbered in non-decreasing order of the number of squares in them. Here is an example for a 5×5 chessboard.

1	2	5	6	9
2	5	6	9	8
5	6	9	8	7
6	9	8	7	4
9	8	7	4	3

Let $D[i][j]$ denote the number of ways to place j bishops on diagonals with indices up to i which have the same color as diagonal i . Then $i = 1 \dots 2N-1$ and $j = 0 \dots K$.

We can calculate $D[i][j]$ using only values of $D[i-2]$ (we subtract 2 because we only consider diagonals of the same color as i). There are two ways to get $D[i][j]$. Either we place all j bishops on previous diagonals: then there are $D[i-2][j]$ ways to achieve this. Or we place one bishop on diagonal i and $j-1$ bishops on previous diagonals. The number of ways to do this equals the number of squares in diagonal i minus $j-1$, because each of $j-1$ bishops placed on previous diagonals will block one square on the current diagonal. The number of squares in diagonal i can be calculated as follows:

```
int squares (int i) {
    if (i & 1)
        return i / 4 * 2 + 1;
    else
```

```

        return (i - 1) / 4 * 2 + 2;
    }

```

The base case is simple: $D[i][0] = 1$, $D[1][1] = 1$.

Once we have calculated all values of $D[i][j]$, the answer can be obtained as follows: consider all possible numbers of bishops placed on black diagonals $i=0 \dots K$, with corresponding numbers of bishops on white diagonals $K-i$. The bishops placed on black and white diagonals never attack each other, so the placements can be done independently. The index of the last black diagonal is $2N-1$, the last white one is $2N-2$. For each i we add $D[2N-1][i] * D[2N-2][K-i]$ to the answer.

3.1.2 Implementation

```

int bishop_placements(int N, int K)
{
    if (K > 2 * N - 1)
        return 0;

    vector<vector<int>> D(N * 2, vector<int>(K + 1));
    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] * (squares(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];
    return ans;
}

```

3.2 Balanced bracket sequences

A **balanced bracket sequence** is a string consisting of only brackets, such that this sequence, when inserted certain numbers and mathematical operations, gives a valid mathematical expression. Formally you can define balanced bracket sequence with:

- e (the empty string) is a balanced bracket sequence.
- if s is a balanced bracket sequence, then so is (s) .
- if s and t are balanced bracket sequences, then so is st .

For instance $((()))$ is a balanced bracket sequence, but $(())$ is not.

Of course you can define other bracket sequences also with multiple bracket types in a similar fashion.

In this article we discuss some classic problems involving balanced bracket sequences (for simplicity we will only call them sequences): validation, number of sequences, finding the lexicographical next sequence, generating all sequences of a certain size, finding the index of sequence, and generating the k -th sequences. We will also discuss two variations for the problems, the simpler version when only one type of brackets is allowed, and the harder case when there are multiple types.

3.2.1 Balance validation

We want to check if a given string is balanced or not.

At first suppose there is only one type of bracket. For this case there exists a very simple algorithm. Let depth be the current number of open brackets. Initially depth = 0. We iterate over all character of the string, if the current bracket character is an opening bracket, then we increment depth, otherwise we decrement it. If at any time the variable depth gets negative, or at the end it is different from 0, then the string is not a balanced sequence. Otherwise it is.

If there are several bracket types involved, then the algorithm needs to be changes. Instead of a counter depth we create a stack, in which we will store all opening brackets that we meet. If the current bracket character is an opening one, we put it onto the stack. If it is a closing one, then we check if the stack is non-empty, and if the top element of the stack is of the same type as the current closing bracket. If both conditions are fulfilled, then we remove the opening bracket from the stack. If at any time one of the conditions is not fulfilled, or at the end the stack is not empty, then the string is not balanced. Otherwise it is.

3.2.2 Number of balanced sequences

Formula

The number of balanced bracket sequences with only one bracket type can be calculated using the [Catalan numbers](#). The number of balanced bracket sequences

of length $2n$ (n pairs of brackets) is:

$$\frac{1}{n+1} \binom{2n}{n}$$

If we allow k types of brackets, then each pair be of any of the k types (independently of the others), thus the number of balanced bracket sequences is:

$$\frac{1}{n+1} \binom{2n}{n} k^n$$

Dynamic programming

On the other hand these numbers can be computed using **dynamic programming**. Let $d[n]$ be the number of regular bracket sequences with n pairs of bracket. Note that in the first position there is always an opening bracket. And somewhere later is the corresponding closing bracket of the pair. It is clear that inside this pair there is a balanced bracket sequence, and similarly after this pair there is a balanced bracket sequence. So to compute $d[n]$, we will look at how many balanced sequences of i pairs of brackets are inside this first bracket pair, and how many balanced sequences with $n-1-i$ pairs are after this pair. Consequently the formula has the form:

$$d[n] = \sum_{i=0}^{n-1} d[i] \cdot d[n-1-i]$$

The initial value for this recurrence is $d[0] = 1$.

3.2.3 Finding the lexicographical next balanced sequence

Here we only consider the case with one valid bracket type.

Given a balanced sequence, we have to find the next (in lexicographical order) balanced sequence.

It should be obvious, that we have to find the rightmost opening bracket, which we can replace by a closing bracket without violation the condition, that there are more closing brackets than opening brackets up to this position. After replacing this position, we can fill the remaining part of the string with the lexicographically minimal one: i.e. first with as much opening brackets as possible, and then fill up the remaining positions with closing brackets. In other words we try to leave as long as possible prefix unchanged, and the suffix gets replaced by the lexicographically minimal one.

To find this position, we can iterate over the character from right to left, and maintain the balance depth of open and closing brackets. When we meet an opening brackets, we will decrement depth, and when we meet a closing bracket, we increase it. If we are at some point meet an opening bracket, and the balance after processing this symbol is positive, then we have found the rightmost position that we can change. We change the symbol, compute the number of opening and

closing brackets that we have to add to the right side, and arrange them in the lexicographically minimal way.

If we find do suitable position, then this sequence is already the maximal possible one, and there is no answer.

```
bool next_balanced_sequence(string & s) {
    int n = s.size();
    int depth = 0;
    for (int i = n - 1; 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;
            string next = s.substr(0, i) + ')' + string(open, '(') + string(close, ')');
            s.swap(next);
            return true;
        }
    }
    return false;
}
```

This function computes in $O(n)$ time the next balanced bracket sequence, and returns false if there is no next one.

3.2.4 Finding all balanced sequences

Sometimes it is required to find and output all balanced bracket sequences of a specific length n .

To generate then, we can start with the lexicographically smallest sequence $((\dots((\dots)))$, and then continue to find the next lexicographically sequences with the algorithm described in the previous section.

However, if the length of the sequence is not very long (e.g. n smaller than 12), then we can also generate all permutations conveniently with the C++ STL function `next_permutation`, and check each one for balanceness.

Also they can be generate using the ideas we used for counting all sequences with dynamic programming. We will discuss the ideas in the next two sections.

3.2.5 Sequence index

Given a balanced bracket sequence with n pairs of brackets. We have to find its index in the lexicographically ordered list of all balanced sequences with n bracket pairs.

Let's define an auxiliary array $d[i][j]$, where i is the length of the bracket sequence (semi-balanced, each closing bracket has a corresponding opening

bracket, but not every opening bracket has necessarily a corresponding closing one), and j is the current balance (difference between opening and closing brackets). $d[i][j]$ is the number of such sequences that fit the parameters. We will calculate these numbers with only one bracket type.

For the start value $i = 0$ the answer is obvious: $d[0][0] = 1$, and $d[0][j] = 0$ for $j > 0$. Now let $i > 0$, and we look at the last character in the sequence. If the last character was an opening bracket $($, then the state before was $(i - 1, j - 1)$, if it was a closing bracket $)$, then the previous state was $(i - 1, j + 1)$. Thus we obtain the recursion formula:

$$d[i][j] = d[i - 1][j - 1] + d[i - 1][j + 1]$$

$d[i][j] = 0$ holds obviously for negative j . Thus we can compute this array in $O(n^2)$.

Now let us generate the index for a given sequence.

First let there be only one type of brackets. We will use the counter *depth* which tells us how nested we currently are, and iterate over the characters of the sequence. If the current character $s[i]$ is equal to $($, then we increment *depth*. If the current character $s[i]$ is equal to $)$, then we must add $d[2n - i - 1][\text{depth} + 1]$ to the answer, taking all possible endings starting with a $($ into account (which are lexicographically smaller sequences), and then decrement *depth*.

Now let there be k different bracket types.

Thus, when we look at the current character $s[i]$ before recomputing *depth*, we have to go through all bracket types that are smaller than the current character, and try to put this bracket into the current position (obtaining a new balance $\text{ndepth} = \text{depth} \pm 1$), and add the number of ways to finish the sequence (length $2n - i - 1$, balance *ndepth*) to the answer:

$$d[2n - i - 1][\text{ndepth}] \cdot k^{\frac{2n - i - 1 - \text{ndepth}}{2}}$$

This formula can be derived as follows: First we “forget” that there are multiple bracket types, and just take the answer $d[2n - i - 1][\text{ndepth}]$. Now we consider how the answer will change if we have k types of brackets. We have $2n - i - 1$ undefined positions, of which *ndepth* are already predetermined because of the opening brackets. But all the other brackets $((2n - i - 1 - \text{ndepth})/2$ pairs) can be of any type, therefore we multiply the number by such a power of k .

3.2.6 Finding the k -th sequence

Let n be the number of bracket pairs in the sequence. We have to find the k -th balanced sequence in lexicographically sorted list of all balanced sequences for a given k .

As in the previous section we compute the auxiliary array $d[i][j]$, the number of semi-balanced bracket sequences of length i with balance j .

First, we start with only one bracket type.

We will iterate over the characters in the string we want to generate. As in the previous problem we store a counter *depth*, the current nesting depth. In

each position we have to decide if we use an opening of a closing bracket. To have to put an opening bracket character, it $d[2n - i - 1][\text{depth} + 1] \geq k$. We increment the counter depth, and move on to the next character. Otherwise we decrement k by $d[2n - i - 1][\text{depth} + 1]$, put a closing bracket and move on.

```
string kth_balanced(int n, int k) {
    vector<vector<int>>> d(2*n+1, vector<int>(n+1, 0));
    d[0][0] = 1;
    for (int i = 1; i <= 2*n; i++) {
        d[i][0] = d[i-1][1];
        for (int j = 1; j < n; j++)
            d[i][j] = d[i-1][j-1] + d[i-1][j+1];
        d[i][n] = d[i-1][n-1];
    }

    string ans;
    int depth = 0;
    for (int i = 0; i < 2*n; i++) {
        if (depth + 1 <= n && d[2*n-i-1][depth+1] >= k) {
            ans += '(';
            depth++;
        } else {
            ans += ')';
            if (depth + 1 <= n)
                k -= d[2*n-i-1][depth+1];
            depth--;
        }
    }
    return ans;
}
```

Now let there be k types of brackets. The solution will only differ slightly in that we have to multiply the value $d[2n - i - 1][\text{ndepth}]$ by $k^{(2n-i-1-\text{ndepth})/2}$ and take into account that there can be different bracket types for the next character.

Here is an implementation using two types of brackets: round and square:

```
string kth_balanced2(int n, int k) {
    vector<vector<int>>> d(2*n+1, vector<int>(n+1, 0));
    d[0][0] = 1;
    for (int i = 1; i <= 2*n; i++) {
        d[i][0] = d[i-1][1];
        for (int j = 1; j < n; j++)
            d[i][j] = d[i-1][j-1] + d[i-1][j+1];
        d[i][n] = d[i-1][n-1];
    }

    string ans;
    int depth = 0;
    stack<char> st;
    for (int i = 0; i < 2*n; i++) {
        // '('
        if (depth + 1 <= n) {
```

```

        int cnt = d[2*n-i-1][depth+1] << ((2*n-i-1-depth-1) / 2);
        if (cnt >= k) {
            ans += '(';
            st.push('(');
            depth++;
            continue;
        }
        k -= cnt;
    }

    // ')'
    if (depth && st.top() == '(') {
        int cnt = d[2*n-i-1][depth-1] << ((2*n-i-1-depth+1) / 2);
        if (cnt >= k) {
            ans += ')';
            st.pop();
            depth--;
            continue;
        }
        k -= cnt;
    }

    // '['
    if (depth + 1 <= n) {
        int cnt = d[2*n-i-1][depth+1] << ((2*n-i-1-depth-1) / 2);
        if (cnt >= k) {
            ans += '[';
            st.push('[');
            depth++;
            continue;
        }
        k -= cnt;
    }

    // ']'
    ans += ']';
    st.pop();
    depth--;
}
return ans;
}

```

3.3 Counting labeled graphs

3.3.1 Labeled graphs

Let the number of vertices in a graph be n . We have to compute the number G_n of labeled graphs with n vertices (labeled means that the vertices are marked with the numbers from 1 to n). The edges of the graphs are considered undirected, and loops and multiple edges are forbidden.

We consider the set of all possible edges of the graph. For each edge (i, j) we can assume that $i < j$ (because the graph is undirected, and there are no loops). Therefore the set of all edges has the cardinality $\binom{n}{2}$, i.e. $\frac{n(n-1)}{2}$.

Since any labeled graph is uniquely determined by its edges, the number of labeled graphs with n vertices is equal to:

$$G_n = 2^{\frac{n(n-1)}{2}}$$

3.3.2 Connected labeled graphs

Here, we additionally impose the restriction that the graph has to be connected.

Let's denote the required number of connected graphs with n vertices as C_n .

We will first discuss how many **disconnected** graphs exists. Then the number of connected graphs will be G_n minus the number of disconnected graphs. Even more, we will count the number of **disconnected, rooted graphs**. A rooted graph is a graph, where we emphasize one vertex by labeling it as root. Obviously we have n possibilities to root a graph with n labeled vertices, therefore we will need to divide the number of disconnected rooted graphs by n at the end to get the number of disconnected graphs.

The root vertex will appear in a connected component of size $1, \dots, n-1$. There are $k \binom{n}{k} C_k G_{n-k}$ graphs such that the root vertex is in a connected component with k vertices (there are $\binom{n}{k}$ ways to choose k vertices for the component, these are connected in one of C_k ways, the root vertex can be any of the k vertices, and the remainder $n-k$ vertices can be connected/disconnected in any way, which gives a factor of G_{n-k}). Therefore the number of disconnected graphs with n vertices is:

$$\frac{1}{n} \sum_{k=1}^{n-1} k \binom{n}{k} C_k G_{n-k}$$

And finally the number of connected graphs is:

$$C_n = G_n - \frac{1}{n} \sum_{k=1}^{n-1} k \binom{n}{k} C_k G_{n-k}$$

3.3.3 Labeled graphs with k connected components

Based on the formula from the previous section, we will learn how to count the number of labeled graphs with n vertices and k connected components.

This number can be computed using dynamic programming. We will compute $D[i][j]$ - the number of labeled graphs with i vertices and j components - for each $i \leq n$ and $j \leq k$.

Let's discuss how to compute the next element $D[n][k]$ if we already know the previous values. We use a common approach, we take the last vertex (index n). This vertex belongs to some component. If the size of this component be s , then there are $\binom{n-1}{s-1}$ ways to choose such a set of vertices, and C_s ways to connect them. After removing this component from the graph we have $n - s$ remaining vertices with $k - 1$ connected components. Therefore we obtain the following recurrence relation:

$$D[n][k] = \sum_{s=1}^n \binom{n-1}{s-1} C_s D[n-s][k-1]$$

Part VII

Numerical Methods

Chapter 1

Search

1.1 Ternary Search

We are given a function $f(x)$ which is unimodal on an interval $[l; r]$. By unimodal function, we mean one of two behaviors of the function:

1. The function strictly increases first, reaches a maximum (at one point or at a segment), and then strictly decreases.
2. The function strictly decreases first, reaches a minimum, and then strictly increases.

In this article we will assume the first scenario, and the second is be completely symmetrical to it.

The task is to find the maximum of function $f(x)$ on the interval $[l; r]$.

1.1.1 Algorithm

Consider any 2 points m_1 , and m_2 in this interval: $l < m_1 < m_2 < r$. We evaluate the function at m_1 and m_2 , i.e. find the values of $f(m_1)$ and $f(m_2)$. Now, we get one of three options:

- $f(m_1) < f(m_2)$

The desired maximum can not be located on the left side of m_1 , i.e. on the interval $[l; m_1]$, since either both points m_1 and m_2 or just m_1 belong to the area where the function increases. In either case, this means that we have to search for the maximum in the segment $[m_1, r]$.

- $f(m_1) > f(m_2)$

This situation is symmetrical to the previous one: the maximum can not be located on the right side of m_2 , i.e. on the interval $[m_2; r]$, and the search space is reduced to the segment $[l; m_2]$.

- $f(m_1) = f(m_2)$

We can see that either both of these points belong to the area where the value of the function is maximized, or m_1 is in the area of increasing values and m_2 is in the area of descending values (here we used the strictness of function increasing/decreasing). Thus, the search space is reduced to $[m_1; m_2]$. To simplify the code, this case can be combined with any of the previous cases.

Thus, based on the comparison of the values in the two inner points, we can replace the current interval $[l; r]$ with a new, shorter interval $[l'; r']$. Repeatedly applying the described procedure to the interval, we can get arbitrarily short interval. Eventually its length will be less than a certain pre-defined constant (accuracy), and the process can be stopped. This is a numerical method, so we can assume that after that the function reaches its maximum at all points of the last interval $[l; r]$. Without loss of generality, we can take $f(l)$ as the return value.

We didn't impose any restrictions on the choice of points m_1 and m_2 . This choice will define the convergence rate and the accuracy of the implementation. The most common way is to choose the points so that they divide the interval $[l; r]$ into three equal parts. Thus, we have

$$m_1 = l + \frac{(r - l)}{3}$$

$$m_2 = r - \frac{(r - l)}{3}$$

If m_1 and m_2 are chosen to be closer to each other, the convergence rate will increase slightly.

Run time analysis

$$T(n) = T(2n/3) + 1 = \Theta(\log n)$$

It can be visualized as follows: every time after evaluating the function at points m_1 and m_2 , we are essentially ignoring about one third of the interval, either the left or right one. Thus the size of the search space is $2n/3$ of the original one.

Applying [Master's Theorem](#), we get the desired complexity estimate.

The case of the integer arguments

If $f(x)$ takes integer parameter, the interval $[l; r]$ becomes discrete. Since we did not impose any restrictions on the choice of points m_1 and m_2 , the correctness of the algorithm is not affected. m_1 and m_2 can still be chosen to divide $[l; r]$ into 3 approximately equal parts.

The difference occurs in the stopping criterion of the algorithm. Ternary search will have to stop when $(r - l) < 3$, because in that case we can no longer select m_1 and m_2 to be different from each other as well as from l and r , and this can cause infinite iterating. Once $(r - l) < 3$, the remaining pool of candidate

points $(l, l+1, \dots, r)$ needs to be checked to find the point which produces the maximum value $f(x)$.

1.1.2 Implementation

```
double ternary_search(double l, double r) {
    double eps = 1e-9;           //set the error limit here
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);       //evaluates the function at m1
        double f2 = f(m2);       //evaluates the function at m2
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l);                 //return the maximum of f(x) in [l, r]
}
```

Here `eps` is in fact the absolute error (not taking into account errors due to the inaccurate calculation of the function).

Instead of the criterion `r - l > eps`, we can select a constant number of iterations as a stopping criterion. The number of iterations should be chosen to ensure the required accuracy. Typically, in most programming challenges the error limit is 10^{-6} and thus 200 - 300 iterations are sufficient. Also, the number of iterations doesn't depend on the values of l and r , so the number of iterations sets required relative error.

1.1.3 Practice Problems

- [Codechef - Race time](#)
- [Hackerearth - Rescuer](#)
- [Spoj - Building Construction](#)
- [Codeforces - Weakness and Poorness](#)
- [LOJ - Closest Distance](#)
- [GYM - Dome of Circus \(D\)](#)
- [UVA - Galactic Taxes](#)
- [GYM - Chasing the Cheetahs \(A\)](#)
- [UVA - 12197 - Trick or Treat](#)
- [SPOJ - Building Construction](#)
- [Codeforces - Devu and his Brother](#)

1.2 Newton's method for finding roots

This is an iterative method invented by Isaac Newton around 1664. However, sometimes this method is called the Raphson method, since Raphson invented the same algorithm a few years after Newton, but his article was published much earlier.

The task is as follows. Given the following equation:

$$f(x) = 0$$

We want to solve the equation, more precisely, to find one of its roots (it is assumed that the root exists). It is assumed that $f(x)$ is continuous and differentiable on an interval $[a; b]$.

1.2.1 Algorithm

The input parameters of the algorithm consist of not only the function $f(x)$ but also the initial approximation - some x_0 , with which the algorithm starts.

Suppose we have already calculated x_i , calculate x_{i+1} as follows. Draw the 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 x -coordinate of the point found, and we repeat the whole process from the beginning.

It is not difficult 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 “good” (smooth), and x_i is close enough to the root, then x_{i+1} will be even closer to the desired root.

The rate of convergence is quadratic, which, conditionally speaking, means that the number of exact digits in the approximate value x_i doubles with each iteration.

1.2.2 Application for calculating the square root

Let's use the calculation of square root as an example of Newton's method.

If we substitute $f(x) = \sqrt{x}$, then after simplifying the expression, we get:

$$x_{i+1} = \frac{x_i + \frac{n}{x_i}}{2}$$

The first typical variant of the problem is when a rational number n is given, and its root must be calculated with some accuracy `eps`:

```
double sqrt_newton(double n) {
    const double eps = 1E-15;
    double x = 1;
    for (;;) {
        double nx = (x + n / x) / 2;
        if (abs(x - nx) < eps)
```

```

        break;
    x = nx;
}
return x;
}

```

Another common variant of the problem is when we need to calculate the integer root (for the given n find the largest x such that $x^2 \leq n$). Here it is necessary to slightly change the termination condition of the algorithm, since it may happen that x will start to “jump” near the answer. Therefore, we add a condition that if the value x has decreased in the previous step, and it tries to increase at the current step, then the algorithm must be stopped.

```

int isqrt_newton(int 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;
    }
    return x;
}

```

Finally, we are given the third variant - for the case of bignum arithmetic. Since the number n can be large enough, it makes sense to pay attention to the initial approximation. Obviously, the closer it is to the root, the faster the result will be achieved. It is simple enough and effective to take the initial approximation as the number $2^{\text{bits}/2}$, where bits is the number of bits in the number n . Here is the Java code that demonstrates this variant:

```

public static BigInteger isqrtNewton(BigInteger n) {
    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;
    }
    return a;
}

```

For example, this code is executed in 60 milliseconds for $n = 10^{1000}$, and if we remove the improved selection of the initial approximation (just starting with 1), then it will be executed in about 120 milliseconds.

1.2.3 Practice Problems

- [UVa 10428 - The Roots](#)

Chapter 2

Integration

2.1 Integration by Simpson's formula

We are going to calculate the value of a definite integral

$$\int_a^b f(x) dx$$

The solution described here was published in one of the dissertations of **Thomas Simpson** in 1743.

2.1.1 Simpson's formula

Let n be some natural number. We divide the integration segment $[a; b]$ into $2n$ equal parts:

$$x_i = a + ih, \quad i = 0 \dots 2n,$$

$$h = \frac{b - a}{2n}.$$

Now we calculate the integral separately on each of the segments $[x_{2i-2}, x_{2i}]$, $i = 1 \dots n$, and then add all the values.

So, suppose we consider the next segment $[x_{2i-2}, x_{2i}]$, $i = 1 \dots n$. Replace the function $f(x)$ on it with a parabola $P(x)$ passing through 3 points $(x_{2i-2}, x_{2i-1}, x_{2i})$. Such a parabola always exists and is unique; it can be found analytically. For instance we could construct it using the Lagrange polynomial interpolation. The only remaining thing left to do is to integrate this polynomial. If you do this for a general function f , you receive a remarkable simple expression:

$$\int_{x_{2i-2}}^{x_{2i}} f(x) dx \approx \int_{x_{2i-2}}^{x_{2i}} P(x) dx = (f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})) \frac{h}{3}$$

Adding these values over all segments, we obtain the final **Simpson's formula**:

$$\int_a^b f(x) dx \approx (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \dots + 4f(x_{2N-1}) + f(x_{2N})) \frac{h}{3}$$

2.1.2 Error

The error in approximating an integral by Simpson's formula is

$$-\frac{1}{90} \left(\frac{b-a}{2} \right)^5 f^{(4)}(\xi)$$

where ξ is some number between a and b .

The error is asymptotically proportional to $(b-a)^5$. However, the above derivations suggest an error proportional to $(b-a)^4$. Simpson's rule gains an extra order because the points at which the integrand is evaluated are distributed symmetrically in the interval $[a, b]$.

2.1.3 Implementation

Here $f(x)$ is some user function.

```
const int N = 1000 * 1000; // number of steps (already multiplied by 2)

double simpson_integration(double a, double b){
    double h = (b - a) / N;
    double s = f(a) + f(b); // a = x_0 and b = x_{2n}
    for (int i = 1; i <= N - 1; ++i) { // Refer to final Simpson's formula
        double x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

2.1.4 Practice Problems

- [URI - Environment Protection](#)

Part VIII

Geometry

Chapter 1

Elementary operations

1.1 Basic Geometry

In this article we will consider basic operations on points in Euclidean space which maintains the foundation of the whole analytical geometry. We will consider for each point \mathbf{r} the vector $\vec{\mathbf{r}}$ directed from $\mathbf{0}$ to \mathbf{r} . Later we will not distinguish between \mathbf{r} and $\vec{\mathbf{r}}$ and use the term **point** as a synonym for **vector**.

1.1.1 Linear operations

Both 2D and 3D points maintain linear space, which means that for them sum of points and multiplication of point by some number are defined. Here are those basic implementations for 2D:

```
struct point2d {
    ftype x, y;
    point2d() {}
    point2d(ftype x, ftype y): x(x), y(y) {}
    point2d& operator+=(const point2d &t) {
        x += t.x;
        y += t.y;
        return *this;
    }
    point2d& operator-=(const point2d &t) {
        x -= t.x;
        y -= t.y;
        return *this;
    }
    point2d& operator*=(ftype t) {
        x *= t;
        y *= t;
        return *this;
    }
    point2d& operator/=(ftype t) {
        x /= t;
        y /= t;
        return *this;
    }
}
```

```

    }
    point2d operator+(const point2d &t) const {
        return point2d(*this) += t;
    }
    point2d operator-(const point2d &t) const {
        return point2d(*this) -= t;
    }
    point2d operator*(ftype t) const {
        return point2d(*this) *= t;
    }
    point2d operator/(ftype t) const {
        return point2d(*this) /= t;
    }
};
point2d operator*(ftype a, point2d b) {
    return b * a;
}

```

And 3D points:

```

struct point3d {
    ftype x, y, z;
    point3d() {}
    point3d(ftype x, ftype y, ftype z): x(x), y(y), z(z) {}
    point3d& operator+=(const point3d &t) {
        x += t.x;
        y += t.y;
        z += t.z;
        return *this;
    }
    point3d& operator-=(const point3d &t) {
        x -= t.x;
        y -= t.y;
        z -= t.z;
        return *this;
    }
    point3d& operator*=(ftype t) {
        x *= t;
        y *= t;
        z *= t;
        return *this;
    }
    point3d& operator/=(ftype t) {
        x /= t;
        y /= t;
        z /= t;
        return *this;
    }
    point3d operator+(const point3d &t) const {
        return point3d(*this) += t;
    }
    point3d operator-(const point3d &t) const {

```

```

        return point3d(*this) -= t;
    }
    point3d operator*(ftype t) const {
        return point3d(*this) *= t;
    }
    point3d operator/(ftype t) const {
        return point3d(*this) /= t;
    }
};
point3d operator*(ftype a, point3d b) {
    return b * a;
}

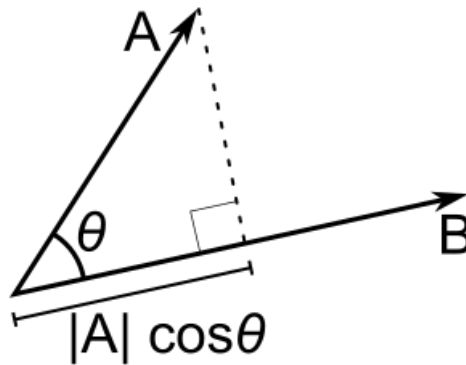
```

Here `ftype` is some type used for coordinates, usually `int`, `double` or `long`.

1.1.2 Dot product

Definition

The dot (or scalar) product $\mathbf{a} \cdot \mathbf{b}$ for vectors \mathbf{a} and \mathbf{b} can be defined in two identical ways. Geometrically it is product of the length of the first vector by the length of the projection of the second vector onto the first one. As you may see from the image below this projection is nothing but $|\mathbf{a}| \cos \theta$ where θ is the angle between \mathbf{a} and \mathbf{b} . Thus $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \cos \theta \cdot |\mathbf{b}|$.



The dot product holds some notable properties:

1. $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$
2. $(\alpha \cdot \mathbf{a}) \cdot \mathbf{b} = \alpha \cdot (\mathbf{a} \cdot \mathbf{b})$
3. $(\mathbf{a} + \mathbf{b}) \cdot \mathbf{c} = \mathbf{a} \cdot \mathbf{c} + \mathbf{b} \cdot \mathbf{c}$

I.e. it is a commutative function which is linear with respect to both arguments. Let's denote the unit vectors as

$$\mathbf{e}_x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \mathbf{e}_y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{e}_z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

With this notation we can write the vector $\mathbf{r} = (x; y; z)$ as $r = x \cdot \mathbf{e}_x + y \cdot \mathbf{e}_y + z \cdot \mathbf{e}_z$. And since for unit vectors

$$\mathbf{e}_x \cdot \mathbf{e}_x = \mathbf{e}_y \cdot \mathbf{e}_y = \mathbf{e}_z \cdot \mathbf{e}_z = 1, \mathbf{e}_x \cdot \mathbf{e}_y = \mathbf{e}_y \cdot \mathbf{e}_z = \mathbf{e}_z \cdot \mathbf{e}_x = 0$$

we can see that in terms of coordinates for $\mathbf{a} = (x_1; y_1; z_1)$ and $\mathbf{b} = (x_2; y_2; z_2)$ holds

$$\mathbf{a} \cdot \mathbf{b} = (x_1 \cdot \mathbf{e}_x + y_1 \cdot \mathbf{e}_y + z_1 \cdot \mathbf{e}_z) \cdot (x_2 \cdot \mathbf{e}_x + y_2 \cdot \mathbf{e}_y + z_2 \cdot \mathbf{e}_z) = x_1 x_2 + y_1 y_2 + z_1 z_2$$

That is also the algebraic definition of the dot product. From this we can write functions which calculate it.

```
ftype dot(point2d a, point2d b) {
    return a.x * b.x + a.y * b.y;
}
ftype dot(point3d a, point3d b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

When solving problems one should use algebraic definition to calculate dot products, but keep in mind geometric definition and properties to use it.

Properties

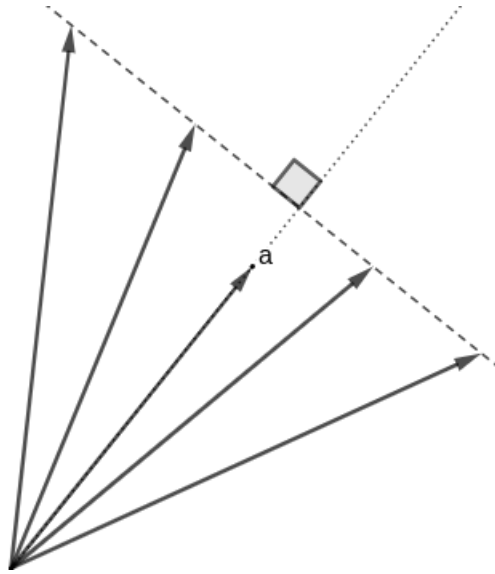
We can define many geometrical properties via the dot product. For example

1. Norm of \mathbf{a} (squared length): $|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a}$
2. Length of \mathbf{a} : $|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$
3. Projection of \mathbf{a} onto \mathbf{b} : $\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{b}|}$
4. Angle between vectors: $\arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \cdot |\mathbf{b}|}\right)$
5. From the previous point we may see that the dot product is positive if the angle between them is acute, negative if it is obtuse and it equals zero if they are orthogonal, i.e. they form a right angle.

Note that all these functions do not depend on the number of dimensions, hence they will be the same for the 2D and 3D case:

```
ftype norm(point2d a) {
    return dot(a, a);
}
double abs(point2d a) {
    return sqrt(norm(a));
}
double proj(point2d a, point2d b) {
    return dot(a, b) / abs(b);
}
double angle(point2d a, point2d b) {
    return acos(dot(a, b) / abs(a) / abs(b));
}
```

To see the next important property we should take a look at the set of points \mathbf{r} for which $\mathbf{r} \cdot \mathbf{a} = C$ for some fixed constant C . You can see that this set of points is exactly the set of points for which the projection onto \mathbf{a} is the point $C \cdot \frac{\mathbf{a}}{|\mathbf{a}|}$ and they form a hyperplane orthogonal to \mathbf{a} . You can see the vector \mathbf{a} alongside with several such vectors having same dot product with it in 2D on the picture below:

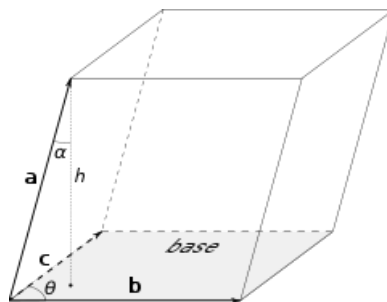


In 2D these vectors will form a line, in 3D they will form a plane. Note that this result allows us to define a line in 2D as $\mathbf{r} \cdot \mathbf{n} = C$ or $(\mathbf{r} - \mathbf{r}_0) \cdot \mathbf{n} = 0$ where \mathbf{n} is vector orthogonal to the line and \mathbf{r}_0 is any vector already present on the line and $C = \mathbf{r}_0 \cdot \mathbf{n}$. In the same manner a plane can be defined in 3D.

1.1.3 Cross product

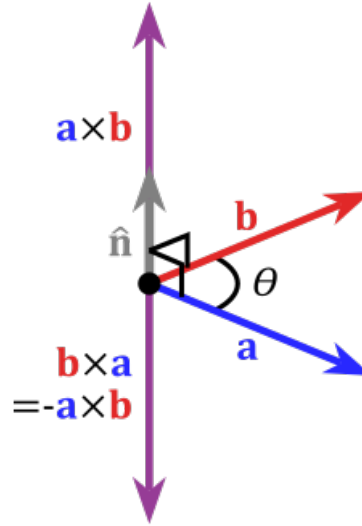
Definition

Assume you have three vectors \mathbf{a} , \mathbf{b} and \mathbf{c} in 3D space joined in a parallelepiped as in the picture below:



How would you calculate its volume? From school we know that we should multiply the area of the base with the height, which is projection of \mathbf{a} onto

direction orthogonal to base. That means that if we define $\mathbf{b} \times \mathbf{c}$ as the vector which is orthogonal to both \mathbf{b} and \mathbf{c} and which length is equal to the area of the parallelogram formed by \mathbf{b} and \mathbf{c} then $|\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})|$ will be equal to the volume of the parallelepiped. For integrity we will say that $\mathbf{b} \times \mathbf{c}$ will be always directed in such way that the rotation from the vector \mathbf{b} to the vector \mathbf{c} from the point of $\mathbf{b} \times \mathbf{c}$ is always counter-clockwise (see the picture below).



This defines the cross (or vector) product $\mathbf{b} \times \mathbf{c}$ of the vectors \mathbf{b} and \mathbf{c} and the triple product $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$ of the vectors \mathbf{a} , \mathbf{b} and \mathbf{c} .

Some notable properties of cross and triple products:

1. $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$
2. $(\alpha \cdot \mathbf{a}) \times \mathbf{b} = \alpha \cdot (\mathbf{a} \times \mathbf{b})$
3. For any \mathbf{b} and \mathbf{c} there is exactly one vector \mathbf{r} such that $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = \mathbf{a} \cdot \mathbf{r}$ for any vector \mathbf{a} . Indeed if there are two such vectors \mathbf{r}_1 and \mathbf{r}_2 then $\mathbf{a} \cdot (\mathbf{r}_1 - \mathbf{r}_2) = 0$ for all vectors \mathbf{a} which is possible only when $\mathbf{r}_1 = \mathbf{r}_2$.
4. $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = \mathbf{b} \cdot (\mathbf{c} \times \mathbf{a}) = -\mathbf{a} \cdot (\mathbf{c} \times \mathbf{b})$
5. $(\mathbf{a} + \mathbf{b}) \times \mathbf{c} = \mathbf{a} \times \mathbf{c} + \mathbf{b} \times \mathbf{c}$. Indeed for all vectors \mathbf{r} the chain of equations holds:

$$\mathbf{r} \cdot ((\mathbf{a} + \mathbf{b}) \times \mathbf{c}) = (\mathbf{a} + \mathbf{b}) \cdot (\mathbf{c} \times \mathbf{r}) = \mathbf{a} \cdot (\mathbf{c} \times \mathbf{r}) + \mathbf{b} \cdot (\mathbf{c} \times \mathbf{r}) = \mathbf{r} \cdot (\mathbf{a} \times \mathbf{c}) + \mathbf{r} \cdot (\mathbf{b} \times \mathbf{c}) = \mathbf{r} \cdot (\mathbf{a} \times \mathbf{c} + \mathbf{b} \times \mathbf{c})$$

Which proves $(\mathbf{a} + \mathbf{b}) \times \mathbf{c} = \mathbf{a} \times \mathbf{c} + \mathbf{b} \times \mathbf{c}$ due to point 3.

6. $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| \cdot |\mathbf{b}| \sin \theta$ where θ is angle between \mathbf{a} and \mathbf{b} , since $|\mathbf{a} \times \mathbf{b}|$ equals to the area of the parallelogram formed by \mathbf{a} and \mathbf{b} .

Given all this and that the following equation holds for the unit vectors

$$\mathbf{e}_x \times \mathbf{e}_x = \mathbf{e}_y \times \mathbf{e}_y = \mathbf{e}_z \times \mathbf{e}_z = \mathbf{0}, \mathbf{e}_x \times \mathbf{e}_y = \mathbf{e}_z, \mathbf{e}_y \times \mathbf{e}_z = \mathbf{e}_x, \mathbf{e}_z \times \mathbf{e}_x = \mathbf{e}_y$$

we can calculate the cross product of $\mathbf{a} = (x_1; y_1; z_1)$ and $\mathbf{b} = (x_2; y_2; z_2)$ in coordinate form:

$$\begin{aligned}\mathbf{a} \times \mathbf{b} &= (x_1 \cdot \mathbf{e}_x + y_1 \cdot \mathbf{e}_y + z_1 \cdot \mathbf{e}_z) \times (x_2 \cdot \mathbf{e}_x + y_2 \cdot \mathbf{e}_y + z_2 \cdot \mathbf{e}_z) = \\ &= (y_1 z_2 - z_1 y_2) \mathbf{e}_x + (z_1 x_2 - x_1 z_2) \mathbf{e}_y + (x_1 y_2 - y_1 x_2) \mathbf{e}_z\end{aligned}$$

Which also can be written in the more elegant form:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{e}_x & \mathbf{e}_y & \mathbf{e}_z \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}, \quad a \cdot (b \times c) = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

Here $|\cdot|$ stands for the determinant of a matrix.

Some kind of cross product (namely the pseudo-scalar product) can also be implemented in the 2D case. If we would like to calculate the area of parallelogram formed by vectors \mathbf{a} and \mathbf{b} we would compute $|\mathbf{e}_z \cdot (\mathbf{a} \times \mathbf{b})| = |x_1 y_2 - y_1 x_2|$. Another way to obtain the same result is to multiply $|\mathbf{a}|$ (base of parallelogram) with the height, which is the projection of vector \mathbf{b} onto vector \mathbf{a} rotated by 90° which in turn is $\hat{\mathbf{a}} = (-y_1; x_1)$. That is, to calculate $|\hat{\mathbf{a}} \cdot \mathbf{b}| = |x_1 y_2 - y_1 x_2|$.

If we will take the sign into consideration then the area will be positive if the rotation from \mathbf{a} to \mathbf{b} (i.e. from the view of the point of \mathbf{e}_z) is performed counter-clockwise and negative otherwise. That defines the pseudo-scalar product. Note that it also equals $|\mathbf{a}| \cdot |\mathbf{b}| \sin \theta$ where θ is angle from \mathbf{a} to \mathbf{b} count counter-clockwise (and negative if rotation is clockwise).

Let's implement all this stuff!

```
point3d cross(point3d a, point3d b) {
    return point3d(a.y * b.z - a.z * b.y,
                   a.z * b.x - a.x * b.z,
                   a.x * b.y - a.y * b.x);
}
ftype triple(point3d a, point3d b, point3d c) {
    return dot(a, cross(b, c));
}
ftype cross(point2d a, point2d b) {
    return a.x * b.y - a.y * b.x;
}
```

Properties

As for the cross product, it equals to the zero vector iff the vectors \mathbf{a} and \mathbf{b} are collinear (they form a common line, i.e. they are parallel). The same thing holds for the triple product, it is equal to zero iff the vectors \mathbf{a} , \mathbf{b} and \mathbf{c} are coplanar (they form a common plane).

From this we can obtain universal equations defining lines and planes. A line can be defined via its direction vector \mathbf{d} and an initial point \mathbf{r}_0 or by two points \mathbf{a} and \mathbf{b} . It is defined as $(\mathbf{r} - \mathbf{r}_0) \times \mathbf{d} = 0$ or as $(\mathbf{r} - \mathbf{a}) \times (\mathbf{b} - \mathbf{a}) = 0$. As for planes, it can be defined by three points \mathbf{a} , \mathbf{b} and \mathbf{c} as $(\mathbf{r} - \mathbf{a}) \cdot ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) = 0$ or by initial point \mathbf{r}_0 and two direction vectors lying in this plane \mathbf{d}_1 and \mathbf{d}_2 : $(\mathbf{r} - \mathbf{r}_0) \cdot (\mathbf{d}_1 \times \mathbf{d}_2) = 0$.

In 2D the pseudo-scalar product also may be used to check the orientation between two vectors because it is positive if the rotation from the first to the second vector is clockwise and negative otherwise. And, of course, it can be used to calculate areas of polygons, which is described in a different article. A triple product can be used for the same purpose in 3D space.

1.1.4 Exercises

Line intersection

There are many possible ways to define a line in 2D and you shouldn't hesitate to combine them. For example we have two lines and we want to find their intersection points. We can say that all points from first line can be parameterized as $\mathbf{r} = \mathbf{a}_1 + t \cdot \mathbf{d}_1$ where \mathbf{a}_1 is initial point, \mathbf{d}_1 is direction and t is some real parameter. As for second line all its points must satisfy $(\mathbf{r} - \mathbf{a}_2) \times \mathbf{d}_2 = 0$. From this we can easily find parameter t :

$$(\mathbf{a}_1 + t \cdot \mathbf{d}_1 - \mathbf{a}_2) \times \mathbf{d}_2 = 0 \quad \Rightarrow \quad t = \frac{(\mathbf{a}_2 - \mathbf{a}_1) \times \mathbf{d}_2}{\mathbf{d}_1 \times \mathbf{d}_2}$$

Let's implement function to intersect two lines.

```
point2d intersect(point2d a1, point2d d1, point2d a2, point2d d2) {
    return a1 + cross(a2 - a1, d2) / cross(d1, d2) * d1;
}
```

Planes intersection

However sometimes it might be hard to use some geometric insights. For example, you're given three planes defined by initial points \mathbf{a}_i and directions \mathbf{d}_i and you want to find their intersection point. You may note that you just have to solve the system of equations:

$$\begin{cases} \mathbf{r} \cdot \mathbf{n}_1 = \mathbf{a}_1 \cdot \mathbf{n}_1, \\ \mathbf{r} \cdot \mathbf{n}_2 = \mathbf{a}_2 \cdot \mathbf{n}_2, \\ \mathbf{r} \cdot \mathbf{n}_3 = \mathbf{a}_3 \cdot \mathbf{n}_3 \end{cases}$$

Instead of thinking on geometric approach, you can work out an algebraic one which can be obtained immediately. For example, given that you already implemented a point class, it will be easy for you to solve this system using Cramer's rule because the triple product is simply the determinant of the matrix obtained from the vectors being its columns:

```
point3d intersect(point3d a1, point3d n1, point3d a2, point3d n2, point3d a3, point3d n3) {
    point3d x(n1.x, n2.x, n3.x);
    point3d y(n1.y, n2.y, n3.y);
    point3d z(n1.z, n2.z, n3.z);
    point3d d(dot(a1, n1), dot(a2, n2), dot(a3, n3));
    return point3d(triple(d, y, z),
                   triple(x, d, z),
                   triple(x, y, d));
}
```

```
        triple(x, y, d)) / triple(n1, n2, n3);  
}
```

Now you may try to find out approaches for common geometric operations yourself to get used to all this stuff.

1.2 Finding the equation of a line for a segment

The task is: given the coordinates of the ends of a segment, construct a line passing through it.

We assume that the segment is non-degenerate, i.e. has a length greater than zero (otherwise, of course, infinitely many different lines pass through it).

Two-dimensional case

Let the given segment be PQ i.e. the known coordinates of its ends P_x, P_y, Q_x, Q_y .

It is necessary to construct **the equation of a line in the plane** passing through this segment, i.e. find the coefficients A, B, C in the equation of a line:

$$Ax + By + C = 0.$$

Note that for the required triples (A, B, C) there are **infinitely many** solutions which describe the given segment: you can multiply all three coefficients by an arbitrary non-zero number and get the same straight line. Therefore, our task is to find one of these triples.

It is easy to verify (by substitution of these expressions and the coordinates of the points P and Q into the equation of a straight line) that the following set of coefficients fits:

$$\begin{aligned} A &= P_y - Q_y, \\ B &= Q_x - P_x, \\ C &= -AP_x - BP_y. \end{aligned}$$

Integer case

An important advantage of this method of constructing a straight line is that if the coordinates of the ends were integer, then the coefficients obtained will also be **integer**. In some cases, this allows one to perform geometric operations without resorting to real numbers at all.

However, there is a small drawback: for the same straight line different triples of coefficients can be obtained. To avoid this, but do not go away from the integer coefficients, you can apply the following technique, often called **rationing**. Find the **greatest common divisor** of numbers $|A|, |B|, |C|$, we divide all three coefficients by it, and then we make the normalization of the sign: if $A < 0$ or $A = 0, B < 0$ then multiply all three coefficients by -1 . As a result, we will come to the conclusion that for identical straight lines, identical triples of coefficients will be obtained, which makes it easy to check straight lines for equality.

Real case

When working with real numbers, you should always be aware of errors.

The coefficients A and B will have the order of the original coordinates, the coefficient C is of the order of the square of them. This may already be quite large numbers, and, for example, when we [intersect straight lines](#), they will become even larger, which can lead to large rounding errors already when the coordinates of the end points are of order 10^3 .

Therefore, when working with real numbers, it is desirable to produce the so-called **normalization**, this is straightforward: namely, to make the coefficients such that $A^2 + B^2 = 1$. To do this, calculate the number Z :

$$Z = \sqrt{A^2 + B^2},$$

and divide all three coefficients A, B, C by it.

Thus, the order of the coefficients A and B will not depend on the order of the input coordinates, and the coefficient 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 straight lines - in fact, after such a normalization, for the same straight line, only two triples of coefficients can be obtained: up to multiplication by -1 . Accordingly, if we make an additional normalization taking into account the sign (if $A < -\varepsilon$ or $|A| < \varepsilon$, $B < -\varepsilon$ then multiply by -1), the resulting coefficients will be unique.

Three-dimensional and multidimensional case

Already in the three-dimensional case there is **no simple equation** describing a straight line (it can be defined as the intersection of two planes, that is, a system of two equations, but this is an inconvenient method).

Consequently, in the three-dimensional and multidimensional cases we must use the **parametric method of defining a straight line**, i.e. as a point p and a vector v :

$$p + vt, \quad t \in \mathbb{R}.$$

Those. a straight line is all points that can be obtained from a point p adding a vector v with an arbitrary coefficient.

The **construction** of a straight line in a parametric form along the coordinates of the ends of a segment is trivial, we just take one end of the segment for the point p , and the vector from the first to the second end — for the vector v .

1.3 Intersection Point of Lines

You are given two lines, described via the equations $a_1x + b_1y + c_1 = 0$ and $a_2x + b_2y + c_2 = 0$. We have to find the intersection point of the lines, or determine that the lines are parallel.

1.3.1 Solution

If two lines are not parallel, they intersect. To find their intersection point, we need to solve the following system of linear equations:

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$$

Using Cramer's rule, we can immediately write down the solution for the system, which will give us the required intersection point of the lines:

$$x = -\frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{c_1b_2 - c_2b_1}{a_1b_2 - a_2b_1},$$

$$y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}.$$

If the denominator equals 0, i.e.

$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1b_2 - a_2b_1 = 0$$

then either the system has no solutions (the lines are parallel and distinct) or there are infinitely many solutions (the lines overlap). If we need to distinguish these two cases, we have to check if coefficients c are proportional with the same ratio as the coefficients a and b . To do that we only have calculate the following determinants, and if they both equal 0, the lines overlap:

$$\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}, \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}$$

Notice, a different approach for computing the intersection point is explained in the article [Basic Geometry](#).

1.3.2 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;
}
```

1.4 Check if two segments intersect

You are given two segments (a, b) and (c, d) . You have to check if they intersect. Of course, you may find their intersection and check if it isn't empty, but this can't be done in integers for segments with integer coordinates. The approach described here can work in integers.

1.4.1 Algorithm

Firstly, consider the case when the segments are part of the same line. In this case it is sufficient to check if their projections on Ox and Oy intersect. In the other case a and b must not lie on the same side of line (c, d) , and c and d must not lie on the same side of line (a, b) . It can be checked with a couple of cross products.

1.4.2 Implementation

The given algorithm is implemented for integer points. Of course, it can be easily modified to work with doubles.

```
struct pt {
    long long x, y;
    pt() {}
    pt(long long _x, long long _y) : x(_x), y(_y) {}
    pt operator-(const pt& p) const { return pt(x - p.x, y - p.y); }
    long long cross(const pt& p) const { return x * p.y - y * p.x; }
    long long cross(const pt& a, const pt& b) const { return (a - *this).cross(b - *this); }
};

int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1; }

bool inter1(long long a, long long b, long long c, long long d) {
    if (a > b)
        swap(a, b);
    if (c > d)
        swap(c, d);
    return max(a, c) <= min(b, d);
}

bool check_inter(const pt& a, const pt& b, const pt& c, const pt& d) {
    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);
    return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
           sgn(c.cross(d, a)) != sgn(c.cross(d, b));
}
```

1.5 Finding Intersection of Two Segments

You are given two segments AB and CD, described as pairs of their endpoints. Each segment can be a single point if its endpoints are the same. You have to find the intersection of these segments, which can be empty (if the segments don't intersect), a single point or a segment (if the given segments overlap).

1.5.1 Solution

We can find the intersection point of segments in the same way as [the intersection of lines](#): reconstruct line equations from the segments' endpoints and check whether they are parallel.

If the lines are not parallel, we need to find their point of intersection and check whether it belongs to both segments (to do this it's sufficient to verify that the intersection point belongs to each segment projected on X and Y axes). In this case the answer will be either "no intersection" or the single point of lines' intersection.

The case of parallel lines is slightly more complicated (the case of one or more segments being a single point also belongs here). In this case we need to check that both segments belong to the same line. If they don't, the answer is "no intersection". If they do, the answer is the intersection of the segments belonging to the same line, which is obtained by ordering the endpoints of both segments in the increasing order of certain coordinate and taking the rightmost of left endpoints and the leftmost of right endpoints.

If both segments are single points, these points have to be identical, and it makes sense to perform this check separately.

In the beginning of the algorithm let's add a bounding box check - it is necessary for the case when the segments belong to the same line, and (being a lightweight check) it allows the algorithm to work faster on average on random tests.

1.5.2 Implementation

Here is the implementation, including all helper functions for lines and segments processing.

The main function `intersect` returns true if the segments have a non-empty intersection, and stores endpoints of the intersection segment in arguments `left` and `right`. If the answer is a single point, the values written to `left` and `right` 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; }
};

double det(double a, double b, double c, double d)
{
    return 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);
}
}
```

1.6 Circle-Line Intersection

Given the coordinates of the center of a circle and its radius, and the equation of a line, you're required to find the points of intersection.

1.6.1 Solution

Instead of solving the system of two equations, we will approach the problem geometrically. This way we get a more accurate solution from the point of view of numerical stability.

We assume without loss of generality that the circle is centered at the origin. If it's not, we translate it there and correct the C constant in the line equation. So we have a circle centered at $(0,0)$ of radius r and a line with equation $Ax + By + C = 0$.

Let's start by find the point on the line which is closest to the origin (x_0, y_0) . First, it has to be at a distance

$$d_0 = \frac{|C|}{\sqrt{A^2 + B^2}}$$

Second, since the vector (A, B) is perpendicular to the line, the coordinates of the point must be proportional to the coordinates of this vector. Since we know the distance of the point to the origin, we just need to scale the vector (A, B) to this length, and we'll get:

$$x_0 = -\frac{AC}{A^2 + B^2}$$

$$y_0 = -\frac{BC}{A^2 + B^2}$$

The minus signs are not obvious, but they can be easily verified by substituting x_0 and y_0 in the equation of the line.

At this stage we can determine the number of intersection points, and even find the solution when there is one or zero points. Indeed, if the distance from (x_0, y_0) to the origin d_0 is greater than the radius r , the answer is **zero points**. If $d_0 = r$, the answer is **one point** (x_0, y_0) . If $d_0 < r$, there are two points of intersection, and now we have to find their coordinates.

So, we know that the point (x_0, y_0) is inside the circle. The two points of intersection, (a_x, a_y) and (b_x, b_y) , must belong to the line $Ax + By + C = 0$ and must be at the same distance d from (x_0, y_0) , and this distance is easy to find:

$$d = \sqrt{r^2 - \frac{C^2}{A^2 + B^2}}$$

Note that the vector $(-B, A)$ is collinear to the line, and thus we can find the points in question by adding and subtracting vector $(-B, A)$, scaled to the length d , to the point (x_0, y_0) .

Finally, the equations of the two points of intersection are:

$$m = \sqrt{\frac{d^2}{A^2 + B^2}}$$

$$a_x = x_0 + B \cdot m, a_y = y_0 - A \cdot m$$

$$b_x = x_0 - B \cdot m, b_y = y_0 + A \cdot m$$

Had we solved the original system of equations using algebraic methods, we would likely get an answer in a different form with a larger error. The geometric method described here is more graphic and more accurate.

1.6.2 Implementation

As indicated at the outset, we assume that the circle is centered at the origin, and therefore the input to the program is the radius r of the circle and the parameters A , B and C of the equation of the line.

```
double r, a, b, c; // given as 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';
}
```

1.6.3 Practice Problems

- [CODECHEF: ANDOOR](#)

1.7 Circle-Circle Intersection

You are given two circles on a 2D plane, each one described as coordinates of its center and its radius. Find the points of their intersection (possible cases: one or two points, no intersection or circles coincide).

1.7.1 Solution

Let's reduce this problem to the [circle-line intersection problem](#).

Assume without loss of generality that the first circle is centered at the origin (if this is not true, we can move the origin to the center of the first circle and adjust the coordinates of intersection points accordingly at output time). 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 first equation from the second one to get rid of the second powers of variables:

$$\begin{aligned}x^2 + y^2 &= r_1^2 \\x \cdot (-2x_2) + y \cdot (-2y_2) + (x_2^2 + y_2^2 + r_1^2 - r_2^2) &= 0\end{aligned}$$

Thus, we've reduced the original problem to the problem of finding intersections of the first circle and a line:

$$\begin{aligned}Ax + By + C &= 0 \\A &= -2x_2 \\B &= -2y_2 \\C &= x_2^2 + y_2^2 + r_1^2 - r_2^2\end{aligned}$$

And this problem can be solved as described in the [corresponding article](#).

The only degenerate case we need to consider separately is when the centers of the circles coincide. In this case $x_2 = y_2 = 0$, and the line equation will be $C = r_1^2 - r_2^2 = 0$. If the radii of the circles are the same, there are infinitely many intersection points, if they differ, there are no intersections.

1.7.2 Practice Problems

- [RadarFinder](#)
- [Runaway to a shadow - Codeforces Round #357](#)
- [ASC 1 Problem F "Get out!"](#)
- [SPOJ: CIRCINT](#)
- [UVA - 10301 - Rings and Glue](#)

1.8 Length of the union of segments

Given n segments on a line, each described by a pair of coordinates (a_{i1}, a_{i2}) . We have to find the length of their union.

The following algorithm was proposed by Klee in 1977. It works in $O(n \log n)$ and has been proven to be the asymptotically optimal.

1.8.1 Solution

We store in an array x the endpoints of all the segments sorted by their values. And additionally we store whether it is a left end or a right end of a segment. Now we iterate over the array, keeping a counter c of currently opened segments. Whenever the current element is a left end, we increase this counter, and otherwise we decrease it. To compute the answer, we take the length between the last to x values $x_i - x_{i-1}$, whenever we come to a new coordinate, and there is currently at least one segment is open.

1.8.2 Implementation

```
int length_union(const vector<pair<int, int>> &a) {
    int n = a.size();
    vector<pair<int, bool>> x(n*2);
    for (int i = 0; i < n; i++) {
        x[i*2] = {a[i].first, false};
        x[i*2+1] = {a[i].second, true};
    }

    sort(x.begin(), x.end());

    int result = 0;
    int c = 0;
    for (int i = 0; i < n * 2; i++) {
        if (i > 0 && x[i].first > x[i-1].first && c > 0)
            result += x[i].first - x[i-1].first;
        if (x[i].second)
            c++;
        else
            --c;
    }
    return result;
}
```

Chapter 2

Polygons

2.1 Oriented area of a triangle

Given three points p_1 , p_2 and p_3 , calculate an oriented (signed) area of a triangle formed by them. The sign of the area is determined in the following way: imagine you are standing in the plane at point p_1 and are facing p_2 . You go to p_2 and if p_3 is to your right (then we say the three vectors turn “clockwise”), the sign of the area is positive, otherwise it is negative. If the three points are collinear, the area is zero.

Using this signed area, we can both get the regular unsigned area (as the absolute value of the signed area) and determine if the points lie clockwise or counterclockwise in their specified order (which is useful, for example, in convex hull algorithms).

2.1.1 Calculation

We can use the fact that a determinant of a 2×2 matrix is equal to the signed area of a parallelogram spanned by column (or row) vectors of the matrix. This is analog to the definition of the cross product in 2D (see [Basic Geometry](#)). By dividing this area by two we get the area of a triangle that we are interested in. We will use $p_1\vec{p}_2$ and $p_2\vec{p}_3$ as the column vectors and calculate a 2×2 determinant:

$$2S = \begin{vmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{vmatrix} = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)$$

2.1.2 Implementation

```
int signed_area_parallelogram(point2d p1, point2d p2, point2d p3) {
    return cross(p2 - p1, p3 - p1);
}

double triangle_area(point2d p1, point2d p2, point2d p3) {
    return abs(signed_area_parallelogram(p1, p2, p3)) / 2.0;
}
```

```
bool clockwise(point2d p1, point2d p2, point2d p3) {  
    return signed_area_parallelogram(p1, p2, p3) < 0;  
}  
  
bool counter_clockwise(point2d p1, point2d p2, point2d p3) {  
    return signed_area_parallelogram(p1, p2, p3) > 0;  
}
```

2.1.3 Practice Problems

- [Codechef - Chef and Polygons](#)

2.2 Finding area of simple polygon in $O(N)$

Let a simple polygon (i.e. without self intersection, not necessarily convex) be given. It is required to calculate its area given its vertices.

2.2.1 Method 1

This is easy to do if we go through all edges and add trapezoid areas bounded by each edge and x-axis. The area needs to be taken with sign so that the extra area will be reduced. Hence, the formula is as follows:

$$A = \sum_{(p,q) \in \text{edges}} \frac{(p_x - q_x) \cdot (p_y + q_y)}{2}$$

Code:

```
double area(const vector<point>& fig) {
    double res = 0;
    for (unsigned i = 0; i < fig.size(); i++) {
        point p = i ? fig[i - 1] : fig.back();
        point q = fig[i];
        res += (p.x - q.x) * (p.y + q.y);
    }
    return fabs(res) / 2;
}
```

2.2.2 Method 2

We can choose a point O arbitrarily, iterate over all edges adding the oriented area of the triangle formed by the edge and point O . Again, due to the sign of area, extra area will be reduced.

This method is better as it can be generalized to more complex cases (such as when some sides are arcs instead of straight lines)

2.3 Check if point belongs to the convex polygon in $O(\log N)$

Consider the following problem: you are given a convex polygon with integer vertices and a lot of queries. Each query is a point, for which we should determine whether it lies inside or on the boundary of the polygon or not. Suppose the polygon is ordered counter-clockwise. We will answer each query in $O(\log n)$ online.

2.3.1 Algorithm

Let's pick the point with the smallest x-coordinate. If there are several of them, we pick the one with the smallest y-coordinate. Let's denote it as p_0 . Now all other points p_1, \dots, p_n of the polygon are ordered by their polar angle from the chosen point (because the polygon is ordered counter-clockwise).

If the point belongs to the polygon, it belongs to some triangle p_0, p_i, p_{i+1} (maybe more than one if it lies on the boundary of triangles). Consider the triangle p_0, p_i, p_{i+1} such that p belongs to this triangle and i is maximum among all such triangles.

There is one special case. p lies on the segment (p_0, p_n) . This case we will check separately. Otherwise all points p_j with $j \leq i$ are counter-clockwise from p with respect to p_0 , and all other points are not counter-clockwise from p . This means that we can apply binary search for the point p_i , such that p_i is not counter-clockwise from p with respect to p_0 , and i is maximum among all such points. And afterwards we check if the point is actually in the determined triangle.

The sign of $(a - c) \times (b - c)$ will tell us, if the point a is clockwise or counter-clockwise from the point b with respect to the point c . If $(a - c) \times (b - c) > 0$, then the point a is to the right of the vector going from c to b , which means clockwise from b with respect to c . And if $(a - c) \times (b - c) < 0$, then the point is to the left, or counter clockwise. And it is exactly on the line between the points b and c .

Back to the algorithm: Consider a query point p . Firstly, we must check if the point lies between p_1 and p_n . Otherwise we already know that it cannot be part of the polygon. This can be done by checking if the cross product $(p_1 - p_0) \times (p - p_0)$ is zero or has the same sign with $(p_1 - p_0) \times (p_n - p_0)$, and $(p_n - p_0) \times (p - p_0)$ is zero or has the same sign with $(p_n - p_0) \times (p_1 - p_0)$. Then we handle the special case in which p is part of the line (p_0, p_1) . And then we can binary search the last point from p_1, \dots, p_n which is not counter-clockwise from p with respect to p_0 . For a single point p_i this condition can be checked by checking that $(p_i - p_0) \times (p - p_0) \leq 0$. After we found such a point p_i , we must test if p lies inside the triangle p_0, p_i, p_{i+1} . To test if it belongs to the triangle, we may simply check that $|(p_i - p_0) \times (p_{i+1} - p_0)| = |(p_0 - p) \times (p_i - p)| + |(p_i - p) \times (p_{i+1} - p)| + |(p_{i+1} - p) \times (p_0 - p)|$. This checks if the area of the triangle p_0, p_i, p_{i+1} has to exact same size as the sum of the sizes of the triangle p_0, p_i, p , the triangle p_0, p, p_{i+1} and the triangle p_i, p_{i+1}, p . If

p is outside, then the sum of those three triangle will be bigger than the size of the triangle. If it is inside, then it will be equal.

2.3.2 Implementation

The function `prepair` will make sure that the lexicographical smallest point (smallest x value, and in ties smallest y value) will be p_0 , and computes the vectors $p_i - p_0$. Afterwards the function `pointInConvexPolygon` computes the result of a query.

```

struct pt{
    long long x, y;
    pt(){}
    pt(long long _x, long long _y):x(_x), y(_y){}
    pt operator+(const pt & p) const { return pt(x + p.x, y + p.y); }
    pt operator-(const pt & p) const { return pt(x - p.x, y - p.y); }
    long long cross(const pt & p) const { return x * p.y - y * p.x; }
    long long dot(const pt & p) const { return x * p.x + y * p.y; }
    long long cross(const pt & a, const pt & b) const { return (a - *this).cross(b - *this); }
    long long dot(const pt & a, const pt & b) const { return (a - *this).dot(b - *this); }
    long long sqrLen() const { return this->dot(*this); }
};

bool lexComp(const pt & l, const pt & r){
    return l.x < r.x || (l.x == r.x && l.y < r.y);
}

int sgn(long long val){
    return val > 0 ? 1 : (val == 0 ? 0 : -1);
}

vector<pt> seq;
int n;

bool pointInTriangle(pt a, pt b, pt c, pt point){
    long long s1 = abs(a.cross(b, c));
    long long s2 = abs(point.cross(a, b)) + abs(point.cross(b, c)) + abs(point.cross(c, a));
    return s1 == s2;
}

void prepare(vector<pt> & points){
    n = points.size();
    int pos = 0;
    for(int i = 1; i < n; i++){
        if(lexComp(points[i], points[pos]))
            pos = i;
    }
    rotate(points.begin(), points.begin() + pos, points.end());

    n--;
    seq.resize(n);
    for(int i = 0; i < n; i++)

```

```

        seq[i] = points[i + 1] - points[0];
    }

    bool pointInConvexPolygon(pt point){
        if(seq[0].cross(point) != 0 && sgn(seq[0].cross(point)) != sgn(seq[0].cross(seq[n - 1])))
            return false;
        if(seq[n - 1].cross(point) != 0 && sgn(seq[n - 1].cross(point)) != sgn(seq[n - 1].cross(s
            return false;

        if(seq[0].cross(point) == 0)
            return seq[0].sqrLen() >= point.sqrLen();

        int l = 0, r = n - 1;
        while(r - l > 1){
            int mid = (l + r)/2;
            int pos = mid;
            if(seq[pos].cross(point) >= 0) l = mid;
            else r = mid;
        }
        int pos = l;
        return pointInTriangle(seq[pos], seq[pos + 1], pt(0, 0), point);
    }

```

2.3.3 Problems

[SGU253 Theodore Roosevelt](#)

2.4 Pick's Theorem

A polygon without self-intersections is called lattice if all its vertices have integer coordinates in some 2D grid. Pick's theorem provides a way to compute the area of this polygon through the number of vertices that are lying on the boundary and the number of vertices that lie strictly inside the polygon.

2.4.1 Formula

Given a certain lattice polygon with non-zero area.

We denote its area by S , the number of points with integer coordinates lying strictly inside the polygon by I and the number of points lying on polygon sides by B .

Then, the **Pick's formula** states:

$$S = I + \frac{B}{2} - 1$$

In particular, if the values of I and B for a polygon are given, the area can be calculated in $O(1)$ without even knowing the vertices.

This formula was discovered and proven by Austrian mathematician Georg Alexander Pick in 1899.

2.4.2 Proof

The proof is carried out in many stages: from simple polygons to arbitrary ones:

- A single square: $S = 1, I = 0, B = 4$, which satisfies the formula.
- An arbitrary non-degenerate rectangle with sides parallel to coordinate axes: Assume a and b be the length of the sides of rectangle. Then, $S = ab, I = (a - 1)(b - 1), B = 2(a + b)$. On substituting, we see that formula is true.
- A right angle with legs parallel to the axes: To prove this, note that any such triangle can be obtained by cutting off a rectangle by a diagonal. Denoting the number of integral points lying on diagonal by c , it can be shown that Pick's formula holds for this triangle regardless of c .
- An arbitrary triangle: Note that any such triangle can be turned into a rectangle by attaching it to sides of right-angled triangles with legs parallel to the axes (you will not need more than 3 such triangles). From here, we can get correct formula for any triangle.
- An arbitrary polygon: To prove this, triangulate it, ie, divide into triangles with integral coordinates. Further, it is possible to prove that Pick's theorem retains its validity when a polygon is added to a triangle. Thus, we have proven Pick's formula for arbitrary polygon.

2.4.3 Generalization to higher dimensions

Unfortunately, this simple and beautiful formula cannot be generalized to higher dimensions.

John Reeve demonstrated this by proposing a tetrahedron (**Reeve tetrahedron**) with following vertices in 1957:

$$A = (0, 0, 0), B = (1, 0, 0), C = (0, 1, 0), D = (1, 1, k),$$

where k can be any natural number. Then for any k , the tetrahedron $ABCD$ does not contain integer point inside it and has only 4 points on its borders, A, B, C, D . Thus, the volume and surface area may vary in spite of unchanged number of points within and on boundary. Therefore, Pick's theorem doesn't allow generalizations.

However, higher dimensions still has a generalization using **Ehrhart polynomials** but they are quite complex and depends not only on points inside but also on the boundary of polytype.

2.4.4 Extra Resources

A few simple examples and a simple proof of Pick's theorem can be found [here](#).

2.5 Lattice points inside non-lattice polygon

For lattice polygons there is Pick's formula to enumerate the lattice points inside the polygon. What about polygons with arbitrary vertices?

Let's process each of the polygon's edges individually, and after that we may sum up the amounts of lattice points under each edge considering its orientations to choose a sign (like in calculating the area of a polygon using trapezoids).

First of all we should note that if current edge has endpoints in $A = (x_1; y_1)$ and $B = (x_2; y_2)$ then it can be represented as a linear function:

$$y = y_1 + (y_2 - y_1) \cdot \frac{x - x_1}{x_2 - x_1} = \left(\frac{y_2 - y_1}{x_2 - x_1} \right) \cdot x + \left(\frac{y_1 x_2 - x_1 y_2}{x_2 - x_1} \right)$$

$$y = k \cdot x + b, \quad k = \frac{y_2 - y_1}{x_2 - x_1}, \quad b = \frac{y_1 x_2 - x_1 y_2}{x_2 - x_1}$$

Now we will perform a substitution $x = x' + \lceil x_1 \rceil$ so that $b' = b + k \cdot \lceil x_1 \rceil$. This allows us to work with $x'_1 = 0$ and $x'_2 = x_2 - \lceil x_1 \rceil$. Let's denote $n = \lfloor x'_2 \rfloor$.

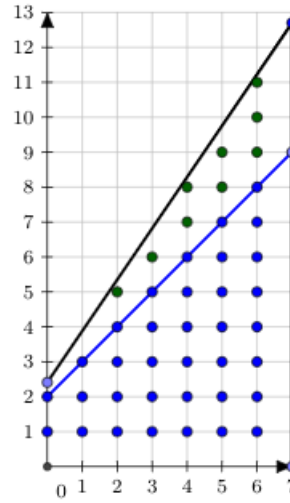
We will not sum up points at $x = n$ and on $y = 0$ for the integrity of the algorithm. They may be added manually afterwards. Thus we have to sum up $\sum_{x'=0}^{n-1} \lfloor k' \cdot x' + b' \rfloor$. We also assume that $k' \geq 0$ and $b' \geq 0$. Otherwise one should substitute $x' = -t$ and add $\lceil |b'| \rceil$ to b' .

Let's discuss how we can evaluate a sum $\sum_{x=0}^{n-1} \lfloor k \cdot x + b \rfloor$. We have two cases:

- $k \geq 1$ or $b \geq 1$. Then we should start with summing up points below $y = \lfloor k \rfloor \cdot x + \lfloor b \rfloor$. Their amount equals to

$$\sum_{x=0}^{n-1} \lfloor k \rfloor \cdot x + \lfloor b \rfloor = \frac{(\lfloor k \rfloor)(n-1) + 2\lfloor b \rfloor)n}{2}.$$

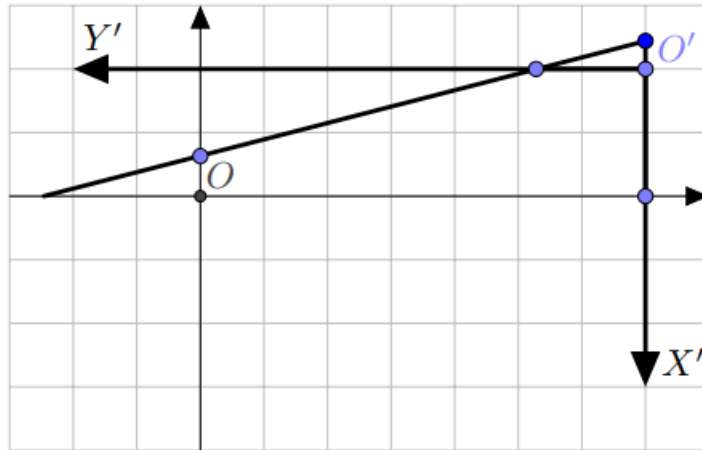
Now we are interested only in points $(x; y)$ such that $\lfloor k \rfloor \cdot x + \lfloor b \rfloor < y \leq k \cdot x + b$. This amount is the same as the number of points such that $0 < y \leq (k - \lfloor k \rfloor) \cdot x + (b - \lfloor b \rfloor)$. So we reduced our problem to $k' = k - \lfloor k \rfloor$, $b' = b - \lfloor b \rfloor$ and both k' and b' less than 1 now. Here is a picture, we just summed up blue points and subtracted the blue linear function from the black one to reduce problem to smaller values for k and b :



- $k < 1$ and $b < 1$. If $\lfloor k \cdot n + b \rfloor$ equals 0, we can safely return 0. If this is not the case, we can say that there are no lattice points such that $x < 0$ and $0 < y \leq k \cdot x + b$. That means that we will have the same answer if we consider new reference system in which $O' = (n; \lfloor k \cdot n + b \rfloor)$, axis x' is directed down and axis y' is directed to the left. For this reference system we are interested in lattice points on the set

$$\left\{ (x; y) \mid 0 \leq x < \lfloor k \cdot n + b \rfloor, 0 < y \leq \frac{x + (k \cdot n + b) - \lfloor k \cdot n + b \rfloor}{k} \right\}$$

which returns us back to the case $k > 1$. You can see new reference point O' and axes X' and Y' in the picture below:



As you see, in new reference system linear function will have coefficient $\frac{1}{k}$ and its zero will be in the point $\lfloor k \cdot n + b \rfloor - (k \cdot n + b)$ which makes formula above correct.

2.5.1 Complexity analysis

We have to count at most $\frac{(k(n-1) + 2b)n}{2}$ points. Among them we will count $\frac{\lfloor k \rfloor(n-1) + 2\lfloor b \rfloor}{2}$ on the very first step. We may consider that b is negligibly small because we can start with making it less than 1. In that case we can say that we count about $\frac{\lfloor k \rfloor}{k} \geq \frac{1}{2}$ of all points. Thus we will finish in $O(\log n)$ steps.

2.5.2 Implementation

Here is simple function which calculates number of integer points $(x; y)$ such for $0 \leq x < n$ and $0 < y \leq \lfloor kx + b \rfloor$:

```
int count_lattices(Fraction k, Fraction b, long long n) {
    auto fk = k.floor();
    auto fb = b.floor();
    auto cnt = 0LL;
    if (k >= 1 || b >= 1) {
        cnt += (fk * (n - 1) + 2 * fb) * n / 2;
        k -= fk;
        b -= fb;
    }
    auto t = k * n + b;
    auto ft = t.floor();
    if (ft >= 1) {
        cnt += count_lattices(1 / k, (t - t.floor()) / k, t.floor());
    }
    return cnt;
}
```

Here `Fraction` is some class handling rational numbers. On practice it seems that if all denominators and numerators are at most C by absolute value then in the recursive calls they will be at most C^2 if you keep dividing numerators and denominators by their greatest common divisor. Given this assumption we can say that one may use doubles and require accuracy of ε^2 where ε is accuracy with which k and b are given. That means that in floor one should consider numbers as integer if they differs at most by ε^2 from an integer.

Chapter 3

Convex hull

3.1 Convex Hull construction using Graham's Scan

In this article we will discuss the problem of constructing a convex hull from a set of points.

Consider N points given on a plane, and the objective is to generate a convex hull, i.e. the smallest convex polygon that contains all the given points.

The algorithm used here is **Graham's scan** (proposed in 1972 by Graham) with improvements by Andrew (1979). The algorithm allows for the construction of a convex hull in $O(N \log N)$ using only comparison, addition and multiplication operations. The algorithm is asymptotically optimal (as it is proven that there is no algorithm asymptotically better), with the exception of a few problems where parallel or online processing is involved.

3.1.1 Description

The algorithm first finds the leftmost and rightmost points A and B. In the event multiple such points exist, the lowest among the left (lowest Y-coordinate) is taken as A, and the highest among the right (highest Y-coordinate) is taken as B. Clearly, A and B must both belong to the convex hull as they are the farthest away and they cannot be contained by any line formed by a pair among the given points.

Now, draw a line through AB. This divides all the other points into two sets, S1 and S2, where S1 contains all the points above the line connecting A and B, and S2 contains all the points below the line joining A and B. The points that lie on the line joining A and B may belong to either set. The points A and B belong to both sets. Now the algorithm constructs the upper set S1 and the lower set S2 and then combines them to obtain the answer.

To get the upper set, we sort all points by the x-coordinate. For each point we check if either - the current point is the last point, (which we defined as B), or if the orientation between the line between A and the current point and the line between the current point and B is clockwise. In those cases the current point belongs to the upper set S1. Checking for clockwise or anticlockwise nature can be done by checking the [orientation](#).

If the given point belongs to the upper set, we check the angle made by the line connecting the current point and the last point in the upper convex hull, with the line connecting the last point and the second last point of the convex hull that we have been simultaneously creating. If the angle is not clockwise, we remove the most recent point added to the upper convex hull as the current point will be able to contain the previous point once it is added to the convex hull.

The same logic applies for the lower set S2. If either - the current point is B, or the orientation of the lines, formed by A and the current point and the current point and B, is counterclockwise - then it belongs to S2.

If the given point belongs to the lower set, we act similarly as for a point on the upper set except we check for a counterclockwise orientation instead of a clockwise orientation. Thus, if the angle made by the line connecting the current point and the last point in the lower convex hull, with the line connecting the last point and the second last point of the convex hull is not counterclockwise, we remove the most recent point added to the lower convex hull as the current point will be able to contain the previous point once added to the hull.

The final convex hull is obtained from the union of the upper and lower convex hull, and the implementation is as follows.

3.1.2 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 a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;
}

bool ccw(pt a, pt b, pt c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 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 (int i = 1; i < (int)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 (int i = 0; i < (int)up.size(); i++)
    a.push_back(up[i]);
for (int i = down.size() - 2; i > 0; i--)
    a.push_back(down[i]);
}
```

3.2 Convex hull trick and Li Chao tree

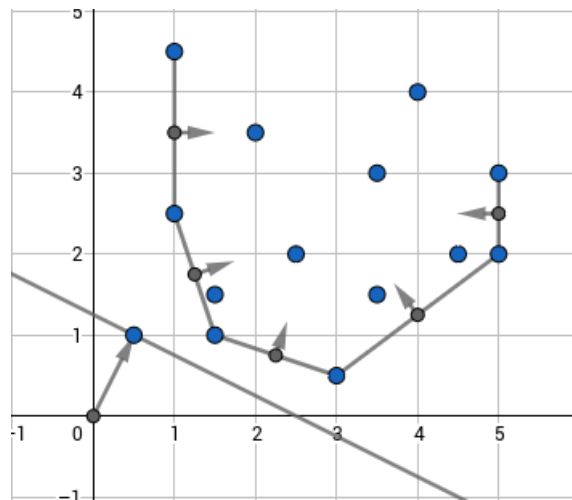
Consider the following problem. There are n cities. You want to travel from city 1 to city n by car. To do this you have to buy some gasoline. It is known that a liter of gasoline costs $cost_k$ in the k^{th} city. Initially your fuel tank is empty and you spend one liter of gasoline per kilometer. Cities are located on the same line in ascending order with k^{th} city having coordinate x_k . Also you have to pay $toll_k$ to enter k^{th} city. Your task is to make the trip with minimum possible cost. It's obvious that the solution can be calculated via dynamic programming:

$$dp_i = toll_i + \min_{j < i} (cost_j \cdot (x_i - x_j) + dp_j)$$

Naive approach will give you $O(n^2)$ complexity which can be improved to $O(n \log n)$ or $O(n \log[C\varepsilon^{-1}])$ where C is largest possible $|x_i|$ and ε is precision with which x_i is considered ($\varepsilon = 1$ for integers which is usually the case). To do this one should note that the problem can be reduced to adding linear functions $k \cdot x + b$ to the set and finding minimum value of the functions in some particular point x . There are two main approaches one can use here.

3.2.1 Convex hull trick

Idea of this approach is to maintain a lower convex hull of linear functions. Actually it would be a bit more convenient to consider them not as linear functions but as points $(k; b)$ on the plane such that we will have to find the point which has the least dot product with a given point $(x; 1)$, that is, for this point $kx + b$ is minimized which is the same as initial problem. Such minimum will necessarily be on lower convex envelope of these points as can be seen below:



One has to keep points in convex hull alongside with normal vectors of edges of convex hull. When you have $(x; 1)$ query you'll have to find normal vector closest to it in terms of angles between them, then optimum linear function will correspond to one of its endpoints. To see that one should note that points

having same dot product with $(x; 1)$ lie on same line which is orthogonal to $(x; 1)$ so optimum linear function will be the one in which tangent to convex hull which is collinear with normal to $(x; 1)$ touches the hull. This point can be found as one such that normals of edges lying to the left and to the right of it are headed in different sides of $(x; 1)$.

This approach is useful when queries of adding linear functions are monotone in terms of k or if we work offline, i.e. we may firstly add all linear functions and answer queries afterwards. So we cannot solve the cities/gasoline problems using this way. That would require handling online queries. When it comes to deal with online queries however, things will go tough and one will have to use some kind of set data structure to implement a proper convex hull. Online approach will however not be considered in this article due to its hardness and because second approach (which is Li Chao tree) allows to solve the problem way simpler. Worth mentioning that one can still use this approach in online without complications by square-root-decomposition. That is, rebuild convex hull from scratch each \sqrt{n} new lines.

To implement this approach one should begin with some wrap for points class, here we suggest to use C++ complex number type.

```
typedef int ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype cross(point a, point b) {
    return (conj(a) * b).y();
}
```

Here we will assume that when linear functions are added, their k only increases and we want to find minimum values. We will keep points in vector *hull* and normal vectors in vector *vecs*. When we add a new point, we have to look at the angle formed between last edge in convex hull and vector from last point in convex hull to new point. This angle have to be directed counter-clockwise, that is the dot product of the last normal vector in the hull (directed inside hull) and the vector from the last point to the new one has to be non-negative. As long as this isn't true, we should erase the last point in the convex hull alongside with the corresponding edge.

```
vector<point> hull, vecs;

void add_line(ftype k, ftype b) {
    point nw = {k, b};
    while(!vecs.empty() && dot(vecs.back(), nw - hull.back()) < 0) {
        hull.pop_back();
        vecs.pop_back();
    }
```

```

    }
    if(!hull.empty()) {
        vecs.push_back(1i * (nw - hull.back()));
    }
    hull.push_back(nw);
}

```

Now to get the minimum value in some point we will find the first normal vector in the convex hull that is directed counter-clockwise from $(x; 1)$. The left endpoint of such edge will be the answer. To check if vector a is not directed counter-clockwise of vector b , we should check if their cross product $[a, b]$ is positive.

```

int get(ftype x) {
    point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](point a, point b) {
        return cross(a, b) > 0;
    });
    return dot(query, hull[it - vecs.begin()]);
}

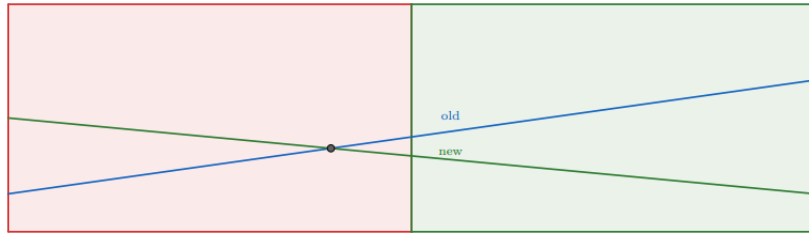
```

3.2.2 Li Chao tree

Assume you're given a set of functions such that each two can intersect at most once. Let's keep in each vertex of a segment tree some function in such way, that if we go from root to the leaf it will be guaranteed that one of the functions we met on the path will be the one giving the minimum value in that leaf. Let's see how to construct it.

Assume we're in some vertex corresponding to half-segment $[l; r)$ and the function f_{old} is kept there and we add the function f_{new} . Then the intersection point will be either in $[l; m)$ or in $[m; r)$ where $m = \lfloor \frac{l+r}{2} \rfloor$. We can efficiently find that out by comparing the values of the functions in points l and m . If the dominating function changes, then it is in $[l; m)$ otherwise it is in $[m; r)$. Now for the half of the segment with no intersection we will pick the lower function and write it in the current vertex. You can see that it will always be the one which is lower in point m . After that we recursively go to the other half of the segment with the function which was the upper one. As you can see this will keep correctness on the first half of segment and in the other one correctness will be maintained during the recursive call. Thus we can add functions and check the minimum value in the point in $O(\log[C\varepsilon^{-1}])$.

Here is the illustration of what is going on in the vertex when we add new function:



Let's go to implementation now. Once again we will use complex numbers to keep linear functions.

```
typedef int ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype f(point a, ftype x) {
    return dot(a, {x, 1});
}
```

We will keep functions in the array *line* and use binary indexing of the segment tree. If you want to use it on large numbers or doubles, you should use a dynamic segment tree. The segment tree should be initialized with default values, e.g. with lines $0x + \infty$.

```
const int maxn = 2e5;

point line[4 * maxn];

void add_line(point nw, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    bool lef = f(nw, l) < f(line[v], l);
    bool mid = f(nw, m) < f(line[v], m);
    if(mid) {
        swap(line[v], nw);
    }
    if(r - l == 1) {
        return;
    }
    else if(lef != mid) {
        add_line(nw, 2 * v, l, m);
    }
    else {
        add_line(nw, 2 * v + 1, m, r);
    }
}
```

Now to get the minimum in some point x we simply choose the minimum value along the path to the point.


```
int get(int x, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    if(r - l == 1) {
        return f(line[v], x);
    } else if(x < m) {
        return min(f(line[v], x), get(x, 2 * v, l, m));
    } else {
        return min(f(line[v], x), get(x, 2 * v + 1, m, r));
    }
}
```

3.2.3 Problems

- [CS Academy - Squared Ends](#)
- [Codeforces - Escape Through Leaf](#)
- [CodeChef - Polynomials](#)

Chapter 4

Sweep-line

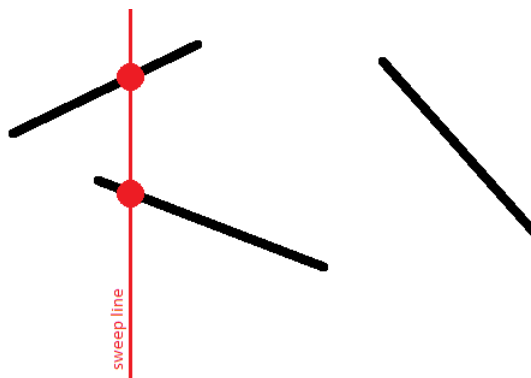
4.1 Search for a pair of intersecting segments

Given n line segments on the plane. It is required to check whether at least two of them intersect with each other. If the answer is yes, then print this pair of intersecting segments; it is enough to choose any of them among several answers.

The naive solution algorithm is to iterate over all pairs of segments in $O(n^2)$ and check for each pair whether they intersect or not. This article describes an algorithm with the runtime time $O(n \log n)$, which is based on the **sweep line algorithm**.

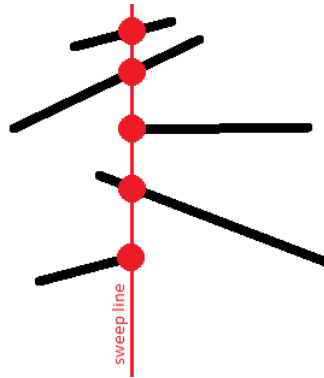
4.1.1 Algorithm

Let's draw a vertical line $x = -\infty$ mentally and start moving this line to the right. In the course of its movement, this line will meet with segments, and at each time a segment intersect with our line it intersects in exactly one point (we will assume that there are no vertical segments).

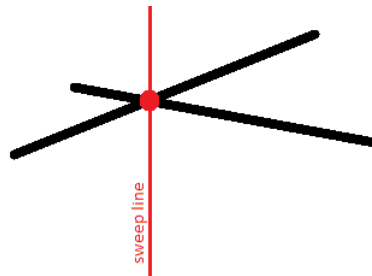


Thus, for each segment, at some point in time, its point will appear on the sweep line, then with the movement of the line, this point will move, and finally, at some point, the segment will disappear from the line.

We are interested in the **relative order of the segments** along the vertical. Namely, we will store a list of segments crossing the sweep line at a given time, where the segments will be sorted by their y -coordinate on the sweep line.



This order is interesting because intersecting segments will have the same y -coordinate at least at one time:



We formulate key statements:

- To find an intersecting pair, it is sufficient to consider **only adjacent segments** at each fixed position of the sweep line.
- It is enough to consider the sweep line not in all possible real positions $(-\infty \dots +\infty)$, but **only in those positions when new segments appear or old ones disappear**. In other words, it is enough to limit yourself only to the positions equal to the abscissas of the end points of the segments.
- When a new line segment appears, it is enough to **insert** it to the desired location in the list obtained for the previous sweep line. We should only check for the intersection of the **added segment with its immediate neighbors in the list above and below**.
- If the segment disappears, it is enough to **remove** it from the current list. After that, it is necessary **check for the intersection of the upper and lower neighbors in the list**.
- Other changes in the sequence of segments in the list, except for those described, do not exist. No other intersection checks are required.

To understand the truth of these statements, the following remarks are sufficient:

- Two disjoint segments never change their **relative order**. In fact, if one segment was first higher than the other, and then became lower, then between these two moments there was an intersection of these two segments.

- Two non-intersecting segments also cannot have the same y -coordinates.
- From this it follows that at the moment of the segment appearance we can find the position for this segment in the queue, and we will not have to rearrange this segment in the queue any more: **its order relative to other segments in the queue will not change.**
- Two intersecting segments at the moment of their intersection point will be neighbors of each other in the queue.
- Therefore, for finding pairs of intersecting line segments is sufficient to check the intersection of all and only those pairs of segments that sometime during the movement of the sweep line at least once were neighbors to each other. It is easy to notice that it is enough only to check the added segment with its upper and lower neighbors, as well as when removing the segment — its upper and lower neighbors (which after removal will become neighbors of each other).
- It should be noted that at a fixed position of the sweep line, we must **first add all the segments** that start at this x -coordinate, and only **then remove all the segments** that end here. Thus, we do not miss the intersection of segments on the vertex: i.e. such cases when two segments have a common vertex.
- Note that **vertical segments** do not actually affect the correctness of the algorithm. These segments are distinguished by the fact that they appear and disappear at the same time. However, due to the previous comment, we know that all segments will be added to the queue first, and only then they will be deleted. Therefore, if the vertical segment intersects with some other segment opened at that moment (including the vertical one), it will be detected. **In what place of the queue to place vertical segments?** After all, a vertical segment does not have one specific y -coordinate, it extends for an entire segment along the y -coordinate. However, it is easy to understand that any coordinate from this segment can be taken as a y -coordinate.

Thus, the entire algorithm will perform no more than $2n$ tests on the intersection of a pair of segments, and will perform $O(n)$ operations with a queue of segments ($O(1)$ operations at the time of appearance and disappearance of each segment).

The final **asymptotic behavior of the algorithm** is thus $O(n \log n)$.

4.1.2 Implementation

We present the full implementation of the described 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);
    }
};

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;
}

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;

set<seg>::iterator prev(set<seg>::iterator it) {
    return it == s.begin() ? s.end() : --it;
}

set<seg>::iterator next(set<seg>::iterator it) {
    return ++it;
}

pair<int, int> solve(const vector<seg>& a) {
    int n = (int)a.size();
    vector<event> e;
    for (int i = 0; i < n; ++i) {
        e.push_back(event(min(a[i].p.x, a[i].q.x), +1, i));
        e.push_back(event(max(a[i].p.x, a[i].q.x), -1, i));
    }
    sort(e.begin(), e.end());

    s.clear();
    where.resize(a.size());
    for (size_t i = 0; i < e.size(); ++i) {
        int id = e[i].id;
        if (e[i].tp == +1) {
            set<seg>::iterator nxt = s.lower_bound(a[id]), prv = prev(nxt);
            if (nxt != s.end() && intersect(*nxt, a[id]))
                return make_pair(nxt->id, id);
            if (prv != s.end() && intersect(*prv, a[id]))
                return make_pair(prv->id, id);
            where[id] = s.insert(nxt, a[id]);
        } else {
            set<seg>::iterator nxt = next(where[id]), prv = prev(where[id]);
            if (nxt != s.end() && prv != s.end() && intersect(*nxt, *prv))
                return make_pair(prv->id, nxt->id);
            s.erase(where[id]);
        }
    }

    return make_pair(-1, -1);
}

```

The main function here is `solve()`, which returns the number of found intersecting segments, or $(-1, -1)$, if there are no intersections.

Checking for the intersection of two segments is carried out by the `intersect()` function, using an **algorithm based on the oriented area of the triangle**.

The queue of segments is the global variable `s`, a `set<event>`. Iterators that specify the position of each segment in the queue (for convenient removal of segments from the queue) are stored in the global array `where`.

Two auxiliary functions `prev()` and `next()` are also introduced, which return iterators to the previous and next elements (or `end()`, if one does not exist).

The constant `EPS` denotes the error of comparing two real numbers (it is

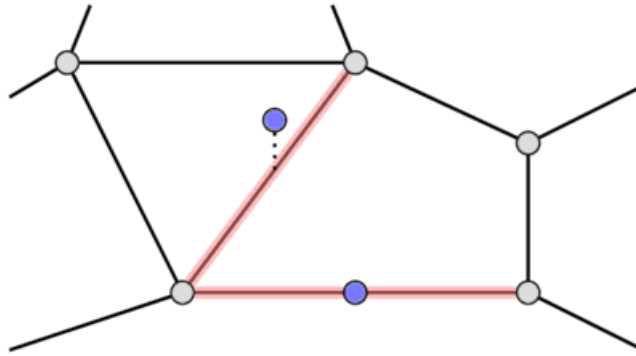
mainly used when checking two segments for intersection).

4.2 Point location in $O(\log n)$

Consider the following problem: you are given a [planar subdivision](#) without no vertices of degree one and zero, and a lot of queries. Each query is a point, for which we should determine the face of the subdivision it belongs to. We will answer each query in $O(\log n)$ offline. This problem may arise when you need to locate some points in a Voronoi diagram or in some simple polygon.

4.2.1 Algorithm

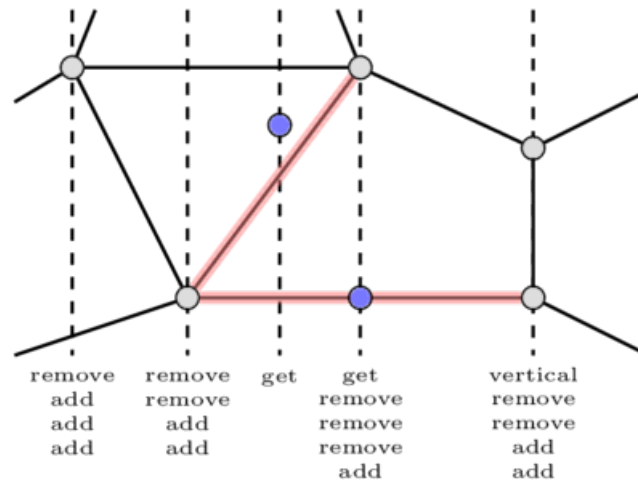
Firstly, for each query point $p(x_0, y_0)$ we want to find such an edge that if the point belongs to any edge, the point lies on the edge we found, otherwise this edge must intersect the line $x = x_0$ at some unique point (x_0, y) where $y < y_0$ and this y is maximum among all such edges. The following image shows both cases.



We will solve this problem offline using the sweep line algorithm. Let's iterate over x-coordinates of query points and edges' endpoints in increasing order and keep a set of edges s . For each x-coordinate we will add some events beforehand.

The events will be of four types: *add*, *remove*, *vertical*, *get*. For each vertical edge (both endpoints have the same x-coordinate) we will add one *vertical* event for the corresponding x-coordinate. For every other edge we will add one *add* event for the minimum of x-coordinates of the endpoints and one *remove* event for the maximum of x-coordinates of the endpoints. Finally, for each query point we will add one *get* event for its x-coordinate.

For each x-coordinate we will sort the events by their types in order (*vertical*, *get*, *remove*, *add*). The following image shows all events in sorted order for each x-coordinate.



We will keep two sets during the sweep-line process. A set t for all non-vertical edges, and one set $vert$ especially for the vertical ones. We will clear the set $vert$ at the beginning of processing each x-coordinate.

Now let's process the events for a fixed x-coordinate.

- If we got a *vertical* event, we will simply insert the minimum y-coordinate of the corresponding edge's endpoints to $vert$.
- If we got a *remove* or *add* event, we will remove the corresponding edge from t or add it to t .
- Finally, for each *get* event we must check if the point lies on some vertical edge by performing a binary search in $vert$. If the point doesn't lie on any vertical edge, we must find the answer for this query in t . To do this, we again make a binary search. In order to handle some degenerate cases (e.g. in case of the triangle $(0, 0), (0, 2), (1, 1)$ when we query the point $(0, 0)$), we must answer all *get* events again after we processed all the events for this x-coordinate and choose the best of two answers.

Now let's choose a comparator for the set t . This comparator should check if one edge doesn't lie above other for every x-coordinate they both cover. Suppose that we have two edges (a, b) and (c, d) . Then the comparator is (in pseudocode):
 $val = \text{sgn}((b - a) \times (c - a)) + \text{sgn}((b - a) \times (d - a))$ if $val \neq 0$ then return val
 $val > 0$ $val = \text{sgn}((d - c) \times (a - c)) + \text{sgn}((d - c) \times (b - c))$ return $val < 0$

Now for every query we have the corresponding edge. How to find the face? If we couldn't find the edge it means that the point is in the outer face. If the point belongs to the edge we found, the face is not unique. Otherwise, there are two candidates - the faces that are bounded by this edge. How to check which one is the answer? Note that the edge is not vertical. Then the answer is the face that is above this edge. Let's find such a face for each non-vertical edge. Consider a counter-clockwise traversal of each face. If during this traversal we

increased x-coordinate while passing through the edge, then this face is the face we need to find for this edge.

4.2.2 Notes

Actually, with persistent trees this approach can be used to answer the queries online.

4.2.3 Implementation

The following code is implemented for integers, but it can be easily modified to work with doubles (by changing the compare methods and the point type). This implementation assumes that the subdivision is correctly stored inside a [DCEL](#) and the outer face is numbered -1 . For each query a pair $(1, i)$ is returned if the point lies strictly inside the face number i , and a pair $(0, i)$ is returned if the point lies on the edge number i .

```
typedef long long ll;

bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& x) { return le(x, 0) ? eq(x, 0) ? 0 : -1 : 1; }

struct pt {
    ll x, y;
    pt() {}
    pt(ll _x, ll _y) : x(_x), y(_y) {}
    pt operator-(const pt& a) const { return pt(x - a.x, y - a.y); }
    ll dot(const pt& a) const { return x * a.x + y * a.y; }
    ll dot(const pt& a, const pt& b) const { return (a - *this).dot(b - *this); }
    ll cross(const pt& a) const { return x * a.y - y * a.x; }
    ll cross(const pt& a, const pt& b) const { return (a - *this).cross(b - *this); }
    bool operator==(const pt& a) const { return a.x == x && a.y == y; }
};

struct Edge {
    pt l, r;
};

bool edge_cmp(Edge* edge1, Edge* edge2)
{
    const pt a = edge1->l, b = edge1->r;
    const pt c = edge2->l, d = edge2->r;
    int val = sgn(a.cross(b, c)) + sgn(a.cross(b, d));
    if (val != 0)
        return val > 0;
    val = sgn(c.cross(d, a)) + sgn(c.cross(d, b));
    return val < 0;
}
```

```

}

enum EventType { DEL = 2, ADD = 3, GET = 1, VERT = 0 };

struct Event {
    EventType type;
    int pos;
    bool operator<(const Event& event) const { return type < event.type; }
};

vector<Edge*> sweepline(vector<Edge*> planar, vector<pt> queries)
{
    using pt_type = decltype(pt::x);

    // collect all x-coordinates
    auto s =
        set<pt_type, std::function<bool(const pt_type&, const pt_type&)>>(lt);
    for (pt p : queries)
        s.insert(p.x);
    for (Edge* e : planar) {
        s.insert(e->l.x);
        s.insert(e->r.x);
    }

    // map all x-coordinates to ids
    int cid = 0;
    auto id =
        map<pt_type, int, std::function<bool(const pt_type&, const pt_type&)>>(
            lt);
    for (auto x : s)
        id[x] = cid++;

    // create events
    auto t = set<Edge*, decltype(*edge_cmp)>(edge_cmp);
    auto vert_cmp = [] (const pair<pt_type, int>& l,
                        const pair<pt_type, int>& r) {
        if (!eq(l.first, r.first))
            return lt(l.first, r.first);
        return l.second < r.second;
    };
    auto vert = set<pair<pt_type, int>, decltype(vert_cmp)>(vert_cmp);
    vector<vector<Event>> events(cid);
    for (int i = 0; i < (int)queries.size(); i++) {
        int x = id[queries[i].x];
        events[x].push_back(Event{GET, i});
    }
    for (int i = 0; i < (int)planar.size(); i++) {
        int lx = id[planar[i]->l.x], rx = id[planar[i]->r.x];
        if (lx > rx) {
            swap(lx, rx);
            swap(planar[i]->l, planar[i]->r);
        }
    }
}

```

```

    if (lx == rx) {
        events[lx].push_back(Event{VERT, i});
    } else {
        events[lx].push_back(Event{ADD, i});
        events[rx].push_back(Event{DEL, i});
    }
}

// perform sweep line algorithm
vector<Edge*> ans(queries.size(), nullptr);
for (int x = 0; x < cid; x++) {
    sort(events[x].begin(), events[x].end());
    vert.clear();
    for (Event event : events[x]) {
        if (event.type == DEL) {
            t.erase(planar[event.pos]);
        }
        if (event.type == VERT) {
            vert.insert(make_pair(
                min(planar[event.pos]->l.y, planar[event.pos]->r.y),
                event.pos));
        }
        if (event.type == ADD) {
            t.insert(planar[event.pos]);
        }
        if (event.type == GET) {
            auto jt = vert.upper_bound(
                make_pair(queries[event.pos].y, planar.size()));
            if (jt != vert.begin()) {
                --jt;
                int i = jt->second;
                if (ge(max(planar[i]->l.y, planar[i]->r.y),
                    queries[event.pos].y)) {
                    ans[event.pos] = planar[i];
                    continue;
                }
            }
            Edge* e = new Edge;
            e->l = e->r = queries[event.pos];
            auto it = t.upper_bound(e);
            if (it != t.begin())
                ans[event.pos] = *(--it);
            delete e;
        }
    }

    for (Event event : events[x]) {
        if (event.type != GET)
            continue;
        if (ans[event.pos] != nullptr &&
            eq(ans[event.pos]->l.x, ans[event.pos]->r.x))
            continue;
    }
}

```

```

        Edge* e = new Edge;
        e->l = e->r = queries[event.pos];
        auto it = t.upper_bound(e);
        delete e;
        if (it == t.begin())
            e = nullptr;
        else
            e = *(--it);
        if (ans[event.pos] == nullptr) {
            ans[event.pos] = e;
            continue;
        }
        if (e == nullptr)
            continue;
        if (e == ans[event.pos])
            continue;
        if (id[ans[event.pos]->r.x] == x) {
            if (id[e->l.x] == x) {
                if (gt(e->l.y, ans[event.pos]->r.y))
                    ans[event.pos] = e;
            }
        } else {
            ans[event.pos] = e;
        }
    }
}
return ans;
}

struct DCEL {
    struct Edge {
        pt origin;
        Edge* nxt = nullptr;
        Edge* twin = nullptr;
        int face;
    };
    vector<Edge*> body;
};

vector<pair<int, int>> point_location(DCEL planar, vector<pt> queries)
{
    vector<pair<int, int>> ans(queries.size());
    vector<Edge*> planar2;
    map<intptr_t, int> pos;
    map<intptr_t, int> added_on;
    int n = planar.body.size();
    for (int i = 0; i < n; i++) {
        if (planar.body[i]->face > planar.body[i]->twin->face)
            continue;
        Edge* e = new Edge;
        e->l = planar.body[i]->origin;

```

```

    e->r = planar.body[i]->twin->origin;
    added_on[(intptr_t)e] = i;
    pos[(intptr_t)e] =
        lt(planar.body[i]->origin.x, planar.body[i]->twin->origin.x)
        ? planar.body[i]->face
        : planar.body[i]->twin->face;
    planar2.push_back(e);
}
auto res = sweepline(planar2, queries);
for (int i = 0; i < (int)queries.size(); i++) {
    if (res[i] == nullptr) {
        ans[i] = make_pair(1, -1);
        continue;
    }
    pt p = queries[i];
    pt l = res[i]->l, r = res[i]->r;
    if (eq(p.cross(l, r), 0) && le(p.dot(l, r), 0)) {
        ans[i] = make_pair(0, added_on[(intptr_t)res[i]]);
        continue;
    }
    ans[i] = make_pair(1, pos[(intptr_t)res[i]]);
}
for (auto e : planar2)
    delete e;
return ans;
}

```

4.2.4 Problems

[TIMUS1848 - Fly Hunt](#)

Chapter 5

Miscellaneous

5.1 Finding the nearest pair of points

5.1.1 Problem statement

Given n points on the plane. Each point p_i is defined by its coordinates (x_i, y_i) . It is required to find among them two such points, such that the distance between them is minimal:

$$\min_{\substack{i,j=0\dots n-1, \\ i \neq j}} \rho(p_i, p_j).$$

We take the usual Euclidean distances:

$$\rho(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

The trivial algorithm - iterating over all pairs and calculating the distance for each — works in $O(n^2)$.

The algorithm running in time $O(n \log n)$ is described below. This algorithm was proposed by Preparata in 1975. Preparata and Shamos also showed that this algorithm is optimal in the decision tree model.

5.1.2 Algorithm

We construct an algorithm according to the general scheme of **divide-and-conquer** algorithms: the algorithm is designed as a recursive function, to which we pass a set of points; this recursive function splits this set in half, calls itself recursively on each half, and then performs some operations to combine the answers. The operation of combining consist of detecting the cases when one point of the optimal solution fell into one half, and the other point into the other (in this case, recursive calls from each of the halves cannot detect this pair separately). The main difficulty, as always in case of divide and conquer algorithms, lies in the effective implementation of the merging stage. If a set of n points is passed to the recursive function, then the merge stage should work no more than $O(n)$, then the asymptotics of the whole algorithm $T(n)$ will be found from the equation:

$$T(n) = 2T(n/2) + O(n).$$

The solution to this equation, as is known, is $T(n) = O(n \log n)$.

So, we proceed on to the construction of the algorithm. In order to come to an effective implementation of the merge stage in the future, we will divide the set of points into two subsets, according to their x -coordinates: In fact, we draw some vertical line dividing the set of points into two subsets of approximately the same size. It is convenient to make such a partition as follows: We sort the points in the standard way as pairs of numbers, i.e.:

$$p_i < p_j \iff (x_i < x_j) \vee ((x_i = x_j) \wedge (y_i < y_j))$$

Then take the middle point after sorting p_m ($m = \lfloor n/2 \rfloor$), and all the points before it and the p_m itself are assigned to the first half, and all the points after it - to the second half:

$$A_1 = \{p_i \mid i = 0 \dots m\}$$

$$A_2 = \{p_i \mid i = m + 1 \dots n - 1\}.$$

Now, calling recursively on each of the sets A_1 and A_2 , we will find the answers h_1 and h_2 for each of the halves. And take the best of them: $h = \min(h_1, h_2)$.

Now we need to make a **merge stage**, i.e. we try to find such pairs of points, for which the distance between which is less than h and one point is lying in A_1 and the other in A_2 . It is obvious that it is sufficient to consider only those points that are separated from the vertical line by a distance less than h , i.e. the set B of the points considered at this stage is equal to:

$$B = \{p_i \mid |x_i - x_m| < h\}.$$

For each point in the set B , we try to find the points that are closer to it than h . For example, it is sufficient to consider only those points whose y -coordinate differs by no more than h . Moreover, it makes no sense to consider those points whose y -coordinate is greater than the y -coordinate of the current point. Thus, for each point p_i we define the set of considered points $C(p_i)$ as follows:

$$C(p_i) = \{p_j \mid p_j \in B, \ y_i - h < y_j \leq y_i\}.$$

If we sort the points of the set B by y -coordinate, it will be very easy to find $C(p_i)$: these are several points in a row ahead to the point p_i .

So, in the new notation, the **merging stage** looks like this: build a set B , sort the points in it by y -coordinate, then for each point $p_i \in B$ consider all points $p_j \in C(p_i)$, and for each pair (p_i, p_j) calculate the distance and compare with the current best distance.

At first glance, this is still a non-optimal algorithm: it seems that the sizes of sets $C(p_i)$ will be of order n , and the required asymptotics will not work. However, surprisingly, it can be proved that the size of each of the sets $C(p_i)$ is

a quantity $O(1)$, i.e. it does not exceed some small constant regardless of the points themselves. Proof of this fact is given in the next section.

Finally, we pay attention to the sorting, which the above algorithm contains: first, sorting by pairs (x, y) , and then second, sorting the elements of the set B by y . In fact, both of these sorts inside the recursive function can be eliminated (otherwise we would not reach the $O(n)$ estimate for the **merging stage**, and the general asymptotics of the algorithm would be $O(n \log^2 n)$). It is easy to get rid of the first sort — it is enough to perform this sort before starting the recursion: after all, the elements themselves do not change inside the recursion, so there is no need to sort again. With the second sorting a little more difficult to perform, performing it previously will not work. But, remembering the merge sort, which also works on the principle of divide-and-conquer, we can simply embed this sort in our recursion. Let recursion, taking some set of points (as we remember, ordered by pairs (x, y)), return the same set, but sorted by the y -coordinate. To do this, simply merge (in $O(n)$) the two results returned by recursive calls. This will result in a set sorted by y -coordinate.

5.1.3 Evaluation of the asymptotics

To show that the above algorithm is actually executed in $O(n \log n)$, we need to prove the following fact: $|C(p_i)| = O(1)$.

So, let us consider some point p_i ; recall that the set $C(p_i)$ is a set of points whose y -coordinate lies in the segment $[y_i - h; y_i]$, and, moreover, along the x coordinate, the point p_i itself, and all the points of the set $C(p_i)$ lie in the band width $2h$. In other words, the points we are considering p_i and $C(p_i)$ lie in a rectangle of size $2h \times h$.

Our task is to estimate the maximum number of points that can lie in this rectangle $2h \times h$; thus, we estimate the maximum size of the set $C(p_i)$. At the same time, when evaluating, we must not forget that there may be repeated points.

Remember that h was obtained from the results of two recursive calls — on sets A_1 and A_2 , and A_1 contains points to the left of the partition line and partially on it, A_2 contains the remaining points of the partition line and points to the right of it. For any pair of points from A_1 , as well as from A_2 , the distance can not be less than h — otherwise it would mean incorrect operation of the recursive function.

To estimate the maximum number of points in the rectangle $2h \times h$ we divide it into two squares $h \times h$, the first square include all points $C(p_i) \cap A_1$, and the second contains all the others, i.e. $C(p_i) \cap A_2$. It follows from the above considerations that in each of these squares the distance between any two points is at least h .

We show that there are at most four points in each square. For example, this can be done as follows: divide the square into 4 sub-squares with sides $h/2$. Then there can be no more than one point in each of these sub-squares (since even the diagonal is equal to $h/\sqrt{2}$, which is less than h). Therefore, there can be no more than 4 points in the whole square.

So, we have proved that in a rectangle $2h \times h$ can not be more than $4 \cdot 2 = 8$ points, and, therefore, the size of the set $C(p_i)$ cannot exceed 7, as required.

5.1.4 Implementation

We introduce a data structure to store a point (its coordinates and a number) and comparison operators required for two types of sorting:

```

struct pt {
    int x, y, id;
};

struct cmp_x {
    bool operator()(const pt & a, const pt & b) const {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
};

struct cmp_y {
    bool operator()(const pt & a, const pt & b) const {
        return a.y < b.y;
    }
};

int n;
vector<pt> a;

```

For a convenient implementation of recursion, we introduce an auxiliary function `upd_ans()`, which will calculate the distance between two points and check whether it is better than the current answer:

```

double mindist;
pair<int, int> best_pair;

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));
    if (dist < mindist) {
        mindist = dist;
        best_pair = {a.id, b.id};
    }
}

```

Finally, the implementation of the recursion itself. It is assumed that before calling it, the array $a[]$ is already sorted by x -coordinate. In recursion we pass just two pointers l, r , which indicate that it should look for the answer for $a[l \dots r]$. If the distance between r and l is too small, the recursion must be stopped, and perform a trivial algorithm to find the nearest pair and then sort the subarray by y -coordinate.

To merge two sets of points received from recursive calls into one (ordered by y -coordinate), we use the standard STL `merge()` function, and create an auxiliary

buffer $t[]$ (one for all recursive calls). (Using `inplace_merge()` is impractical because it generally does not work in linear time.)

Finally, the set B is stored in the same array t .

```
vector<pt> 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.begin() + l, a.begin() + r, cmp_y());
        return;
    }

    int m = (l + r) >> 1;
    int midx = a[m].x;
    rec(l, m);
    rec(m, r);

    merge(a.begin() + l, a.begin() + m, a.begin() + m, a.begin() + r, t.begin(), cmp_y());
    copy(t.begin(), t.begin() + r - l, a.begin() + 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 integer, then at the time of the recursion you can not move to fractional values, and store in *mindist* the square of the minimum distance.

In the main program, recursion should be called as follows:

```
t.resize(n);
sort(a.begin(), a.end(), cmp_x());
mindist = 1E20;
rec(0, n);
```

5.1.5 Generalization: finding a triangle with minimal perimeter

The algorithm described above is interestingly generalized to this problem: among a given set of points, choose three different points so that the sum of pairwise distances between them is the smallest.

In fact, to solve this problem, the algorithm remains the same: we divide the field into two halves of the vertical line, call the solution recursively on both halves, choose the minimum *minper* from the found perimeters, build a strip with the thickness of $\text{minper}/2$, and iterate through all triangles that can improve the answer. (Note that the triangle with perimeter $\leq \text{minper}$ has the longest side $\leq \text{minper}/2$.)

5.1.6 Practice problems

- [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]

5.2 Delaunay triangulation and Voronoi diagram

Consider a set $\{p_i\}$ of points on the plane. A **Voronoi diagram** $V(\{p_i\})$ of $\{p_i\}$ is a partition of the plane into n regions V_i , where $V_i = \{p \in \mathbb{R}^2; \rho(p, p_i) = \min \rho(p, p_k)\}$. The cells of the Voronoi diagram are polygons (possibly infinite). A **Delaunay triangulation** $D(\{p_i\})$ of $\{p_i\}$ is a triangulation where every point p_i is outside or on the boundary of the circumcircle of each triangle $T \in D(\{p_i\})$.

There is a nasty degenerated case when the Voronoi diagram isn't connected and Delaunay triangulation doesn't exist. This case is when all points are collinear.

5.2.1 Properties

The Delaunay triangulation maximizes the minimum angle among all possible triangulations.

The Minimum Euclidean spanning tree of a point set is a subset of edges of its' Delaunay triangulation.

5.2.2 Duality

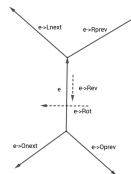
Suppose that $\{p_i\}$ is not collinear and among $\{p_i\}$ no four points lie on one circle. Then $V(\{p_i\})$ and $D(\{p_i\})$ are dual, so if we obtain one of them, we may obtain the other in $O(n)$. What to do if it's not the case? The collinear case may be processed easily. Otherwise, V and D' are dual, where D' is obtained from D by removing all the edges such that two triangles on this edge share the circumcircle.

5.2.3 Building Delaunay and Voronoi

Because of the duality, we only need a fast algorithm to compute only one of V and D . We will describe how to build $D(\{p_i\})$ in $O(n \log n)$. The triangulation will be built via divide-and-conquer algorithm due to Guibas and Stolfi.

5.2.4 Quad-edge data structure

During the algorithm D will be stored inside the quad-edge data structure. This structure is described in the picture:



In the algorithm we will use the following functions on edges:

1. **make_edge(a, b)** This function creates an isolated edge from point **a** to point **b** together with its' reverse edge and both dual edges.

2. `splice(a, b)` This is a key function of the algorithm. It swaps `a->Onext` with `b->Onext` and `a->Onext->Rot->Onext` with `b->Onext->Rot->Onext`.
3. `delete_edge(e)` This function deletes `e` from the triangulation. To delete `e`, we may simply call `splice(e, e->Oprev)` and `splice(e->Rev, e->Rev->Oprev)`.
4. `connect(a, b)` This function creates a new edge `e` from `a->Dest` to `b->Org` in such a way that `a`, `b`, `e` all have the same left face. To do this, we call `e = make_edge(a->Dest, b->Org)`, `splice(e, a->Lnext)` and `splice(e->Rev, b)`.

5.2.5 Algorithm

The algorithm will compute the triangulation and return two quad-edges: the counterclockwise convex hull edge out of the leftmost vertex and the clockwise convex hull edge out of the rightmost vertex.

Let's sort all points by x , and if $x_1 = x_2$ then by y . Let's solve the problem for some segment (l, r) (initially $(l, r) = (0, n-1)$). If $r-l+1 = 2$, we will add an edge $(p[l], p[r])$ and return. If $r-l+1 = 3$, we will firstly add the edges $(p[l], p[l+1])$ and $(p[l+1], p[r])$. We must also connect them using `splice(a->Rev, b)`. Now we must close the triangle. Our next action will depend on the orientation of $p[l], p[l+1], p[r]$. If they are collinear, we can't make a triangle, so we simply return `(a, b->Rev)`. Otherwise, we create a new edge `c` by calling `connect(b, a)`. If the points are oriented counter-clockwise, we return `(a, b->Rev)`. Otherwise we return `(c->Rev, c)`.

Now suppose that $r-l+1 \geq 4$. Firstly, let's solve $L = (l, \frac{l+r}{2})$ and $R = (\frac{l+r}{2} + 1, r)$ recursively. Now we have to merge these triangulations into one triangulation. Note that our points are sorted, so while merging we will add edges from L to R (so-called *cross edges*) and remove some edges from L to L and from R to R . What is the structure of the cross edges? All these edges must cross a line parallel to the y -axis and placed at the splitting x value. This establishes a linear ordering of the cross edges, so we can talk about successive cross edges, the bottom-most cross edge, etc. The algorithm will add the cross edges in ascending order. Note that any two adjacent cross edges will have a common endpoint, and the third side of the triangle they define goes from L to L or from R to R . Let's call the current cross edge the base. The successor of the base will either go from the left endpoint of the base to one of the R -neighbors of the right endpoint or vice versa. Consider the circumcircle of base and the previous cross edge. Suppose this circle is transformed into other circles having base as a chord but lying further into the Oy direction. Our circle will go up for a while, but unless base is an upper tangent of L and R we will encounter a point belonging either to L or to R giving rise to a new triangle without any points in the circumcircle. The new L - R edge of this triangle is the next cross edge added. To do this efficiently, we compute two edges `lcand` and `rcand` so that `lcand` points to the first L point encountered in this process, and `rcand` points to the first R point. Then we choose the one that would be encountered first. Initially base points to the lower tangent of L and R .

5.2.6 Implementation

Note that the implementation of the `in_circle` function is GCC-specific.

```
typedef long long ll;

bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& a) { return a >= 0 ? a : -1; }

struct pt {
    ll x, y;
    pt() { }
    pt(ll _x, ll _y) : x(_x), y(_y) { }
    pt operator-(const pt& p) const {
        return pt(x - p.x, y - p.y);
    }
    ll cross(const pt& p) const {
        return x * p.y - y * p.x;
    }
    ll cross(const pt& a, const pt& b) const {
        return (a - *this).cross(b - *this);
    }
    ll dot(const pt& p) const {
        return x * p.x + y * p.y;
    }
    ll dot(const pt& a, const pt& b) const {
        return (a - *this).dot(b - *this);
    }
    ll sqrLength() const {
        return this->dot(*this);
    }
    bool operator==(const pt& p) const {
        return eq(x, p.x) && eq(y, p.y);
    }
};

const pt inf_pt = pt(1e18, 1e18);

struct QuadEdge {
    pt origin;
    QuadEdge* rot = nullptr;
    QuadEdge* onext = nullptr;
    bool used = false;
    QuadEdge* rev() const {
        return rot->rot;
    }
    QuadEdge* lnext() const {
        return rot->rev()->onext->rot;
    }
}
```

```

    QuadEdge* oprev() const {
        return rot->onext->rot;
    }
    pt dest() const {
        return rev()->origin;
    }
};

QuadEdge* make_edge(pt from, pt to) {
    QuadEdge* e1 = new QuadEdge;
    QuadEdge* e2 = new QuadEdge;
    QuadEdge* e3 = new QuadEdge;
    QuadEdge* e4 = new QuadEdge;
    e1->origin = from;
    e2->origin = to;
    e3->origin = e4->origin = inf_pt;
    e1->rot = e3;
    e2->rot = e4;
    e3->rot = e2;
    e4->rot = e1;
    e1->onext = e1;
    e2->onext = e2;
    e3->onext = e4;
    e4->onext = e3;
    return e1;
}

void splice(QuadEdge* a, QuadEdge* b) {
    swap(a->onext->rot->onext, b->onext->rot->onext);
    swap(a->onext, b->onext);
}

void delete_edge(QuadEdge* e) {
    splice(e, e->oprev());
    splice(e->rev(), e->rev()->oprev());
    delete e->rot;
    delete e->rev()->rot;
    delete e;
    delete e->rev();
}

QuadEdge* connect(QuadEdge* a, QuadEdge* b) {
    QuadEdge* e = make_edge(a->dest(), b->origin);
    splice(e, a->lnext());
    splice(e->rev(), b);
    return e;
}

bool left_of(pt p, QuadEdge* e) {
    return gt(p.cross(e->origin, e->dest()), 0);
}

```



```

bool right_of(pt p, QuadEdge* e) {
    return lt(p.cross(e->origin, e->dest()), 0);
}

template <class T>
T det3(T a1, T a2, T a3, T b1, T b2, T b3, T c1, T c2, T c3) {
    return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) +
        a3 * (b1 * c2 - c1 * b2);
}

bool in_circle(pt a, pt b, pt c, pt d) {
    // If there is __int128, calculate directly.
    // Otherwise, calculate angles.
    #if defined(__LP64__) || defined(_WIN64)
        __int128 det = -det3<__int128>(b.x, b.y, b.sqrLength(), c.x, c.y,
                                         c.sqrLength(), d.x, d.y, d.sqrLength());
        det += det3<__int128>(a.x, a.y, a.sqrLength(), c.x, c.y, c.sqrLength(), d.x,
                               d.y, d.sqrLength());
        det -= det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.sqrLength(), d.x,
                               d.y, d.sqrLength());
        det += det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.sqrLength(), c.x,
                               c.y, c.sqrLength());
        return det > 0;
    #else
        auto ang = [](pt l, pt mid, pt r) {
            ll x = mid.dot(l, r);
            ll y = mid.cross(l, r);
            long double res = atan2((long double)x, (long double)y);
            return res;
        };
        long double kek = ang(a, b, c) + ang(c, d, a) - ang(b, c, d) - ang(d, a, b);
        if (kek > 1e-8)
            return true;
        else
            return false;
    #endif
}

pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<pt>& p) {
    if (r - l + 1 == 2) {
        QuadEdge* res = make_edge(p[l], p[r]);
        return make_pair(res, res->rev());
    }
    if (r - l + 1 == 3) {
        QuadEdge *a = make_edge(p[l], p[l + 1]), *b = make_edge(p[l + 1], p[r]);
        splice(a->rev(), b);
        int sg = sgn(p[l].cross(p[l + 1], p[r]));
        if (sg == 0)
            return make_pair(a, b->rev());
        QuadEdge* c = connect(b, a);
        if (sg == 1)
            return make_pair(a, b->rev());
    }
}

```

```

        else
            return make_pair(c->rev(), c);
    }
    int mid = (l + r) / 2;
    QuadEdge *ldo, *ldi, *rdo, *rdi;
    tie(ldo, ldi) = build_tr(l, mid, p);
    tie(rdi, rdo) = build_tr(mid + 1, r, p);
    while (true) {
        if (left_of(rdi->origin, ldi)) {
            ldi = ldi->lnext();
            continue;
        }
        if (right_of(ldi->origin, rdi)) {
            rdi = rdi->rev()->onext();
            continue;
        }
        break;
    }
    QuadEdge* basel = connect(rdi->rev(), ldi);
    auto valid = [&basel](QuadEdge* e) { return right_of(e->dest(), basel); };
    if (ldi->origin == ldo->origin)
        ldo = basel->rev();
    if (rdi->origin == rdo->origin)
        rdo = basel;
    while (true) {
        QuadEdge* lcand = basel->rev()->onext;
        if (valid(lcand)) {
            while (in_circle(basel->dest(), basel->origin, lcand->dest(),
                             lcand->onext->dest())) {
                QuadEdge* t = lcand->onext;
                delete_edge(lcand);
                lcand = t;
            }
        }
        QuadEdge* rcand = basel->oprev();
        if (valid(rcand)) {
            while (in_circle(basel->dest(), basel->origin, rcand->dest(),
                             rcand->oprev()->dest())) {
                QuadEdge* t = rcand->oprev();
                delete_edge(rcand);
                rcand = t;
            }
        }
        if (!valid(lcand) && !valid(rcand))
            break;
        if (!valid(lcand) ||
            (valid(rcand) && in_circle(lcand->dest(), lcand->origin,
                                      rcand->origin, rcand->dest())))
            basel = connect(rcand, basel->rev());
        else
            basel = connect(basel->rev(), lcand->rev());
    }
}

```

```

    return make_pair(ldo, rdo);
}

vector<tuple<pt, pt, pt>> delaunay(vector<pt> p) {
    sort(p.begin(), p.end(), [](const pt& a, const pt& b) {
        return lt(a.x, b.x) || (eq(a.x, b.x) && lt(a.y, b.y));
    });
    auto res = build_tr(0, (int)p.size() - 1, p);
    QuadEdge* e = res.first;
    vector<QuadEdge*> edges = {e};
    while (lt(e->onext->dest().cross(e->dest(), e->origin), 0))
        e = e->onext;
    auto add = [&p, &e, &edges]() {
        QuadEdge* curr = e;
        do {
            curr->used = true;
            p.push_back(curr->origin);
            edges.push_back(curr->rev());
            curr = curr->lnext();
        } while (curr != e);
    };
    add();
    p.clear();
    int kek = 0;
    while (kek < (int)edges.size()) {
        if (!(e = edges[kek++])->used)
            add();
    }
    vector<tuple<pt, pt, pt>> ans;
    for (int i = 0; i < (int)p.size(); i += 3) {
        ans.push_back(make_tuple(p[i], p[i + 1], p[i + 2]));
    }
    return ans;
}

```

Part IX

Graphs

Chapter 1

Graph traversal

1.1 Breadth-first search

Breadth first search is one of the basic and essential searching algorithms on graphs.

As a result of how the algorithm works, the path found by breadth first search to any node is the shortest path to that node, i.e the path that contains the smallest number of edges in unweighted graphs.

The algorithm works in $O(n + m)$ time, where n is number of vertices and m is the number of edges.

1.1.1 Description of the algorithm

The algorithm takes as input an unweighted graph and the id of the source vertex s . The input graph can be directed or undirected, it does not matter to the algorithm.

The algorithm can be understood as a fire spreading on the graph: at the zeroth step only the source s is on fire. At each step, the fire burning at each vertex spreads to all of its neighbors. In one iteration of the algorithm, the “ring of fire” is expanded in width by one unit (hence the name of the algorithm).

More precisely, the algorithm can be stated as follows: Create a queue q which will contain the vertices to be processed and a Boolean array $used[]$ which indicates for each vertex, if it has been lit (or visited) or not.

Initially, push the source s to the queue and set $used[s] = true$, and for all other vertices v set $used[v] = false$. Then, loop until the queue is empty and in each iteration, pop a vertex from the front of the queue. Iterate through all the edges going out of this vertex and if some of these edges go to vertices that are not already lit, set them on fire and place them in the queue.

As a result, when the queue is empty, the “ring of fire” contains all vertices reachable from the source s , with each vertex reached in the shortest possible way. You can also calculate the lengths of the shortest paths (which just requires maintaining an array of path lengths $d[]$) as well as save information to restore all of these shortest paths (for this, it is necessary to maintain an array of “parents” $p[]$, which stores for each vertex the vertex from which we reached it).

1.1.2 Implementation

We write code for the described algorithm in C++.

```
vector<vector<int>> adj; // adjacency list representation
int n; // number of nodes
int s; // source vertex

queue<int> q;
vector<bool> used(n);
vector<int> d(n), p(n);

q.push(s);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

If we have to restore and display the shortest path from the source to some vertex u , it can be done in the following manner:

```
if (!used[u]) {
    cout << "No path!";
} else {
    vector<int> path;
    for (int v = u; v != -1; v = p[v])
        path.push_back(v);
    reverse(path.begin(), path.end());
    cout << "Path: ";
    for (int v : path)
        cout << v << " ";
}
```

1.1.3 Applications of BFS

- Find the shortest path from a source to other vertices in an unweighted graph.
- Find all connected components in an undirected graph in $O(n + m)$ time: To do this, we just run BFS starting from each vertex, except for vertices which have already been visited from previous runs. Thus, we perform normal BFS from each of the vertices, but do not reset the array *used*[]

each and every time we get a new connected component, and the total running time will still be $O(n + m)$ (performing multiple BFS on the graph without zeroing the array *used*[] is called a series of breadth first searches).

- Finding a solution to a problem or a game with the least number of moves, if each state of the game can be represented by a vertex of the graph, and the transitions from one state to the other are the edges of the graph.
- Finding the shortest path in a graph with weights 0 or 1: This requires just a little modification to normal breadth-first search: if the current edge of zero weight, and distance to the vertex is shorter than the current found distance, then add this vertex not to the back, but to the front of the queue.
- Finding the shortest cycle in a directed unweighted graph: start a breadth-first search from each vertex; as soon as we try to go from the current vertex to an already visited vertex, then it means that we have found the shortest cycle containing the source vertex, and should stop the BFS; from all such cycles (one from each BFS) choose the shortest.
- Find all the edges that lie on any shortest path between a given pair of vertices (a, b) . To do this, run two breadth first searches: one from a and one from b . Let $d_a[]$ be the array containing shortest distances obtained from the first BFS (from a) and $d_b[]$ be the array containing shortest distances obtained from the second BFS from b . Now for every edge (u, v) it is easy to check whether that edge lies on any shortest path between a and b : 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 accomplish that, run two breadth first searches: one from a and one from b . Let $d_a[]$ be the array containing shortest distances obtained from the first BFS (from a) and $d_b[]$ be the array containing shortest distances obtained from the second BFS (from b). Now for each vertex it is easy to check whether it lies on any shortest path between a and b : the criterion is the condition $d_a[v] + d_b[v] = d_a[b]$.
- Find the shortest path of even length from a source vertex s to a target vertex t in an unweighted graph: For this, we must construct an auxiliary graph, whose vertices are the state (v, c) , where v - the current node, $c = 0$ or $c = 1$ - the current parity. Any edge (a, b) of the original graph in this new column will turn into two edges $((u, 0), (v, 1))$ and $((u, 1), (v, 0))$. After that we run a BFS to find the shortest path from the starting vertex $(s, 0)$ to the end vertex $(t, 0)$.

1.1.4 Practice Problems

- [SPOJ: AKBAR](#)
- [SPOJ: NAKANJ](#)
- [SPOJ: WATER](#)
- [SPOJ: MICE AND MAZE](#)

- [Timus: Caravans](#)
- [DevSkills - Halloween Party](#)
- [DevSkills - Ohani And The Link Cut Tree](#)
- [SPOJ - Spiky Mazes](#)
- [SPOJ - Four Chips \(hard\)](#)
- [SPOJ - Inversion Sort](#)
- [Codeforces - Shortest Path](#)
- [SPOJ - Yet Another Multiple Problem](#)
- [UVA 11392 - Binary 3xType Multiple](#)
- [UVA 10968 - KuPellaKeS](#)
- [Codeforces - Police Stations](#)
- [Codeforces - Okabe and City](#)
- [SPOJ - Find the Treasure](#)
- [Codeforces - Bear and Forgotten Tree 2](#)
- [Codeforces - Cycle in Maze](#)
- [UVA - 11312 - Flipping Frustration](#)
- [SPOJ - Ada and Cycle](#)

1.2 Depth First Search

Depth First Search is one of the main graph algorithms.

Depth First Search finds the lexicographical first path in the graph from a source vertex u to each vertex. Depth First Search will also find the shortest paths in a tree (because there only exists one simple path), but on general graphs this is not the case.

The algorithm works in $O(m + n)$ time where n is the number of vertices and m is the number of edges.

1.2.1 Description of the algorithm

The idea behind DFS is to go as deep into the graph as possible, and backtrack once you are at a vertex without any unvisited adjacent vertices.

It is very easy to describe / implement the algorithm recursively: We start the search at one vertex. After visiting a vertex, we further perform a DFS for each adjacent vertex that we haven't visited before. This way we visit all vertices that are reachable from the starting vertex.

For more details check out the implementation.

1.2.2 Applications of Depth First Search

- Find any path in the graph from source vertex u to all vertices.
- Find lexicographical first path in the graph from source u to all vertices.
- Check if a vertex in a tree is an ancestor of some other vertex:

At the beginning and end of each search call we remember the entry and exit “time” of each vertex. Now you can find the answer for any pair of vertices (i, j) in $O(1)$: vertex i is an ancestor of vertex j if and only if $\text{entry}[i] < \text{entry}[j]$ and $\text{exit}[i] > \text{exit}[j]$.

- Find the lowest common ancestor (LCA) of two vertices.
- Topological sorting:
Run a series of depth first searches so as to visit each vertex exactly once in $O(n + m)$ time. The required topological ordering will be the vertices sorted in descending order of exit time.
- Check whether a given graph is acyclic and find cycles in a graph. (As mentioned above by counting back edges in every connected components).
- Find strongly connected components in a directed graph:

First do a topological sorting of the graph. Then transpose the graph and run another series of depth first searches in the order defined by the topological sort. For each DFS call the component created by it is a strongly connected component.

- Find bridges in an undirected graph:

First convert the given graph into a directed graph by running a series of depth first searches and making each edge directed as we go through it, in the direction we went. Second, find the strongly connected components in this directed graph. Bridges are the edges whose ends belong to different strongly connected components.

1.2.3 Classification of edges of a graph

We can classify the edges using the entry and exit time of the end nodes u and v of the edges (u, v) . These classifications are often used for problems like [finding bridges](#) and [finding articulation points](#).

We perform a DFS and classify the encountered edges using the following rules:

If v is not visited:

- Tree Edge - If v is visited after u then edge (u, v) is called a tree edge. In other words, if v is visited for the first time and u is currently being visited then (u, v) is called tree edge. These edges form a DFS tree and hence the name tree edges.

If v is visited before u :

- Back edges - If v is an ancestor of u , then the edge (u, v) is a back edge. v is an ancestor exactly if we already entered v , but not exited it yet. Back edges complete a cycle as there is a path from ancestor v to descendant u (in the recursion of DFS) and an edge from descendant u to ancestor v (back edge), thus a cycle is formed. Cycles can be detected using back edges.
- Forward Edges - If v is a descendant of u , then edge (u, v) is a forward edge. In other words, if we already visited and exited v and $\text{entry}[u] < \text{entry}[v]$ then the edge (u, v) forms a forward edge.
- Cross Edges: if v is neither an ancestor or descendant of u , then edge (u, v) is a cross edge. In other words, if we already visited and exited v and $\text{entry}[u] > \text{entry}[v]$ then (u, v) is a cross edge.

Note: Forward edges and cross edges only exist in directed graphs.

1.2.4 Implementation

```
vector<vector<int>>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<bool> visited;

void dfs(int v) {
    visited[v] = true;
```

```

    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
}

```

This is the most simple implementation of Depth First Search. As described in the applications it might be useful to also compute the entry and exit times and vertex color. We will color all vertices with the color 0, if we haven't visited them, with the color 1 if we visited them, and with the color 2, if we already exited the vertex.

Here is a generic implementation that additionally computes those:

```

vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<int> color;

vector<int> time_in, time_out;
int dfs_timer = 0;

void dfs(int v) {
    time_in[v] = dfs_timer++;
    color[v] = 1;
    for (int u : adj[v])
        if (color[u] == 0)
            dfs(u);
    color[v] = 2;
    time_out[v] = dfs_timer++;
}

```

1.2.5 Practice Problems

- [SPOJ: ABCPATH](#)
- [SPOJ: EAGLE1](#)
- [Codeforces: Kefa and Park](#)
- [Timus: Werewolf](#)
- [Timus: Penguin Avia](#)
- [Timus: Two Teams](#)
- [SPOJ - Ada and Island](#)
- [UVA 657 - The die is cast](#)
- [SPOJ - Sheep](#)
- [SPOJ - Path of the Righteous Man](#)
- [SPOJ - Validate the Maze](#)
- [SPOJ - Ghosts having Fun](#)
- [Codeforces - Underground Lab](#)
- [DevSkills - Maze Tester](#)
- [DevSkills - Tourist](#)
- [Codeforces - Anton and Tree](#)

- [Codeforces - Transformation: From A to B](#)
- [Codeforces - One Way Reform](#)
- [Codeforces - Centroids](#)
- [Codeforces - Generate a String](#)
- [Codeforces - Broken Tree](#)
- [Codeforces - Dasha and Puzzle](#)
- [Codeforces - Making genome In Berland](#)
- [Codeforces - Road Improvement](#)
- [Codeforces - Garland](#)
- [Codeforces - Labeling Cities](#)
- [Codeforces - Send the Fool Futher!](#)
- [Codeforces - The tag Game](#)
- [Codeforces - Leha and Another game about graphs](#)
- [Codeforces - Shortest path problem](#)
- [Codeforces - Upgrading Tree](#)
- [Codeforces - From Y to Y](#)
- [Codeforces - Chemistry in Berland](#)
- [Codeforces - Wizards Tour](#)
- [Codeforces - Ring Road](#)
- [Codeforces - Mail Stamps](#)
- [Codeforces - Ant on the Tree](#)
- [SPOJ - Cactus](#)
- [SPOJ - Mixing Chemicals](#)

Chapter 2

Connected components, bridges, articulations points

2.1 Search for connected components in a graph

Given an undirected graph G with n nodes and m edges. We are required to find in it all the connected components, i.e, several groups of vertices such that within a group each vertex can be reached from another and no path exists between different groups.

2.1.1 An algorithm for solving the problem

- To solve the problem, we can use Depth First Search or Breadth First Search.
- In fact, we will be doing a series of rounds of DFS: The first round will start from first node and all the nodes in the first connected component will be traversed (found). Then we find the first unvisited node of the remaining nodes, and run Depth First Search on it, thus finding a second connected component. And so on, until all the nodes are visited.
- The total asymptotic running time of this algorithm is $O(n + m)$: In fact, this algorithm will not run on the same vertex twice, which means that each edge will be seen exactly two times (at one end and at the other end).

2.1.2 Implementation

```
int n;
vector<int> g[MAXN] ;
bool used[MAXN] ;
vector<int> comp ;

void dfs(int v) {
    used[v] = true ;
    comp.push_back(v);
    for (size_t i = 0; i < (int) 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 most important function that is used is `find_comps()` which finds and displays connected components of the graph.
- The graph is stored in adjacency list representation, i.e `g[i]` contains a list of vertices that have edges from the vertex `i`. The constant `MAXN` should be set equal to the maximum possible number of vertices in the graph.
- Vector `comp` contains a list of nodes in the current connected component.

2.1.3 Practice Problems

- [SPOJ: CCOMPS](#)
- [SPOJ: CT23E](#)
- [CODECHEF: GERALD07](#)

2.2 Finding bridges in a graph in $O(N + M)$

We are given an undirected graph. A bridge is defined as an edge which, when removed, makes the graph disconnected (or more precisely, increases the number of connected components in the graph). The task is to find all bridges in the given graph.

Informally, the problem is formulated as follows: given a map of cities connected with roads, find all “important” roads, i.e. roads which, when removed, cause disappearance of a path between some pair of cities.

The algorithm described here is based on [depth first search](#) and has $O(N + M)$ complexity, where N is the number of vertices and M is the number of edges in the graph.

Note that there also is also the article [Finding Bridges Online](#) - unlike the offline algorithm described here, the online algorithm is able to maintain the list of all bridges in a changing graph (assuming that the only type of change is addition of new edges).

2.2.1 Algorithm

Pick an arbitrary vertex of the graph *root* and run [depth first search](#) from it. Note the following fact (which is easy to prove):

- Let’s say we are in the DFS, looking through the edges starting from vertex v . The current edge (v, to) is a bridge if and only if none of the vertices to and its descendants in the DFS traversal tree has a back-edge to vertex v or any of its ancestors. Indeed, this condition means that there is no other way from v to to except for edge (v, to) .

Now we have to learn to check this fact for each vertex efficiently. We’ll use “time of entry into node” computed by the depth first search.

So, let $tin[v]$ denote entry time for node v . We introduce an array fup which will let us check the fact for each vertex v . $fup[v]$ is the minimum of $tin[v]$, the entry times $tin[p]$ for each node p that is connected to node v via a back-edge (v, p) and the values of $fup[to]$ for each vertex to which is a direct descendant of v in the DFS tree:

$$fup[v] = \min \begin{cases} tin[v] \\ tin[p] & \text{for all } p \text{ for which } (v, p) \text{ is a back edge} \\ fup[to] & \text{for all } to \text{ for which } (v, to) \text{ is a tree edge} \end{cases}$$

Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex v has a child to for which $fup[to] \leq tin[v]$. If $fup[to] = tin[v]$, the back edge comes directly to v , otherwise it comes to one of the ancestors of v .

Thus, the current edge (v, to) in the DFS tree is a bridge if and only if $fup[to] > tin[v]$.

2.2.2 Implementation

The implementation needs to distinguish three cases: when we go down the edge in DFS tree, when we find a back edge to an ancestor of the vertex and when we return to a parent of the vertex. These are the cases:

- $visited[to] = false$ - the edge is part of DFS tree;
- $visited[to] = true \ \&\& \ to \neq parent$ - the edge is back edge to one of the ancestors;
- $to = parent$ - the edge leads back to parent in DFS tree.

To implement this, we need a depth first search function which accepts the parent vertex of the current node.

C++ implementation

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, fup;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = fup[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[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;
    visited.assign(n, false);
    tin.assign(n, -1);
    fup.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

Main function is `find_bridges`; it performs necessary initialization and starts depth first search in each connected component of the graph.

Function `IS_BRIDGE(a, b)` is some function that will process the fact that edge (a, b) is a bridge, for example, print it.

Note that this implementation malfunctions if the graph has multiple edges, since it ignores them. Of course, multiple edges will never be a part of the answer, so `IS_BRIDGE` can check additionally that the reported bridge does is not a multiple edge. Alternatively it's possible to pass to `dfs` the index of the edge used to enter the vertex instead of the parent vertex (and store the indices of all vertices).

2.2.3 Practice Problems

- [UVA #796 “Critical Links”](#) [difficulty: low]
- [UVA #610 “Street Directions”](#) [difficulty: medium]
- [Case of the Computer Network \(Codeforces Round #310 Div. 1 E\)](#) [difficulty: hard]
- [UVA 12363 - Hedge Mazes](#)
- [UVA 315 - Network](#)
- [GYM - Computer Network \(J\)](#)
- [SPOJ - King Graffs Defense](#)
- [SPOJ - Critical Edges](#)
- [Codeforces - Break Up](#)
- [Codeforces - Tourist Reform](#)

2.3 Finding Bridges Online

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). Our task is to find all the bridges in the given graph.

Informally this task can be put as follows: we have to find all the “important” roads on the given road map, i.e. such roads that the removal of any of them will lead to some cities being unreachable from others.

There is already the article [Finding Bridges in \$O\(N + M\)\$](#) which solves this task with a [Depth First Search](#) traversal. This algorithm will be much more complicated, but it has one big advantage: the algorithm described in this article works online, which means that the input graph doesn’t have to be known in advance. The edges are added once at a time, and after each addition the algorithm recounts all the bridges in the current graph. In other words the algorithm is designed to work efficiently on a dynamic, changing graph.

More rigorously the statement of the problem is as follows: Initially the graph is empty and consists of n vertices. Then we receive pairs of vertices (a, b) , which denote an edge added to the graph. After each received edge, i.e. after adding each edge, output the current number of bridges in the graph.

It is also possible to maintain a list of all bridges as well as explicitly support the 2-edge-connected components.

The algorithm described below works in $O(n \log n + m)$ time, where m is the number of edges. The algorithm is based on the data structure [Disjoint Set Union](#). However the implementation in this article takes $O(n \log n + m \log n)$ time, because it uses the simplified version of the DSU without Union by Rank.

2.3.1 Algorithm

First let's define a k -edge-connected component: it is a connected component that remains connected whenever you remove fewer than k edges.

It is very easy to see, that the bridges partition the graph into 2-edge-connected components. If we compress each of those 2-edge-connected components into vertices and only leave the bridges as edges in the compressed graph, then we obtain an acyclic graph, i.e. a forest.

The algorithm described below maintains this forest explicitly as well as the 2-edge-connected components.

It is clear that initially, when the graph is empty, it contains n 2-edge-connected components, which by themselves are not connected.

When adding the next edge (a, b) there can occur three situations:

- Both vertices a and b are in the same 2-edge-connected component - then this edge is not a bridge, and does not change anything in the forest structure, so we can just skip this edge.

Thus, in this case the number of bridges does not change.

- The vertices a and b are in completely different connected components, i.e. each one is part of a different tree. In this case, the edge (a, b) becomes

a new bridge, and these two trees are combined into one (and all the old bridges remain).

Thus, in this case the number of bridges increases by one.

- The vertices a and b are in one connected component, but in different 2-edge-connected components. In this case, this edge forms a cycle along with some of the old bridges. All these bridges end being bridges, and the resulting cycle must be compressed into a new 2-edge-connected component.

Thus, in this case the number of bridges decreases by two or more.

Consequently the whole task is reduced to the effective implementation of all these operations over the forest of 2-edge-connected components.

2.3.2 Data Structures for storing the forest

The only data structure that we need is [Disjoint Set Union](#). In fact we will make two copies of this structure: one will be to maintain the connected components, the other to maintain the 2-edge-connected components. And in addition we store the structure of the trees in the forest of 2-edge-connected components via pointers: Each 2-edge-connected component will store the index `par[]` of its ancestor in the tree.

We will now consistently disassemble every operation that we need to learn to implement:

- Check whether the two vertices lie in the same connected / 2-edge-connected component. It is done with the usual DSU algorithm, we just find and compare the representatives of the DSUs.
- Joining two trees for some edge (a, b) . Since it could turn out that neither the vertex a nor the vertex b are the roots of their trees, the only way to connect these two trees is to re-root one of them. For example you can re-root the tree of vertex a , and then attach it to another tree by setting the ancestor of a to b .

However the question about the effectiveness of the re-rooting operation arises: in order to re-root the tree with the root r to the vertex v , it is necessary to visit all vertices on the path between v and r and redirect the pointers `par[]` in the opposite direction, and also change the references to the ancestors in the DSU that is responsible for the connected components.

Thus, the cost of re-rooting is $O(h)$, where h is the height of the tree. You can make an even worse estimate by saying that the cost is $O(\text{size})$ where `size` is the number of vertices in the tree. The final complexity will not differ.

We now apply a standard technique: we re-root the tree that contains fewer vertices. Then it is intuitively clear that the worst case is when two trees

of approximately equal sizes are combined, but then the result is a tree of twice the size. This does not allow this situation to happen many times.

In general the total cost can be written in the form of a recurrence:

$$T(n) = \max_{k=1 \dots n-1} \{T(k) + T(n-k) + O(\min(k, n-k))\}$$

$T(n)$ is the number of operations necessary to obtain a tree with n vertices by means of re-rooting and unifying trees. A tree of size n can be created by combining two smaller trees of size k and $n-k$. This recurrence has the solution $T(n) = O(n \log n)$.

Thus, the total time spent on all re-rooting operations will be $O(n \log n)$ if we always re-root the smaller of the two trees.

We will have to maintain the size of each connected component, but the data structure DSU makes this possible without difficulty.

- Searching for the cycle formed by adding a new edge (a, b) . Since a and b are already connected in the tree we need to find the [Lowest Common Ancestor](#) of the vertices a and b . The cycle will consist of the paths from b to the LCA, from the LCA to a and the edge a to b .

After finding the cycle we compress all vertices of the detected cycle into one vertex. This means that we already have a complexity proportional to the cycle length, which means that we also can use any LCA algorithm proportional to the length, and don't have to use any fast one.

Since all information about the structure of the tree is available is the ancestor array `par[]`, the only reasonable LCA algorithm is the following: mark the vertices a and b as visited, then we go to their ancestors `par[a]` and `par[b]` and mark them, then advance to their ancestors and so on, until we reach an already marked vertex. This vertex is the LCA that we are looking for, and we can find the vertices on the cycle by traversing the path from a and b to the LCA again.

It is obvious that the complexity of this algorithm is proportional to the length of the desired cycle.

- Compression of the cycle by adding a new edge (a, b) in a tree.

We need to create a new 2-edge-connected component, which will consist of all vertices of the detected cycle (also the detected cycle itself could consist of some 2-edge-connected components, but this does not change anything). In addition it is necessary to compress them in such a way that the structure of the tree is not disturbed, and all pointers `par[]` and two DSUs are still correct.

The easiest way to achieve this is to compress all the vertices of the cycle to their LCA. In fact the LCA is the highest of the vertices, i.e. its ancestor pointer `par[]` remains unchanged. For all the other vertices of the loop the ancestors do not need to be updated, since these vertices simply cease

to exists. But in the DSU of the 2-edge-connected components all these vertices will simply point to the LCA.

We will implement the DSU of the 2-edge-connected components without the Union by rank optimization, therefore we will get the complexity $O(\log n)$ on average per query. To achieve the complexity $O(1)$ on average per query, we need to combine the vertices of the cycle according to Union by rank, and then assign `par[]` accordingly.

2.3.3 Implementation

Here is the final implementation of the whole algorithm.

As mentioned before, for the sake of simplicity the DSU of the 2-edge-connected components is written without Union by rank, therefore the resulting complexity will be $O(\log n)$ on average.

Also in this implementation the bridges themselves are not stored, only their count `bridges`. However it will not be difficult to create a `set` of all bridges.

Initially you call the function `init()`, which initializes the two DSUs (creating a separate set for each vertex, and setting the size equal to one), and sets the ancestors `par`.

The main function is `add_edge(a, b)`, which processes and adds a new edge.

```
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges;
int lca_iteration;
vector<int> last_visit;

void init(int n) {
    par.resize(n);
    dsu_2ecc.resize(n);
    dsu_cc.resize(n);
    dsu_cc_size.resize(n);
    lca_iteration = 0;
    last_visit.assign(n, 0);
    for (int i=0; i<n; ++i) {
        dsu_2ecc[i] = i;
        dsu_cc[i] = i;
        dsu_cc_size[i] = 1;
        par[i] = -1;
    }
    bridges = 0;
}

int find_2ecc(int v) {
    if (v == -1)
        return -1;
    return dsu_2ecc[v] == v ? v : dsu_2ecc[v] = find_2ecc(dsu_2ecc[v]);
}

int find_cc(int v) {
    v = find_2ecc(v);
```

```

    return dsu_cc[v] == v ? v : dsu_cc[v] = find_cc(dsu_cc[v]);
}

void make_root(int v) {
    v = find_2ecc(v);
    int root = v;
    int child = -1;
    while (v != -1) {
        int p = find_2ecc(par[v]);
        par[v] = child;
        dsu_cc[v] = root;
        child = v;
        v = p;
    }
    dsu_cc_size[root] = dsu_cc_size[child];
}

void merge_path (int a, int b) {
    ++lca_iteration;
    vector<int> path_a, path_b;
    int lca = -1;
    while (lca == -1) {
        if (a != -1) {
            a = find_2ecc(a);
            path_a.push_back(a);
            if (last_visit[a] == lca_iteration)
                lca = a;
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            path_b.push_back(b);
            b = find_2ecc(b);
            if (last_visit[b] == lca_iteration)
                lca = b;
            last_visit[b] = lca_iteration;
            b = par[b];
        }
    }

    for (int v : path_a) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
    for (int v : path_b) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
}

```

```

    }
}

void add_edge(int a, int b) {
    a = find_2ecc(a);
    b = find_2ecc(b);
    if (a == b)
        return;

    int ca = find_cc(a);
    int cb = find_cc(b);

    if (ca != cb) {
        ++bridges;
        if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
            swap(a, b);
            swap(ca, cb);
        }
        make_root(a);
        par[a] = dsu_cc[a] = b;
        dsu_cc_size[cb] += dsu_cc_size[a];
    } else {
        merge_path(a, b);
    }
}

```

The DSU for the 2-edge-connected components is stored in the vector `dsu_2ecc`, and the function returning the representative is `find_2ecc(v)`. This function is used many times in the rest of the code, since after the compression of several vertices into one all these vertices cease to exist, and instead only the leader has the correct ancestor `par` in the forest of 2-edge-connected components.

The DSU for the connected components is stored in the vector `dsu_cc`, and there is also an additional vector `dsu_cc_size` to store the component sizes. The function `find_cc(v)` returns the leader of the connectivity component (which is actually the root of the tree).

The re-rooting of a tree `make_root(v)` works as described above: it traverses from the vertex v via the ancestors to the root vertex, each time redirecting the ancestor `par` in the opposite direction. The link to the representative of the connected component `dsu_cc` is also updated, so that it points to the new root vertex. After re-rooting we have to assign the new root the correct size of the connected component. Also we have to be careful that we call `find_2ecc()` to get the representatives of the 2-edge-connected component, rather than some other vertex that have already been compressed.

The cycle finding and compression function `merge_path(a, b)` is also implemented as described above. It searches for the LCA of a and b by rising these nodes in parallel, until we meet a vertex for the second time. For efficiency purposes we choose a unique identifier for each LCA finding call, and mark the traversed vertices with it. This works in $O(1)$, while other approaches like using *set* perform worse. The passed paths are stored in the vectors `path_a`

and `path_b`, and we use them to walk through them a second time up to the LCA, thereby obtaining all vertices of the cycle. All the vertices of the cycle get compressed by attaching them to the LCA, hence the average complexity is $O(\log n)$ (since we don't use Union by rank). All the edges we pass have been bridges, so we subtract 1 for each edge in the cycle.

Finally the query function `add_edge(a, b)` determines the connected components in which the vertices a and b lie. If they lie in different connectivity components, then a smaller tree is re-rooted and then attached to the larger tree. Otherwise if the vertices a and b lie in one tree, but in different 2-edge-connected components, then the function `merge_path(a, b)` is called, which will detect the cycle and compress it into one 2-edge-connected component.

2.4 Finding articulation points in a graph in $O(N+M)$

We are given an undirected graph. An articulation point (or cut vertex) is defined as a vertex which, when removed along with associated edges, makes the graph disconnected (or more precisely, increases the number of connected components in the graph). The task is to find all articulation points in the given graph.

The algorithm described here is based on [depth first search](#) and has $O(N+M)$ complexity, where N is the number of vertices and M is the number of edges in the graph.

2.4.1 Algorithm

Pick an arbitrary vertex of the graph *root* and run [depth first search](#) from it. Note the following fact (which is easy to prove):

- Let's say we are in the DFS, looking through the edges starting from vertex $v \neq \text{root}$. If the current edge (v, to) is such that none of the vertices to or its descendants in the DFS traversal tree has a back-edge to any of ancestors of v , then v is articulation point. Otherwise, v is not an articulation point.
- Let's consider the remaining case of $v = \text{root}$. This vertex will be the point of articulation if and only if this vertex has more than one children in DFS tree.

Now we have to learn to check this fact for each vertex efficiently. We'll use "time of entry into node" computed by the depth first search.

So, let $tin[v]$ denote entry time for node v . We introduce an array $fup[v]$ which will let us check the fact for each vertex v . $fup[v]$ is the minimum of $tin[v]$, the entry times $tin[p]$ for each node p that is connected to node v via a back-edge (v, p) and the values of $fup[to]$ for each vertex to which is a direct descendant of v in the DFS tree:

$$fup[v] = \min \begin{cases} tin[v] \\ tin[p] & \text{for all } p \text{ for which } (v, p) \text{ is a back edge} \\ fup[to] & \text{for all } to \text{ for which } (v, to) \text{ is a tree edge} \end{cases}$$

Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex v has a child to for which $fup[to] < tin[v]$. If $fup[to] = tin[v]$, the back edge comes directly to v , otherwise it comes to one of the ancestors of v .

Thus, the vertex v in the DFS tree is an articulation point if and only if $fup[to] \geq tin[v]$.

2.4.2 Implementation

The implementation needs to distinguish three cases: when we go down the edge in DFS tree, when we find a back edge to an ancestor of the vertex and when we return to a parent of the vertex. These are the cases:

- $visited[to] = false$ - the edge is part of DFS tree;
- $visited[to] = true \ \&\& \ to \neq parent$ - the edge is back edge to one of the ancestors;
- $to = parent$ - the edge leads back to parent in DFS tree.

To implement this, we need a depth first search function which accepts the parent vertex of the current node.

C++ implementation

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, fup;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = fup[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[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);
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    fup.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

Main function is `find_cutpoints`; it performs necessary initialization and starts depth first search in each connected component of the graph.

Function `IS_CUTPOINT(a)` is some function that will process the fact that vertex a is an articulation point, for example, print it (Caution that this can be called multiple times for a vertex).

2.4.3 Practice Problems

- [UVA #10199 “Tourist Guide”](#) [difficulty: low]
- [UVA #315 “Network”](#) [difficulty: low]
- [SPOJ - Submerging Islands](#)

2.5 Finding strongly connected components – Building condensation graph

2.5.1 Definitions

You are given a directed graph G with vertices V and edges E . It is possible that there are loops and multiple edges. Let's denote n as number of vertices and m as number of edges in G .

Strongly connected component is subset of vertices C such that any two vertices of this subset are reachable from each other, i.e. for any $u, v \in C$:

$$u \mapsto v, v \mapsto u$$

where \mapsto means reachability, i.e. existence of the path from first vertex to the second.

It is obvious, that strongly connected components do not intersect each other, i.e. this is a partition of all graph vertices. Thus we can give a definition of condensation graph G^{SCC} as a graph containing every strongly connected component as one vertex. Each vertex of the condensation graph corresponds to the strongly connected component of graph G . There is an oriented edge between two vertices C_i and C_j of the condensation graph if and only if there are two vertices $u \in C_i, v \in C_j$ such that there is an edge in initial graph, i.e. $(u, v) \in E$.

The most important property of the condensation graph is that it is **acyclic**. Indeed, suppose that there is an edge between C and C' , let's prove that there is no edge from C' to C . Suppose that $C' \mapsto C$. Then there are two vertices $u' \in C$ and $v' \in C'$ such that $v' \mapsto u'$. But since u and u' are in the same strongly connected component then there is a path between them; the same for v and v' . As a result, if we join these paths we have that $v \mapsto u$ and at the same time $u \mapsto v$. Therefore u and v should be at the same strongly connected component, so this is contradiction. This completes the proof.

The algorithm described in the next section extracts all strongly connected components in a given graph. It is quite easy to build a condensation graph then.

2.5.2 Description of the algorithm

Described algorithm was independently suggested by Kosaraju and Sharir at 1979. This is an easy-to-implement algorithm based on two series of [depth first search](#), and working for $O(n + m)$ time.

On the first step of the algorithm we are doing sequence of depth first searches, visiting the entire graph. We start at each vertex of the graph and run a depth first search from every non-visited vertex. For each vertex we are keeping track of **exit time** $tout[v]$. These exit times have a key role in an algorithm and this role is expressed in next theorem.

First, let's make notations: let's define exit time $tout[C]$ from the strongly connected component C as maximum of values $tout[v]$ by all $v \in C$. Besides, during the proof of the theorem we will mention entry times $tin[v]$ in each vertex

and in the same way consider $tin[C]$ for each strongly connected component C as minimum of values $tin[v]$ by all $v \in C$.

Theorem. Let C and C' are two different strongly connected components and there is an edge (C, C') in a condensation graph between these two vertices. Then $tout[C] > tout[C']$.

There are two main different cases at the proof depending on which component will be visited by depth first search first, i.e. depending on difference between $tin[C]$ and $tin[C']$: ** The component C was reached first. It means that depth first search comes at some vertex v of component C at some moment, but all other vertices of components C and C' were not visited yet. By condition there is an edge (C, C') in a condensation graph, so not only the entire component C is reachable from v but the whole component C' is reachable as well. It means that depth first search that is running from vertex v will visit all vertices of components C and C' , so they will be descendants for v in a depth first search tree, i.e. for each vertex $u \in C \cup C', u \neq v$ we have that $tout[v] > tout[u]$, as we claimed. ** Assume that component C' was visited first. Similarly, depth first search comes at some vertex v of component C' at some moment, but all other vertices of components C and C' were not visited yet. But by condition there is an edge (C, C') in the condensation graph, so, because of acyclic property of condensation graph, there is no back path from C' to C , i.e. depth first search from vertex v will not reach vertices of C . It means that vertices of C will be visited by depth first search later, so $tout[C] > tout[C']$. This completes the proof.

Proved theorem is **the base of algorithm** for finding strongly connected components. It follows that any edge (C, C') in condensation graph comes from a component with a larger value of $tout$ to component with a smaller value.

If we sort all vertices $v \in V$ by decreasing of their exit moment $tout[v]$ then the first vertex u is going to be a vertex from “root” strongly connected component, i.e. a vertex that no edges in a condensation graph come into. Now we want to run such search from this vertex u so that it will visit all vertices in this strongly connected component, but not others; doing so, we can gradually select all strongly connected components: let’s remove all vertices corresponding to the first selected component, and then let’s find a vertex with the largest value of $tout$, and run this search from it, and so on.

Let’s consider transposed graph G^T , i.e. graph received from G by changing direction of each edge on the opposite. Obviously this graph will have the same strongly connected components, as an initial graph. More over, condensation graph G^{SCC} . It means that there will be no edges from our “root” component to other components.

Thus, for visiting the whole “root” strongly connected component, containing vertex v , is enough to run search from vertex v in graph G^T . This search will visit all vertices of this strongly connected component and only them. As was mentioned before, we can remove these vertices from the graph then, and find the next vertex with a maximal value of $tout[v]$ and run search in transposed graph from it, and so on.

Thus, we built next **algorithm** for selecting strongly connected components:

1st step. Run sequence of depth first search of graph G which will return vertices with increasing exit time $tout$, i.e. some list $order$.

2nd step. Build transposed graph G^T . Run a series of depth (breadth) first searches in the order determined by list $order$ (to be exact in reverse order, i.e. in decreasing order of exit times). Every set of vertices, reached after the next search, will be the next strongly connected component.

Algorithm asymptotic is $O(n + m)$, because it is just two depth (breadth) first searches.

Finally, it is appropriate to mention [topological sort](#) here. First of all, step 1 of the algorithm represents reversed topological sort of graph G (actually this is exactly what vertices' sort by exit time means). Secondly, the algorithm's scheme generates strongly connected components by decreasing order of their exit times, thus it generates components - vertices of condensation graph - in topological sort order.

2.5.3 Implementation

```
vector < vector<int> > g, gr;
vector<bool> used;
vector<int> order, component;

void dfs1 (int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i)
        if (!used[ g[v][i] ])
            dfs1 (g[v][i]);
    order.push_back (v);
}

void dfs2 (int v) {
    used[v] = true;
    component.push_back (v);
    for (size_t i=0; i<gr[v].size(); ++i)
        if (!used[ gr[v][i] ])
            dfs2 (gr[v][i]);
}

int main() {
    int n;
    ... reading n ...
    for (;;) {
        int a, b;
        ... reading next edge (a,b) ...
        g[a].push_back (b);
        gr[b].push_back (a);
    }

    used.assign (n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])
```

```

        dfs1 (i);
    used.assign (n, false);
    for (int i=0; i<n; ++i) {
        int v = order[n-1-i];
        if (!used[v]) {
            dfs2 (v);
            ... printing next component ...
            component.clear();
        }
    }
}

```

Here, g is graph, gr is transposed graph. Function $dfs1$ implements depth first search on graph G , function $dfs2$ - on transposed graph G^T . Function $dfs1$ fills the list $order$ with vertices in increasing order of their exit times (actually, it is making a topological sort). Function $dfs2$ stores all reached vertices in list $component$, that is going to store next strongly connected component after each run.

2.5.4 Literature

- Thomas 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].

2.5.5 Practice Problems

- [SPOJ - Submerging Islands](#)
- [SPOJ - Good Travels](#)
- [SPOJ - Lego](#)
- [Codechef - Chef and Round Run](#)
- [Dev Skills - A Song of Fire and Ice](#)
- [UVA - 11838 - Come and Go](#)
- [UVA 247 - Calling Circles](#)
- [UVA 13057 - Prove Them All](#)
- [UVA 12645 - Water Supply](#)
- [UVA 11770 - Lighting Away](#)
- [UVA 12926 - Trouble in Terrorist Town](#)
- [UVA 11324 - The Largest Clique](#)
- [UVA 11709 - Trust groups](#)
- [UVA 12745 - Wishmaster](#)
- [SPOJ - True Friends](#)
- [SPOJ - Capital City](#)
- [Codeforces - Scheme](#)
- [SPOJ - Ada and Panels](#)

Chapter 3

Single-source shortest paths

3.1 Dijkstra Algorithm

You are given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s . This article discusses finding the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves.

This problem is also called **single-source shortest paths problem**.

3.1.1 Algorithm

Here is an algorithm described by the Dutch computer scientist Edsger W. Dijkstra in 1959.

Let's create an array $d[]$ where for each vertex v we store the current length of the shortest path from s to v in $d[v]$. Initially $d[s] = 0$, and for all other vertices this length equals infinity. In the implementation a sufficiently large number (which is guaranteed to be greater than any possible path length) is chosen as infinity.

$$d[v] = \infty, v \neq s$$

In addition, we maintain a Boolean array $u[]$ which stores for each vertex v whether it's marked. Initially all vertices are unmarked:

$$u[v] = \text{false}$$

The Dijkstra's algorithm runs for n iterations. At each iteration a vertex v is chosen as unmarked vertex which has the least value $d[v]$:

Evidently, in the first iteration the starting vertex s will be selected.

The selected vertex v is marked. Next, from vertex v **relaxations** are performed: all edges of the form (v, to) are considered, and for each vertex to the algorithm tries to improve the value $d[\text{to}]$. If the length of the current edge equals len , the code for relaxation is:

$$d[\text{to}] = \min(d[\text{to}], d[v] + \text{len})$$

After all such edges are considered, the current iteration ends. Finally, after n iterations, all vertices will be marked, and the algorithm terminates. We claim that the found values $d[v]$ are the lengths of shortest paths from s to all vertices v .

Note that if some vertices are unreachable from the starting vertex s , the values $d[v]$ for them will remain infinite. Obviously, the last few iterations of the algorithm will choose those vertices, but no useful work will be done for them. Therefore, the algorithm can be stopped as soon as the selected vertex has infinite distance to it.

Restoring Shortest Paths

Usually one needs to know not only the lengths of shortest paths but also the shortest paths themselves. Let's see how to maintain sufficient information to restore the shortest path from s to any vertex. We'll maintain an array of predecessors $p[]$ in which for each vertex $v \neq s$ $p[v]$ is the penultimate vertex in the shortest path from s to v . Here we use the fact that if we take the shortest path to some vertex v and remove v from this path, we'll get a path ending in at vertex $p[v]$, and this path will be the shortest for the vertex $p[v]$. This array of predecessors can be used to restore the shortest path to any vertex: starting with v , repeatedly take the predecessor of the current vertex until we reach the starting vertex s to get the required shortest path with vertices listed in reverse order. So, the shortest path P to the vertex v is equal to:

$$P = (s, \dots, p[p[p[v]]], p[p[v]], p[v], v)$$

Building this array of predecessors is very simple: for each successful relaxation, i.e. when for some selected vertex v , there is an improvement in the distance to some vertex to , we update the predecessor vertex for to with vertex v :

$$p[to] = v$$

3.1.2 Proof

The main assertion on which Dijkstra's algorithm correctness is based is the following:

After any vertex v becomes marked, the current distance to it $d[v]$ is the shortest, and will no longer change.

The proof is done by induction. For the first iteration this statement is obvious: the only marked vertex is s , and the distance to it is $d[s] = 0$ is indeed the length of the shortest path to s . Now suppose this statement is true for all previous iterations, i.e. for all already marked vertices; let's prove that it is not violated after the current iteration completes. Let v be the vertex selected in the current iteration, i.e. v is the vertex that the algorithm will mark. Now we have to prove that $d[v]$ is indeed equal to the length of the shortest path to it $l[v]$.

Consider the shortest path P to the vertex v . This path can be split into two parts: P_1 which consists of only marked nodes (at least the starting vertex s is

part of P_1), and the rest of the path P_2 (it may include a marked vertex, but it always starts with an unmarked vertex). Let's denote the first vertex of the path P_2 as p , and the last vertex of the path P_1 as q .

First we prove our statement for the vertex p , i.e. let's prove that $d[p] = l[p]$. This is almost obvious: on one of the previous iterations we chose the vertex q and performed relaxation from it. Since (by virtue of the choice of vertex p) the shortest path to p is the shortest path to q plus edge (p, q) , the relaxation from q set the value of $d[p]$ to the length of the shortest path $l[q]$.

Since the edges' weights are non-negative, the length of the shortest path $l[p]$ (which we just proved to be equal to $d[p]$) does not exceed the length $l[v]$ of the shortest path to the vertex v . Given that $l[v] \leq d[v]$ (because Dijkstra's algorithm could not have found a shorter way than the shortest possible one), we get the inequality:

$$d[p] = l[p] \leq l[v] \leq d[v]$$

On the other hand, since both vertices p and v are unmarked, and the current iteration chose vertex v , not p , we get another inequality:

$$d[p] \geq d[v]$$

From these two inequalities we conclude that $d[p] = d[v]$, and then from previously found equations we get:

$$d[v] = l[v]$$

Q.E.D.

3.1.3 Implementation

Dijkstra's algorithm performs n iterations. On each iteration it selects an unmarked vertex v with the lowest value $d[v]$, marks it and checks all the edges (v, to) attempting to improve the value $d[to]$.

The running time of the algorithm consists of:

- n searches for a vertex with the smallest value $d[v]$ among $O(n)$ unmarked vertices
- m relaxation attempts

For the simplest implementation of these operations on each iteration vertex search requires $O(n)$ operations, and each relaxation can be performed in $O(1)$. Hence, the resulting asymptotic behavior of the algorithm is:

$$O(n^2 + m)$$

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
```

```

    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    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 (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}

```

Here the graph `adj` is stored as adjacency list: for each vertex v `adj[v]` contains the list of edges going from this vertex, i.e. the list of `pair<int,int>` where the first element in the pair is the vertex at the other end of the edge, and the second element is the edge weight.

The function takes the starting vertex s and two vectors that will be used as return values.

First of all, the code initializes arrays: distances $d[]$, labels $u[]$ and predecessors $p[]$. Then it performs n iterations. At each iteration the vertex v is selected which has the smallest distance $d[v]$ among all the unmarked vertices. If the distance to selected vertex v is equal to infinity, the algorithm stops. Otherwise the vertex is marked, and all the edges going out from this vertex are checked. If relaxation along the edge is possible (i.e. distance $d[to]$ can be improved), the distance $d[to]$ and predecessor $p[to]$ are updated.

After performing all the iterations array $d[]$ stores the lengths of the shortest paths to all vertices, and array $p[]$ stores the predecessors of all vertices (except starting vertex s). The path to any vertex t can be restored in the following way:

```

vector<int> restore_path(int s, int t, vector<int> const& p) {
    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());
    return path;
}
```

3.1.4 References

- Edsger Dijkstra. A note on two problems in connexion with graphs [1959]
- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. Introduction to Algorithms [2005]

3.1.5 Practice Problems

- [Timus - Ivan's Car](#) [Difficulty:Medium]
- [Timus - Sightseeing Trip](#)
- [SPOJ - SHPATH](#) [Difficulty:Easy]
- [Codeforces - Dijkstra?](#) [Difficulty:Easy]
- [Codeforces - Shortest Path](#)
- [Codeforces - Jzzhu and Cities](#)
- [Codeforces - The Classic Problem](#)
- [Codeforces - President and Roads](#)
- [Codeforces - Complete The Graph](#)
- [TopCoder - SkiResorts](#)
- [TopCoder - MaliciousPath](#)
- [SPOJ - Ada and Trip](#)
- [LA - 3850 - Here We Go\(relians\) Again](#)
- [GYM - Destination Unknown \(D\)](#)
- [UVA 12950 - Even Obsession](#)
- [GYM - Journey to Grece \(A\)](#)
- [UVA 13030 - Brain Fry](#)
- [UVA 1027 - Toll](#)
- [UVA 11377 - Airport Setup](#)
- [Codeforces - Dynamic Shortest Path](#)
- [UVA 11813 - Shopping](#)
- [UVA 11833 - Route Change](#)
- [SPOJ - Easy Dijkstra Problem](#)
- [LA - 2819 - Cave Raider](#)
- [UVA 12144 - Almost Shortest Path](#)
- [UVA 12047 - Highest Paid Toll](#)
- [UVA 11514 - Batman](#)
- [Codeforces - Team Rocket Rises Again](#)
- [UVA - 11338 - Minefield](#)
- [UVA 11374 - Airport Express](#)
- [UVA 11097 - Poor My Problem](#)
- [UVA 13172 - The music teacher](#)

- [Codeforces - Dirty Arkady's Kitchen](#)
- [SPOJ - Delivery Route](#)
- [SPOJ - Costly Chess](#)

3.2 Dijkstra on sparse graphs

For the statement of the problem, the algorithm with implementation and proof can be found on the article [Dijkstra's algorithm](#).

3.2.1 Algorithm

We recall in the derivation of the complexity of Dijkstra's algorithm we used two factors: the time of finding the unmarked vertex with the smallest distance $d[v]$, and the time of the relaxation, i.e. the time of changing the values $d[to]$.

In the simplest implementation these operations require $O(n)$ and $O(1)$ time. Therefore, since we perform the first operation $O(n)$ times, and the second one $O(m)$ times, we obtained the complexity $O(n^2 + m)$.

It is clear, that this complexity is optimal for a dense graph, i.e. when $m \approx n^2$. However in sparse graphs, when m is much smaller than the maximal number of edges n^2 , the complexity gets less optimal because of the first term. Thus it is necessary to improve the execution time of the first operation (and of course without greatly affecting the second operation by much).

To accomplish that we can use a variation of multiple auxiliary data structures. The most efficient is the **Fibonacci heap**, which allows the first operation to run in $O(\log n)$, and the second operation in $O(1)$. Therefore we will get the complexity $O(n \log n + m)$ for Dijkstra's algorithm, which is also the theoretical minimum for the shortest path search problem. Therefore this algorithm works optimal, and Fibonacci heaps are the optimal data structure. There doesn't exist any data structure, that can perform both operations in $O(1)$, because this would also allow to sort a list of random numbers in linear time, which is impossible. Interestingly there exists an algorithm by Thorup that finds the shortest path in $O(m)$ time, however only works for integer weights, and uses a completely different idea. So this doesn't lead to any contradictions. Fibonacci heaps provide the optimal complexity for this task. However they are quite complex to implement, and also have a quite large hidden constant.

As a compromise you can use data structures, that perform both types of operations (extracting a minimum and updating an item) in $O(\log n)$. Then the complexity of Dijkstra's algorithm is $O(n \log m + m \log n) = O(m \log n)$.

C++ provides two such data structures: `set` and `priority_queue`. The first is based on red-black trees, and the second one on heaps. Therefore `priority_queue` has a smaller constant hidden constant, but also has a drawback: it doesn't support the operation of removing an element. Because of this we need to do a "workaround", that actually leads to a slightly worse factor $\log m$ instead of $\log n$ (although in terms of complexity they are identical).

3.2.2 Implementation

`set`

Let us start with the container `set`. Since we need to store vertices ordered by their values $d[]$, it is convenient to store actual pairs: the distance and the

index of the vertex. As a result in a `set` pairs are automatically sorted by their distances.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);

    d[s] = 0;
    set<pair<int, int>> q;
    q.insert({0, s});
    while (!q.empty()) {
        int v = q.begin()->second;
        q.erase(q.begin());

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                q.erase({d[to], to});
                d[to] = d[v] + len;
                p[to] = v;
                q.insert({d[to], to});
            }
        }
    }
}
```

We don't need the array `u[]` from the normal Dijkstra's algorithm implementation any more. We will use the `set` to store that information, and also find the vertex with the shortest distance with it. It kinda acts like a queue. The main loops executes until there are no more vertices in the `set`/queue. A vertex with the smallest distance gets extracted, and for each successful relaxation we first remove the old pair, and then after the relaxation add the new pair into the queue.

priority_queue

The main difference to the implementation with `set` is that we cannot remove elements from the `priority_queue` (although heaps can support that operation in theory). Therefore we have to cheat a little bit. We simply don't delete the old pair from the queue. As a result a vertex can appear multiple times with different distance in the queue at the same time. Among these pairs we are only interested in the pairs where the first element is equal to the corresponding value in `d[]`, all the other other pairs are old. Therefore we need to make a small modification: at the beginning of each iteration, after extracting the next pair,

we check if it is an important pair or if it is already an old and handled pair. This check is important, otherwise the complexity can increase up to $O(nm)$.

By default a `priority_queue` sorts elements in descending order. To make it sort the elements in ascending order, we can either store the negated distances in it, or pass it a different sorting function. We will do the second option.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);

    d[s] = 0;
    using pii = pair<int, int>;
    priority_queue<pii, vector<pii>, greater<pii>> q;
    q.push({0, s});
    while (!q.empty()) {
        int v = q.top().second;
        int d_v = q.top().first;
        q.pop();
        if (d_v != d[v])
            continue;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

In practice the `priority_queue` version is a little bit faster than the version with `set`.

Getting rid of pairs

You can improve the performance a little bit more if you don't store pairs in the containers, but only the vertex indices. In this case we must overload the comparison operator: it must compare two vertices using the distances stored in `d[]`.

As a result of the relaxation, the distance of some vertices will change. However the data structure will not resort itself automatically. In fact changing distances of vertices in the queue, might destroy the data structure. As before, we need to remove the vertex before we relax it, and then insert it again afterwards.

Since we only can remove from `set`, this optimization is only applicable for the `set` method, and doesn't work with `priority_queue` implementation. In practice this significantly increases the performance, especially when larger data types are used to store distances, like `long long` or `double`.

3.3 Bellman-Ford Algorithm

Single source shortest path with negative weight edges

Suppose that we are given a weighted directed graph G with n vertices and m edges, and some specified vertex v . You want to find the length of shortest paths from vertex v to every other vertex.

Unlike the Dijkstra algorithm, this algorithm can also be applied to graphs containing negative weight edges. However, if the graph contains a negative cycle, then, clearly, the shortest path to some vertices may not exist (due to the fact that the weight of the shortest path must be equal to minus infinity); however, this algorithm can be modified to signal the presence of a cycle of negative weight, or even deduce this cycle.

The algorithm bears the name of two American scientists: Richard Bellman and Lester Ford. Ford actually invented this algorithm in 1956 during the study of another mathematical problem, which eventually reduced to a subproblem of finding the shortest paths in the graph, and Ford gave an outline of the algorithm to solve this problem. Bellman in 1958 published an article devoted specifically to the problem of finding the shortest path, and in this article he clearly formulated the algorithm in the form in which it is known to us now.

3.3.1 Description of the algorithm

Let us assume that the graph contains no negative weight cycle. The case of presence of a negative weight cycle will be discussed below in a separate section.

We will create an array of distances $d[0 \dots n - 1]$, which after execution of the algorithm will contain the answer to the problem. In the beginning we fill it as follows: $d[v] = 0$, and all other elements $d[]$ equal to infinity ∞ .

The algorithm consists of several phases. Each phase scans through all edges of the graph, and the algorithm tries to produce relaxation along each edge (a, b) having weight c . Relaxation along the edges is an attempt to improve the value $d[b]$ using value $d[a] + c$. In fact, it means that we are trying to improve the answer for this vertex using edge (a, b) and current response for vertex a .

It is claimed that $n - 1$ phases of the algorithm are sufficient to correctly calculate the lengths of all shortest paths in the graph (again, we believe that the cycles of negative weight do not exist). For unreachable vertices the distance $d[]$ will remain equal to infinity ∞ .

3.3.2 Implementation

Unlike many other graph algorithms, for Bellman-Ford algorithm, it is more convenient to represent the graph using a single list of all edges (instead of n lists of edges - edges from each vertex). We start the implementation with a structure edge for representing the edges. The input to the algorithm are numbers n , m , list e of edges and the starting vertex v . All the vertices are numbered 0 to $n - 1$.

The simplest implementation

The constant INF denotes the number “infinity” — it should be selected in such a way that it is greater than all possible path lengths.

```
struct edge
{
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    for (int i=0; i<n-1; ++i)
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                d[e[j].b] = min (d[e[j].b], d[e[j].a] + e[j].cost);
    // display d, for example, on the screen
}
```

The check `if (d[e[j].a] < INF)` is needed only if the graph contains negative weight edges: no such verification would result in relaxation from the vertices to which paths have not yet found, and incorrect distance, of the type $\infty - 1$, $\infty - 2$ etc. would appear.

A better implementation

This algorithm can be somewhat speeded up: often we already get the answer in a few phases and no useful work is done in remaining phases, just a waste visiting all edges. So, let’s keep the flag, to tell whether something changed in the current phase or not, and if any phase, nothing changed, the algorithm can be stopped. (This optimization does not improve the asymptotic behavior, i.e., some graphs will still need all $n - 1$ phases, but significantly accelerates the behavior of the algorithm “on an average”, i.e., on random graphs.)

With this optimization, it is generally unnecessary to restrict manually the number of phases of the algorithm to $n - 1$ — the algorithm will stop 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;
}
// display d, for example, on the screen
}

```

Retrieving Path

Let us now consider how to modify the algorithm so that it not only finds the length of shortest paths, but also allows to reconstruct the shortest paths.

For that, let's create another array $p[0 \dots n - 1]$, where for each vertex we store its "predecessor", i.e. the penultimate vertex in the shortest path leading to it. In fact, the shortest path to any vertex a is a shortest path to some vertex $p[a]$, to which we added a at the end of the path.

Note that the algorithm works on the same logic: it assumes that the shortest distance to one vertex is already calculated, and, tries to improve the shortest distance to other vertices from that vertex. Therefore, at the time of improvement we just need to remember $p[]$, i.e, the vertex from which this improvement has occurred.

Following is an implementation of the Bellman-Ford with the retrieval of shortest path to a given node t :

```

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    vector<int> p (n - 1);

    for (;;)
    {
        bool any = false;
        for (int j = 0; j < m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    p[e[j].b] = e[j].a;
                    any = true;
                }
            if (!any) break;
    }

    if (d[t] == INF)
        cout << "No path from " << v << " to " << t << ".";
}

```

```

else
{
    vector<int> path;
    for (int cur = t; cur != -1; cur = p[cur])
        path.push_back (cur);
    reverse (path.begin(), path.end());

    cout << "Path from " << v << " to " << t << ": ";
    for (size_t i=0; i<path.size(); ++i)
        cout << path[i] << ' ';
}
}

```

Here starting from the vertex t , we go through the predecessors till we reach starting vertex with no predecessor, and store all the vertices in the path in the list `path`. This list is a shortest path from v to t , but in reverse order, so we call `reverse()` function over `path` and then output the path.

3.3.3 The proof of the algorithm

First, note that for all unreachable vertices u the algorithm will work correctly, the label $d[u]$ will remain equal to infinity (because the algorithm Bellman-Ford will find some way to all reachable vertices from the start vertex v , and relaxation for all other remaining vertices will never happen).

Let us now prove the following assertion: After the execution of i_{th} phase, the Bellman-Ford algorithm correctly finds all shortest paths whose number of edges does not exceed i .

In other words, for any vertex a let us denote the k number of edges in the shortest path to it (if there are several such paths, you can take any). According to this statement, the algorithm guarantees that after k_{th} phase the shortest path for vertex a will be found.

Proof: Consider an arbitrary vertex a to which there is a path from the starting vertex v , and consider a shortest path to it ($p_0 = v, p_1, \dots, p_k = a$). Before the first phase, the shortest path to the vertex $p_0 = v$ was found correctly. During the first phase, the edge (p_0, p_1) has been checked by the algorithm, and therefore, the distance to the vertex p_1 was correctly calculated after the first phase. Repeating this statement k times, we see that after k_{th} phase the distance to the vertex $p_k = a$ gets calculated correctly, which we wanted to prove.

The last thing to notice is that any shortest path cannot have more than $n - 1$ edges. Therefore, the algorithm sufficiently goes up to the $(n - 1)_{th}$ phase. After that no relaxation are guaranteed to improve the distance to some vertex.

3.3.4 The case of a negative cycle

Everywhere above we considered that there is no negative cycle in the graph (precisely, we are interested in a negative cycle that is reachable from the starting vertex v , and, for an unreachable cycles nothing in the above algorithm changes). In the presence of a negative cycle(s), there are further complications associated

with the fact that distances to all vertices in this cycle, as well as the distances to the vertices reachable from this cycle is not defined — they should be equal to minus infinity ($-\infty$).

It is easy to see that the Bellman-Ford algorithm can endlessly do the relaxation among all vertices of this cycle and the vertices reachable from it. Therefore, if you do not limit the number of phases to $n-1$, the algorithm will run indefinitely, constantly improving the distance from these vertices.

Hence we obtain the criterion for presence of a cycle of negative weights reachable for source vertex v : after $(n-1)_{th}$ phase, if we run algorithm for one more phase, and it performs at least one more relaxation, then the graph contains a negative weight cycle that is reachable from v ; otherwise, such a cycle does not exist.

Moreover, if such a cycle is found, the Bellman-Ford algorithm can be modified so that it retrieves this cycle as a sequence of vertices contained in it. For this, it is sufficient to remember the last vertex x for which there was a relaxation in n_{th} phase. This vertex will either lie in a negative weight cycle, or is reachable from it. To get the vertices that are guaranteed to lie in a negative cycle, starting from the vertex x , pass through to the predecessors n times. Hence we will get the vertex y , namely the vertex in the cycle earliest reachable from source. We have to go from this vertex, through the predecessors, until we get back to the same vertex y (and it will happen, because relaxation in a negative weight cycle occur in a circular manner).

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] << ' ';
}
}

```

Due to the presence of a negative cycle, for n iterations of the algorithm, the distances may go far in the negative (apparently, to negative numbers of order -2^n). Hence in the code, we adopted additional measures against the integer overflow as follows:

```
d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);
```

The above implementation looks for a negative cycle reachable from some starting vertex v ; however, the algorithm can be modified to just looking for any negative cycle in the graph. For this we need to put all the distance $d[i]$ to zero and not infinity — as if we are looking for the shortest path from all vertices simultaneously; the validity of the detection of a negative cycle is not affected.

For more on this topic — see separate article, [Finding a negative cycle in the graph](#).

3.3.5 Shortest Path Faster Algorithm (SPFA)

SPFA is a improvement of the Bellman-Ford algorithm which takes advantage of the fact that not all attempts at relaxation will work. The main idea is to create a queue containing only the vertices that were relaxed but that still could not relax their neighbors. And whenever you can relax some neighbor, you should put him in the queue. This algorithm can also be used to detect negative cycles as the Bellman-Ford.

The worst case of this algorithm is equal to the $O(nm)$ of the Bellman-Ford, but in practice it works much faster and some [people claim that it works even in \$O\(m\)\$ on average](#). However be careful, because this algorithm is deterministic and it is easy to create counterexamples that make the algorithm run in $O(nm)$.

There are some care to be taken in the implementation, such as the fact that the algorithm continues forever if there is a negative cycle. To avoid this, it is possible to create a counter that stores how many times a vertex has been relaxed and stop the algorithm as soon as some vertex got relaxed for the n -th time. Note, also there is no reason to put a vertex in the queue if it is already in.

```

const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

bool spfa(int s, vector<int>& d) {
    int n = adj.size();
    d.assign(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;

    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        inqueue[v] = false;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                if (!inqueue[to]) {
                    q.push(to);
                    inqueue[to] = true;
                    cnt[to]++;
                    if (cnt[to] > n)
                        return false; // negative cycle
                }
            }
        }
    }
}

```

3.3.6 Related problems in online judges

A list of tasks that can be solved using the Bellman-Ford algorithm:

- [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]
- [UVA 12519 - The Farnsworth Parabox](#)

See also the problem list in the article [Finding the negative cycle in a graph](#).

3.4 0-1 BFS

It is well-known, that you can find the shortest paths between a single source and all other vertices in $O(|E|)$ using [Breadth First Search](#) in an **unweighted graph**, i.e. the distance is the minimal number of edges that you need to traverse from the source to another vertex. We can interpret such a graph also as a weighted graph, where every edge has the weight 1. If not all edges in graph have the same weight, that we need a more general algorithm, like [Dijkstra](#) which runs in $O(|V|^2 + |E|)$ or $O(|E| \log |V|)$ time.

However if the weights are more constrained, we can often do better. In this article we demonstrate how we can use BFS to solve the SSSP (single-source shortest path) problem in $O(|E|)$, if the weights of each edge is either 0 or 1.

3.4.1 Algorithm

We can develop the algorithm by closely study Dijkstra's algorithm and think about the consequences that our special graph implies. The general form of Dijkstra's algorithm is (here a **set** is used for the priority queue):

```
d.assign(n, INF);
d[s] = 0;
set<pair<int, int>> q;
q.insert({0, s});
while (!q.empty()) {
    int v = q.begin()->second;
    q.erase(q.begin());

    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;

        if (d[v] + w < d[u]) {
            q.erase({d[u], u});
            d[u] = d[v] + w;
            q.insert({d[u], u});
        }
    }
}
```

We can notice that the difference between the distances between the source **s** and two other vertices in the queue differs by at most one. Especially, we know that $d[v] \leq d[u] \leq d[v] + 1$ for each $u \in Q$. The reason for this is, that we only add vertices with equal distance or with distance plus one to the queue each iteration. Assuming there exists a u in the queue with $d[u] - d[v] > 1$, then u must have been insert in the queue via a different vertex t with $d[t] \geq d[u] - 1 > d[v]$. However this is impossible, since Dijkstra's algorithm iterates over the vertices in increasing order.

This means, that the order of the queue looks like this:

$$Q = \underbrace{v}_{d[v]}, \dots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \dots \underbrace{n}_{d[v]+1}$$

This structure is so simple, that we don't need an actually priority queue, i.e. a balanced binary tree, is an overkill. We can simply use a normal queue, and append new vertices at the beginning if the corresponding edge has weight 0, i.e. if $d[u] = d[v]$, or at the end if the edge has weight 1, i.e. if $d[u] = d[v] + 1$. This way the queue still remains sorted at all time.

```
vector<int> d(n, INF);
d[s] = 0;
deque<int> q;
q.push_front(s);
while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;
        if (d[v] + w < d[u]) {
            d[u] = d[v] + w;
            if (w == 1)
                q.push_back(u);
            else
                q.push_front(u);
        }
    }
}
```

3.4.2 Dial's algorithm

We can extend this even further if we allow the weights of the edges to be even bigger. If every edge in the graph has a weight $\leq k$, than the distances of vertices in the queue will differ by at most k from the distance of v to the source. So we can keep $k + 1$ buckets for the vertices in the queue, and whenever the bucket corresponding to the smallest distance gets empty, we make a cyclic shift to get the bucket with the next higher distance. This extension is called **Dial's algorithm**.

3.5 D'Esopo-Pape algorithm

Given a graph with n vertices and m edges with weights w_i and a starting vertex v_0 . The task is to find the shortest path from the vertex v_0 to every other vertex.

The algorithm from D'Esopo-Pape will work faster than [Dijkstra's algorithm](#) and the [Bellman-Ford algorithm](#) in most cases, and will also work for negative edges. However not for negative cycles.

3.5.1 Description

Let the array d contain the shortest path lengths, i.e. d_i is the current length of the shortest path from the vertex v_0 to the vertex i . Initially this array is filled with infinity for every vertex, except $d_{v_0} = 0$. After the algorithm finishes, this array will contain the shortest distances.

Let the array p contain the current ancestors, i.e. p_i is the direct ancestor of the vertex i on the current shortest path from v_0 to i . Just like the array d , the array p changes gradually during the algorithm and at the end takes its final values.

Now to the algorithm. At each step three sets of vertices are maintained:

- M_0 - vertices, for which the distance has already been calculated (although it might not be the final distance)
- M_1 - vertices, for which the distance currently is calculated
- M_2 - vertices, for which the distance has not yet been calculated

The vertices in the set M_1 are stored in a bidirectional queue (deque).

At each step of the algorithm we take a vertex from the set M_1 (from the front of the queue). Let u be the selected vertex. We put this vertex u into the set M_0 . Then we iterate over all edges coming out of this vertex. Let v be the second end of the current edge, and w its weight.

- If v belongs to M_2 , then v is inserted into the set M_1 by inserting it at the back of the queue. d_v is set to $d_u + w$.
- If v belongs to M_1 , then we try to improve the value of d_v : $d_v = \min(d_v, d_u + w)$. Since v is already in M_1 , we don't need to insert it into M_1 and the queue.
- If v belongs to M_1 , and if d_v can be improved $d_v > d_u + w$, then we improve d_v and insert the vertex v back to the set M_1 , placing it at the beginning of the queue.

And of course, with each update in the array d we also have to update the corresponding element in the array p .

3.5.2 Implementation

We will use an array m to store in which set each vertex is currently.

```

struct Edge {
    int to, w;
};

int n;
vector<vector<Edge>> adj;

const int INF = 1e9;

void shortest_paths(int v0, vector<int>& d, vector<int>& p) {
    d.assign(n, INF);
    d[v0] = 0;
    vector<int> m(n, 2);
    deque<int> q;
    q.push_back(v0);
    p.assign(n, -1);

    while (!q.empty()) {
        int u = q.front();
        q.pop_front();
        m[u] = 0;
        for (Edge e : adj[u]) {
            if (d[e.to] > d[u] + e.w) {
                d[e.to] = d[u] + e.w;
                p[e.to] = u;
                if (m[e.to] == 2) {
                    m[e.to] = 1;
                    q.push_back(e.to);
                } else if (m[e.to] == 0) {
                    m[e.to] = 1;
                    q.push_front(e.to);
                }
            }
        }
    }
}

```

3.5.3 Complexity

The algorithm performs usually quite fast. In most cases even faster than Dijkstra's algorithm. However there exist cases for which the algorithm takes exponential time.

Chapter 4

All-pairs shortest paths

4.1 Floyd-Warshall Algorithm

Given an undirected weighted graph G with n vertices. The task is to find the length of the shortest path d_{ij} between each pair of vertices i and j .

The graph may have negative weight edges, but no negative weight cycles (for then the shortest path is undefined).

This algorithm can also be used to detect the presence of negative cycles. The graph has a negative cycle if at the end of the algorithm, the distance from a vertex v to itself is negative.

This algorithm has been simultaneously published in articles by Robert Floyd and Stephen Warshall in 1962. However, in 1959, Bernard Roy published essentially the same algorithm, but its publication went unnoticed.

4.1.1 Description of the algorithm

The key idea of the algorithm is to partition the process of finding the shortest path between any two vertices to several incremental phases.

Let us number the vertices starting from 1 to n . The matrix of distances is $d[] []$.

Before k -th phase ($k = 1 \dots n$), $d[i][j]$ for any vertices i and j stores the length of the shortest path between the vertex i and vertex j , which contains only the vertices $\{1, 2, \dots, k-1\}$ as internal vertices in the path.

In other words, before k -th phase the value of $d[i][j]$ is equal to the length of the shortest path from vertex i to the vertex j , if this path is allowed to enter only the vertex with numbers smaller than k (the beginning and end of the path are not restricted by this property).

It is easy to make sure that this property holds for the first phase. For $k = 0$, we can fill matrix with $d[i][j] = w_{ij}$ if there exists an edge between i and j with weight w_{ij} and $d[i][j] = \infty$ if there doesn't exist an edge. In practice ∞ will be some high value. As we shall see later, this is a requirement for the algorithm.

Suppose now that we are in the k -th phase, and we want to compute the matrix $d[] []$ so that it meets the requirements for the $(k+1)$ -th phase. We have

to fix the distances for some vertices pairs (i, j) . There are two fundamentally different cases:

- The shortest way from the vertex i to the vertex j with internal vertices from the set $\{1, 2, \dots, k\}$ coincides with the shortest path with internal vertices from the set $\{1, 2, \dots, k-1\}$.

In this case, $d[i][j]$ will not change during the transition.

- The shortest path with internal vertices from $\{1, 2, \dots, k\}$ is shorter.

This means that the new, shorter path passes through the vertex k . This means that we can split the shortest path between i and j into two paths: the path between i and k , and the path between k and j . It is clear that both this paths only use internal vertices of $\{1, 2, \dots, k-1\}$ and are the shortest such paths in that respect. Therefore we already have computed the lengths of those paths before, and we can compute the length of the shortest path between i and j as $d[i][k] + d[k][j]$.

Combining these two cases we find that we can recalculate the length of all pairs (i, j) in the k -th phase in the following way:

$$d_{\text{new}}[i][j] = \min(d[i][j], d[i][k] + d[k][j])$$

Thus, all the work that is required in the 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, the value $d[i][j]$ in the distance matrix is the length of the shortest path between i and j , or is ∞ if the path between the vertices i and j does not exist.

A last remark - we don't need to create a separate distance matrix d_{new} for temporarily storing the shortest paths of the k -th phase, i.e. all changes can be made directly in the matrix d at any phase. In fact at any k -th phase we are at most improving the distance of any path in the distance matrix, hence we cannot worsen the length of the shortest path for any pair of the vertices that are to be processed in the $(k+1)$ -th phase or later.

The time complexity of this algorithm is obviously $O(n^3)$.

4.1.2 Implementation

Let d is a 2D array of size $n \times n$, which is filled according to the 0-th phase as explained earlier. Also we will set $d[i][i] = 0$ for any i at the 0-th phase.

Then the algorithm is implemented as follows:

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

It is assumed that if there is no edge between any two vertices i and j , then the matrix at $d[i][j]$ contains a large number (large enough so that it is greater than the length of any path in this graph). Then this edge will always be unprofitable to take, and the algorithm will work correctly.

However if there are negative weight edges in the graph, special measures have to be taken. Otherwise the resulting values in matrix may be of the form $\infty - 1$, $\infty - 2$, etc., which, of course, still indicates that between the respective vertices doesn't exist a path. Therefore, if the graph has negative weight edges, it is better to write the Floyd-Warshall algorithm in the following way, so that it does not perform transitions using paths that don't exist.

```
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]);
        }
    }
}
```

4.1.3 Retrieving the sequence of vertices in the shortest path

It is easy to maintain additional information with which it will be possible to retrieve the shortest path between any two given vertices in the form of a sequence of vertices.

For this, in addition to the distance matrix $d[][]$, a matrix of ancestors $p[][]$ must be maintained, which will contain the number of the phase where the shortest distance between two vertices was last modified. It is clear that the number of the phase is nothing more than a vertex in the middle of the desired shortest path. Now we just need to find the shortest path between vertices i and $p[i][j]$, and between $p[i][j]$ and j . This leads to a simple recursive reconstruction algorithm of the shortest path.

4.1.4 The case of real weights

If the weights of the edges are not integer but real, it is necessary to take the errors, which occur when working with float types, into account.

The Floyd-Warshall algorithm has the unpleasant effect, that the errors accumulate very quickly. In fact if there is an error in the first phase of δ , this error may propagate to the second iteration as 2δ , to the third iteration as 4δ , and so on.

To avoid this the algorithm can be modified to take the error ($\text{EPS} = \delta$) into account by using following comparison:

```
if (d[i][k] + d[k][j] < d[i][j] - EPS)
    d[i][j] = d[i][k] + d[k][j];
```

4.1.5 The case of negative cycles

Formally the Floyd-Warshall algorithm does not apply to graphs containing negative weight cycle(s). But for all pairs of vertices i and j for which there doesn't exist a path starting at i , visiting a negative cycle, and end at j , the algorithm will still work correctly.

For the pair of vertices for which the answer does not exist (due to the presence of a negative cycle in the path between them), the Floyd algorithm will store any number (perhaps highly negative, but not necessarily) in the distance matrix. However it is possible to improve the Floyd-Warshall algorithm, so that it carefully treats such pairs of vertices, and outputs them, for example as $-\text{INF}$.

This can be done in the following way: let us run the usual Floyd-Warshall algorithm for a given graph. Then a shortest path between vertices i and j does not exist, if and only if, there is a vertex t that is reachable from i and also from j , for which $d[t][t] < 0$.

In addition, when using the Floyd-Warshall algorithm for graphs with negative cycles, we should keep in mind that situations may arise in which distances can get exponentially fast into the negative. Therefore integer overflow must be handled by limiting the minimal distance by some value (e.g. $-\text{INF}$).

To learn more about finding negative cycles in a graph, see the separate article [Finding a negative cycle in the graph](#).

4.1.6 Practice Problems

- [UVA: Page Hopping](#)
- [SPOJ: Possible Friends](#)
- [CODEFORCES: Greg and Graph](#)
- [SPOJ: CHICAGO - 106 miles to Chicago](#)
- [UVA 10724 - Road Construction](#)
- [UVA 117 - The Postal Worker Rings Once](#)
- [Codeforces - Traveling Graph](#)
- [UVA - 1198 - The Geodetic Set Problem](#)
- [UVA - 10048 - Audiophobia](#)
- [UVA - 125 - Numbering Paths](#)
- [LOJ - Travel Company](#)
- [UVA 423 - MPI Maelstrom](#)
- [UVA 1416 - Warfare And Logistics](#)
- [UVA 1233 - USHER](#)
- [UVA 10793 - The Orc Attack](#)
- [UVA 10099 The Tourist Guide](#)
- [UVA 869 - Airline Comparison](#)
- [UVA 13211 - Geonosis](#)
- [SPOJ - Defend the Rohan](#)
- [Codeforces - Roads in Berland](#)
- [Codeforces - String Problem](#)
- [GYM - Manic Moving \(C\)](#)
- [SPOJ - Arbitrage](#)

- [UVA - 12179 - Randomly-priced Tickets](#)
- [LOJ - 1086 - Jogging Trails](#)
- [SPOJ - Ingredients](#)

4.2 Number of paths of fixed length / Shortest paths of fixed length

The following article describes solutions to these two problems built on the same idea: reduce the problem to the construction of matrix and compute the solution with the usual matrix multiplication or with a modified multiplication.

4.2.1 Number of paths of a fixed length

We are given a directed, unweighted graph G with n vertices and we are given an integer k . The task is the following: for each pair of vertices (i, j) we have to find the number of paths of length k between these vertices. Paths don't have to be simple, i.e. vertices and edges can be visited any number of times in a single path.

We assume that the graph is specified with an adjacency matrix, i.e. the matrix $G[\][\]$ of size $n \times n$, where each element $G[i][j]$ equal to 1 if the vertex i is connected with j by an edge, and 0 if they are not connected by an edge. The following algorithm works also in the case of multiple edges: if some pair of vertices (i, j) is connected with m edges, then we can record this in the adjacency matrix by setting $G[i][j] = m$. Also the algorithm works if the graph contains loops (a loop is an edge that connect a vertex with itself).

It is obvious that the constructed adjacency matrix is the answer to the problem for the case $k = 1$. It contains the number of paths of length 1 between each pair of vertices.

We will build the solution iteratively: Let's assume we know the answer for some k . Here we describe a method how we can construct the answer for $k + 1$. Denote by C_k the matrix for the case k , and by C_{k+1} the matrix we want to construct. With the following formula we can compute every entry of C_{k+1} :

$$C_{k+1}[i][j] = \sum_{p=1}^n C_k[i][p] \cdot G[p][j]$$

It is easy to see that the formula computes nothing other than the product of the matrices C_k and G :

$$C_{k+1} = C_k \cdot G$$

Thus the solution of the problem can be represented as follows:

$$C_k = \underbrace{G \cdot G \cdots G}_{k \text{ times}} = G^k$$

It remains to note that the matrix products can be raised to a high power efficiently using [Binary exponentiation](#). This gives a solution with $O(n^3 \log k)$ complexity.

4.2.2 Shortest paths of a fixed length

We are given a directed weighted graph G with n vertices and an integer k . For each pair of vertices (i, j) we have to find the length of the shortest path between i and j that consists of exactly k edges.

We assume that the graph is specified by an adjacency matrix, i.e. via the matrix $G[\cdot][\cdot]$ of size $n \times n$ where each element $G[i][j]$ contains the length of the edges from the vertex i to the vertex j . If there is no edge between two vertices, then the corresponding element of the matrix will be assigned to infinity ∞ .

It is obvious that in this form the adjacency matrix is the answer to the problem for $k = 1$. It contains the lengths of shortest paths between each pair of vertices, or ∞ if a path consisting of one edge doesn't exist.

Again we can build the solution to the problem iteratively: Let's assume we know the answer for some k . We show how we can compute the answer for $k + 1$. Let us denote L_k the matrix for k and L_{k+1} the matrix we want to build. Then the following formula computes each entry of L_{k+1} :

$$L_{k+1}[i][j] = \min_{p=1 \dots n} (L_k[i][p] + G[p][j])$$

When looking closer at this formula, we can draw an analogy with the matrix multiplication: in fact the matrix L_k is multiplied by the matrix G , the only difference is that instead in the multiplication operation we take the minimum instead of the sum.

$$L_{k+1} = L_k \odot G,$$

where the operation \odot is defined as follows:

$$A \odot B = C \iff C_{ij} = \min_{p=1 \dots n} (A_{ip} + B_{pj})$$

Thus the solution of the task can be represented using the modified multiplication:

$$L_k = \underbrace{G \odot \dots \odot G}_{k \text{ times}} = G^{\odot k}$$

It remains to note that we also can compute this exponentiation efficiently with [Binary exponentiation](#), because the modified multiplication is obviously associative. So also this solution has $O(n^3 \log k)$ complexity.

4.2.3 Generalization of the problems for paths with length up to k

The above solutions solve the problems for a fixed k . However the solutions can be adapted for solving problems for which the paths are allowed to contain no more than k edges.

This can be done by slightly modifying the input graph.

We duplicate each vertex: for each vertex v we create one more vertex v' and add the edge (v, v') and the loop (v', v') . The number of paths between i and j with at most k edges is the same number as the number of paths

between i and j' with exactly $k + 1$ edges, since there is a bijection that maps every path $[p_0 = i, p_1, \dots, p_{m-1}, p_m = j]$ of length $m \leq k$ to the path $[p_0 = i, p_1, \dots, p_{m-1}, p_m = j, j', \dots, j']$ of length $k + 1$.

The same trick can be applied to compute the shortest paths with at most k edges. We again duplicate each vertex and add the two mentioned edges with weight 0.

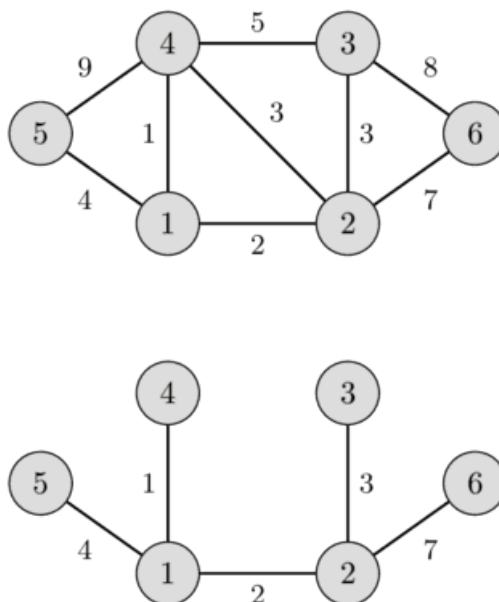
Chapter 5

Spanning trees

5.1 Minimum spanning tree - Prim's algorithm

Given a weighted, undirected graph G with n vertices and m edges. You want to find a spanning tree of this graph which connects all vertices and has the least weight (i.e. the sum of weights of edges is minimal). A spanning tree is a set of edges such that any vertex can reach any other by exactly one simple path. The spanning tree with the least weight is called a minimum spanning tree.

In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.



It is easy to see that any spanning tree will necessarily contain $n - 1$ edges.

This problem appears quite naturally in a lot of problems. For instance in the following problem: there are n cities and for each pair of cities we are given

the cost to build a road between them (or we know that is physically impossible to build a road between them). We have to build roads, such that we can get from each city to every other city, and the cost for building all roads is minimal.

5.1.1 Prim's Algorithm

This algorithm was originally discovered by the Czech mathematician Vojtěch Jarník in 1930. However this algorithm is mostly known as Prim's algorithm after the American mathematician Robert Clay Prim, who rediscovered and republished it in 1957. Additionally Edsger Dijkstra published this algorithm in 1959.

Algorithm description

Here we describe the algorithm in its simplest form. The minimum spanning tree is built gradually by adding edges one at a time. At first the spanning tree consists only of a single vertex (chosen arbitrarily). Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices. Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have $n - 1$ edges).

In the end the constructed spanning tree will be minimal. If the graph was originally not connected, then there doesn't exist a spanning tree, so the number of selected edges will be less than $n - 1$.

Proof

Let the graph G be connected, i.e. the answer exists. We denote by T the resulting graph found by Prim's algorithm, and by S the minimum spanning tree. Obviously T is indeed a spanning tree and a subgraph of G . We only need to show that the weights of S and T coincide.

Consider the first time in the algorithm when we add an edge to T that is not part of S . Let us denote this edge with e , its ends by a and b , and the set of already selected vertices as V ($a \in V$ and $b \notin V$, or visa versa).

In the minimal spanning tree S the vertices a and b are connected by some path P . On this path we can find an edge f such that one end of f lies in V and the other end doesn't. Since the algorithm chose e instead of f , it means that the weight of f is greater or equal to the weight of e .

We add the edge e to the minimum spanning tree S and remove the edge f . By adding e we created a cycle, and since f was also part of the only cycle, by removing it the resulting graph is again free of cycles. And because we only removed an edge from a cycle, the resulting graph is still connected.

The resulting spanning tree cannot have a larger total weight, since the weight of e was not larger than the weight of f , and it also cannot have a smaller weight since S was a minimum spanning tree. This means that by replacing the edge f with e we generated a different minimum spanning tree. And e has to have the same weight as f .

Thus all the edges we pick in Prim's algorithm have the same weights as the edges of any minimum spanning tree, which means that Prim's algorithm really generates a minimum spanning tree.

5.1.2 Implementation

The complexity of the algorithm depends on how we search for the next minimal edge among the appropriate edges. There are multiple approaches leading to different complexities and different implementations.

Trivial implementations: $O(nm)$ and $O(n^2 + m \log n)$

If we search the edge by iterating over all possible edges, then it takes $O(m)$ time to find the edge with the minimal weight. The total complexity will be $O(nm)$. In the worst case this is $O(n^3)$, really slow.

This algorithm can be improved if we only look at one edge from each already selected vertex. For example we can sort the edges from each vertex in ascending order of their weights, and store a pointer to the first valid edge (i.e. an edge that goes to an non-selected vertex). Then after finding and selecting the minimal edge, we update the pointers. This gives a complexity of $O(n^2 + m)$, and for sorting the edges an additional $O(m \log n)$, which gives the complexity $O(n^2 \log n)$ in the worst case.

Below we consider two slightly different algorithms, one for dense and one for sparse graphs, both with a better complexity.

Dense graphs: $O(n^2)$

We approach this problem from a different side: for every not yet selected vertex we will store the minimum edge to an already selected vertex.

Then during a step we only have to look at these minimum weight edges, which will have a complexity of $O(n)$.

After adding an edge some minimum edge pointers have to be recalculated. Note that the weights only can decrease, i.e. the minimal weight edge of every not yet selected vertex might stay the same, or it will be updated by an edge to the newly selected vertex. Therefore this phase can also be done in $O(n)$.

Thus we received a version of Prim's algorithm with the complexity $O(n^2)$.

In particular this implementation is very convenient for the Euclidean Minimum Spanning Tree problem: we have n points on a plane and the distance between each pair of points is the Euclidean distance between them, and we want to find a minimum spanning tree for this complete graph. This task can be solved by the described algorithm in $O(n^2)$ time and $O(n)$ memory, which is not possible with [Kruskal's algorithm](#).

```

int n;
vector<vector<int>> adj; // adjacency matrix of graph
const int INF = 1000000000; // weight INF means there is no edge

struct Edge {
    int w = INF, to = -1;
};

void prim() {
    int total_weight = 0;
    vector<bool> selected(n);
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }

    cout << total_weight << endl;
}

```

The adjacency matrix `adj[][]` of size $n \times n$ stores the weights of the edges, and it uses the weight `INF` if there doesn't exist an edge between two vertices. The algorithm uses two arrays: the flag `selected[]`, which indicates which vertices we already have selected, and the array `min_e[]` which stores the edge with minimal weight to an selected vertex for each not-yet-selected vertex (it stores the weight and the end vertex). The algorithm does n steps, in each iteration the vertex with the smallest edge weight is selected, and the `min_e[]` of all other vertices gets updated.

Sparse graphs: $O(m \log n)$

In the above described algorithm it is possible to interpret the operations of finding the minimum and modifying some values as set operations. These two classical operations are supported by many data structure, for example by `set` in C++ (which are implemented via red-black trees).

The main algorithm remains the same, but now we can find the minimum edge in $O(\log n)$ time. On the other hand recomputing the pointers will now take $O(n \log n)$ time, which is worse than in the previous algorithm.

But when we consider that we only need to update $O(m)$ times in total, and perform $O(n)$ searches for the minimal edge, then the total complexity will be $O(m \log n)$. For sparse graphs this is better than the above algorithm, but for dense graphs this will be slower.

```
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return w < other.w;
    }
};

int n;
vector<vector<Edge>> adj;

void prim() {
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
    for (int i = 0; i < n; ++i) {
        if (q.empty()) {
            cout << "No MST!" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v]) {
            if (e.w < min_e[e.to].w) {
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }
}
```

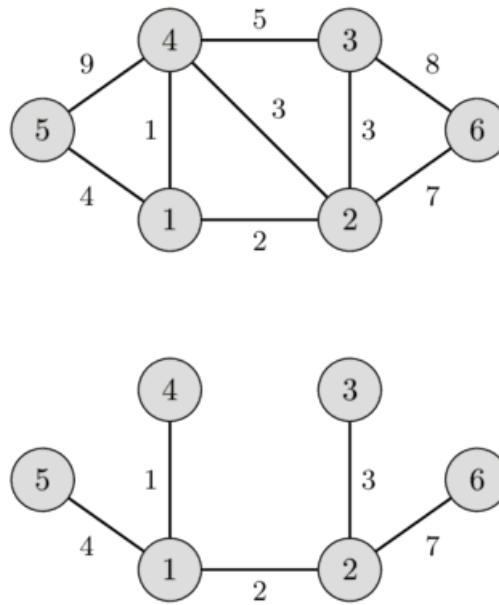
```
        }  
    }  
  
    cout << total_weight << endl;  
}
```

Here the graph is represented via a adjacency list `adj[]`, where `adj[v]` contains all edges (in form of weight and target pairs) for the vertex `v`. `min_e[v]` will store the weight of the smallest edge from vertex `v` to an already selected vertex (again in the form of a weight and target pair). In addition the queue `q` is filled with all not yet selected vertices in the order of increasing weights `min_e`. The algorithm does `n` steps, on each of which it selects the vertex `v` with the smallest weight `min_e` (by extracting it from the beginning of the queue), and then looks through all the edges from this vertex and updates the values in `min_e` (during an update we also need to also remove the old edge from the queue `q` and put in the new edge).

5.2 Minimum spanning tree - Kruskal's algorithm

Given a weighted undirected graph. We want to find a subtree of this graph which connects all vertices (i.e. it is a spanning tree) and has the least weight (i.e. the sum of weights of all the edges is minimum) of all possible spanning trees. This spanning tree is called a minimum spanning tree.

In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.



This article will discuss few important facts associated with minimum spanning trees, and then will give the simplest implementation of Kruskal's algorithm for finding minimum spanning tree.

5.2.1 Properties of the minimum spanning tree

- A minimum spanning tree of a graph is unique, if the weight of all the edges are distinct. Otherwise, there may be multiple minimum spanning trees. (Specific algorithms typically output one of the possible minimum spanning trees).
- Minimum spanning tree is also the tree with minimum product of weights of edges. (It can be easily proved by replacing the weights of all edges with their logarithms)
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph. (This follows from the validity of Kruskal's algorithm).
- The maximum spanning tree (spanning tree with the sum of weights of edges being maximum) of a graph can be obtained similarly to that of the

minimum spanning tree, by changing the signs of the weights of all the edges to their opposite and then applying any of the minimum spanning tree algorithm.

5.2.2 Kruskal's algorithm

This algorithm was described by Joseph Bernard Kruskal, Jr. in 1956.

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph. Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer. After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.

5.2.3 The simplest implementation

The following code directly implements the algorithm described above, and is having $O(M \log M + N^2)$ time complexity. Sorting edges requires $O(M \log N)$ (which is the same as $O(M \log M)$) operations. Information regarding the subtree to which a vertex belongs is maintained with the help of an array `tree_id[]` - for each vertex `v`, `tree_id[v]` stores the number of the tree, to which `v` belongs. For each edge, whether it belongs to the ends of different trees, can be determined in $O(1)$. Finally, the union of the two trees is carried out in $O(N)$ by a simple pass through `tree_id[]` array. Given that the total number of merge operations is $N - 1$, we obtain the asymptotic behavior of $O(M \log N + N^2)$.

```
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<int> tree_id(n);
vector<Edge> result;
for (int i = 0; i < n; i++)
    tree_id[i] = i;

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (tree_id[e.u] != tree_id[e.v]) {
```

```

    cost += e.weight;
    result.push_back(e);

    int old_id = tree_id[e.u], new_id = tree_id[e.v];
    for (int i = 0; i < n; i++) {
        if (tree_id[i] == old_id)
            tree_id[i] = new_id;
    }
}
}

```

5.2.4 Proof of correctness

Why does Kruskal's algorithm give us the correct result?

If the original graph was connected, then also the resulting graph will be connected. Because otherwise there would be two components that could be connected with at least one edge. Though this is impossible, because Kruskal would have chosen one of these edges, since the ids of the components are different. Also the resulting graph doesn't contain any cycles, since we forbid this explicitly in the algorithm. Therefore the algorithm generates a spanning tree.

So why does this algorithm give us a minimum spanning tree?

We can show the proposal "if F is a set of edges chosen by the algorithm at any stage in the algorithm, then there exists a MST that contains all edges of F " using induction.

The proposal is obviously true at the beginning, the empty set is a subset of any MST.

Now let's assume F is some edge set at any stage of the algorithm, T is a MST containing F and e is the new edge we want to add using Kruskal.

If e generates a cycle, then we don't add it, and so the proposal is still true after this step.

In case that T already contains e , the proposal is also true after this step.

In case T doesn't contain the edge e , then $T + e$ will contain a cycle C . This cycle will contain at least one edge f , that is not in F . The set of edges $T - f + e$ will also be a spanning tree. Notice that the weight of f cannot be smaller than the weight of e , because otherwise Kruskal would have chosen f earlier. It also cannot have a bigger weight, since that would make the total weight of $T - f + e$ smaller than the total weight of T , which is impossible since T is already a MST. This means that the weight of e has to be the same as the weight of f . Therefore $T - f + e$ is also a MST, and it contains all edges from $F + e$. So also here the proposal is still fulfilled after the step.

This proves the proposal. Which means that after iterating over all edges the resulting edge set will be connected, and will be contained in a MST, which means that it has to be a MST already.

5.2.5 Improved implementation

We can use the **`**Disjoint Set Union**`** (DSU) data structure to write a faster implementation of the Kruskal's algorithm with the time complexity of about

$O(M \log N)$. [This article](#) details such an approach.

5.2.6 Practice Problems

- [SPOJ - Koicost](#)
- [SPOJ - MaryBMW](#)
- [Codechef - Fullmetal Alchemist](#)
- [Codeforces - Edges in MST](#)
- [UVA 12176 - Bring Your Own Horse](#)
- [UVA 10600 - ACM Contest and Blackout](#)
- [UVA 10724 - Road Construction](#)
- [Hackerrank - Roads in HackerLand](#)
- [UVA 11710 - Expensive subway](#)
- [Codechef - Chefland and Electricity](#)
- [UVA 10307 - Killing Aliens in Borg Maze](#)
- [Codeforces - Flea](#)
- [Codeforces - Igon in Museum](#)
- [Codeforces - Hongcow Builds a Nation](#)
- [DevSkills - Repairing Road](#)
- [UVA - 908 - Re-connecting Computer Sites](#)
- [UVA 1208 - Oreon](#)
- [UVA 1235 - Anti Brute Force Lock](#)
- [UVA 10034 - Freckles](#)
- [UVA 11228 - Transportation system](#)
- [UVA 11631 - Dark roads](#)
- [UVA 11733 - Airports](#)
- [UVA 11747 - Heavy Cycle Edges](#)
- [SPOJ - Blinet](#)
- [SPOJ - Help the Old King](#)
- [Codeforces - Hierarchy](#)

5.3 Minimum spanning tree - Kruskal with Disjoint Set Union

For an explanation of the MST problem and the Kruskal algorithm, first see the [main article on Kruskal's algorithm](#).

In this article we will consider the data structure "[Disjoint Set Union](#)" for implementing Kruskal's algorithm, which will allow the algorithm to achieve the time complexity of $O(M \log N)$.

5.3.1 Description

Just as in the simple version of the Kruskal algorithm, we sort all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e. its set) via calls to the `make_set` function - it will take a total of $O(N)$. We iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two `find_set` calls in $O(1)$ each). Finally, we need to perform the union of the two trees (sets), for which the DSU `union_sets` function will be called - also in $O(1)$. So we get the total time complexity of $O(M \log N + N + M) = O(M \log N)$.

5.3.2 Implementation

Here is an implementation of Kruskal's algorithm with Union by Rank.

```
vector<int> parent, rank;

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]++;
    }
}

struct Edge {
```

```
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<Edge> result;
parent.resize(n);
rank.resize(n);
for (int i = 0; i < n; i++)
    make_set(i);

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
        cost += e.weight;
        result.push_back(e);
        union_sets(e.u, e.v);
    }
}
```

5.3.3 Practice Problems

See [main article on Kruskal's algorithm](#) for the list of practice problems on this topic.

5.4 Kirchhoff's theorem. Finding the number of spanning trees

Problem: You are given a connected undirected graph (with possible multiple edges) represented using an adjacency matrix. Find the number of different spanning trees of this graph.

The following formula was proven by Kirchhoff in 1847.

5.4.1 Kirchhoff's matrix tree theorem

Let A be the adjacency matrix of the graph: $A_{u,v}$ is the number of edges between u and v . Let D be the degree matrix of the graph: a diagonal matrix with $D_{u,u}$ being the degree of vertex u (including multiple edges and loops - edges which connect vertex u with itself).

The Laplacian matrix of the graph is defined as $L = D - A$. According to Kirchhoff's theorem, all cofactors of this matrix are equal to each other, and they are equal to the number of spanning trees of the graph. The (i, j) cofactor of a matrix is the product of $(-1)^{i+j}$ with the determinant of the matrix that you get after removing the i -th row and j -th column. So you can, for example, delete the last row and last column of the matrix L , and the absolute value of the determinant of the resulting matrix will give you the number of spanning trees.

The determinant of the matrix can be found in $O(N^3)$ by using the [Gaussian method](#).

The proof of this theorem is quite difficult and is not presented here; for an outline of the proof and variations of the theorem for graphs without multiple edges and for directed graphs refer to [Wikipedia](#).

5.4.2 Relation to Kirchhoff's circuit laws

Kirchhoff's matrix tree theorem and Kirchhoff's laws for electrical circuit are related in a beautiful way. It is possible to show (using Ohm's law and Kirchhoff's first law) that resistance R_{ij} between two points of the circuit i and j is

$$R_{ij} = \frac{|L^{(i,j)}|}{|L^j|}.$$

Here the matrix L is obtained from the matrix of inverse resistances A ($A_{i,j}$ is inverse of the resistance of the conductor between points i and j) using the procedure described in Kirchhoff's matrix tree theorem. T^j is the matrix with row and column j removed, $T^{(i,j)}$ is the matrix with two rows and two columns i and j removed.

Kirchhoff's theorem gives this formula geometric meaning.

5.4.3 Practice Problems

- [CODECHEF: Roads in Stars](#)
- [SPOJ: Maze](#)

5.5 Prüfer code

In this article we will look at the so-called **Prüfer code** (or Prüfer sequence), which is a way of encoding a labeled tree into a sequence of numbers in a unique way.

With the help of the Prüfer code we will prove **Cayley's formula** (which specified the number of spanning trees in a complete graph). Also we show the solution to the problem of counting the number of ways of adding edges to a graph to make it connected.

Note, we will not consider trees consisting of a single vertex - this is a special case in which multiple statements clash.

5.5.1 Prüfer code

The Prüfer code is a way of encoding a labeled tree with n vertices using a sequence of $n - 2$ integers in the interval $[0; n - 1]$. This encoding also acts as a **bijection** between all spanning trees of a complete graph and the numerical sequences.

Although using the Prüfer code for storing and operating on tree is impractical due the specification of the representation, the Prüfer codes are used frequently: mostly in solving combinatorial problems.

The inventor - Heinz Prüfer - proposed this code in 1918 as a proof for Cayley's formula.

Building the Prüfer code for a given tree

The Prüfer code is constructed as follows. We will repeat the following procedure $n - 2$ times: we select the leaf of the tree with the smallest number, remove it from the tree, and write down the number of the vertex that was connected to it. After $n - 2$ iterations there will only remain 2 vertices, and the algorithm ends.

Thus the Prüfer code for a given tree is a sequence of $n - 2$ numbers, where each number is the number of the connected vertex, i.e. this number is in the interval $[0, n - 1]$.

The algorithm for computing the Prüfer code can be implemented easily with $O(n \log n)$ time complexity, simply by using a data structure to extract the minimum (for instance `set` or `priority_queue` in C++), which contains a list of all the current leafs.

```
vector<vector<int>> adj;

vector<int> pruefer_code() {
    int n = adj.size();
    set<int> leafs;
    vector<int> degree(n);
    vector<bool> killed(n, false);
    for (int i = 0; i < n; i++) {
        degree[i] = adj[i].size();
        if (degree[i] == 1)
```

```

        leafs.insert(i);
    }

    vector<int> code(n - 2);
    for (int i = 0; i < n - 2; i++) {
        int leaf = *leafs.begin();
        leafs.erase(leafs.begin());
        killed[leaf] = true;

        int v;
        for (int u : adj[leaf]) {
            if (!killed[u])
                v = u;
        }

        code[i] = v;
        if (--degree[v] == 1)
            leafs.insert(v);
    }

    return code;
}

```

However the construction can also be implemented in linear time. Such an approach is described in the next section.

Building the Prüfer code for a given tree in linear time

The essence of the algorithm is to use a **moving pointer**, which will always point to the current leaf vertex that we want to remove.

At first glance this seems impossible, because during the process of constructing the Prüfer code the leaf number can increase and decrease. However after a closer look, this is actually not true. The number of leafs will not increase. Either the number decreases by one (we remove one leaf vertex and don't gain a new one), or it stay the same (we remove one leaf vertex and gain another one). In the first case there is no other way than searching for the next smallest leaf vertex. In the second case, however, we can decide in $O(1)$ time, if we can continue using the vertex that became a new leaf vertex, or if we have to search for the next smallest leaf vertex. And in quite a lot of times we can continue with the new leaf vertex.

To do this we will use a variable `ptr`, which will indicate that in the set of vertices between 0 and `ptr` is at most one leaf vertex, namely the current one. All other vertices in that range are either already removed from the tree, or have still more than one adjacent vertices. At the same time we say, that we haven't removed any leaf vertices bigger than `ptr` yet.

This variable is already very helpful in the first case. After removing the current leaf node, we know that there cannot be a leaf node between 0 and `ptr`, therefore we can start the search for the next one directly at `ptr + 1`, and we don't have to start the search back at vertex 0. And in the second case, we can

further distinguish two cases: Either the newly gained leaf vertex is smaller than `ptr`, then this must be the next leaf vertex, since we know that there are no other vertices smaller than `ptr`. Or the newly gained leaf vertex is bigger. But then we also know that it has to be bigger than `ptr`, and can start the search again at `ptr + 1`.

Even though we might have to perform multiple linear searches for the next leaf vertex, the pointer `ptr` only increases and therefore the time complexity in total is $O(n)$.

```
vector<vector<int>>> adj;
vector<int> parent;

void dfs(int v) {
    for (int u : adj[v]) {
        if (u != parent[v]) {
            parent[u] = v;
            dfs(u);
        }
    }
}

vector<int> pruefer_code() {
    int n = adj.size();
    parent.resize(n);
    parent[n-1] = -1;
    dfs(n-1);

    int ptr = -1;
    vector<int> degree(n);
    for (int i = 0; i < n; i++) {
        degree[i] = adj[i].size();
        if (degree[i] == 1 && ptr == -1)
            ptr = i;
    }

    vector<int> code(n - 2);
    int leaf = ptr;
    for (int i = 0; i < n - 2; i++) {
        int next = parent[leaf];
        code[i] = next;
        if (--degree[next] == 1 && next < ptr) {
            leaf = next;
        } else {
            ptr++;
            while (degree[ptr] != 1)
                ptr++;
            leaf = ptr;
        }
    }

    return code;
}
```

In the code we first find for each its ancestor `parent[i]`, i.e. the ancestor that this vertex will have once we remove it from the tree. We can find this ancestor by rooting the tree at the vertex $n - 1$. This is possible because the vertex $n - 1$ will never be removed from the tree. We also compute the degree for each vertex. `ptr` is the pointer that indicates the minimum size of the remaining leaf vertices (except the current one `leaf`). We will either assign the current leaf vertex with `next`, if this one is also a leaf vertex and it is smaller than `ptr`, or we start a linear search for the smallest leaf vertex by increasing the pointer.

It can be easily seen, that this code has the complexity $O(n)$.

Some properties of the Prüfer code

- After constructing the Prüfer code two vertices will remain. One of them is the highest vertex $n - 1$, but nothing else can be said about the other one.
- Each vertex appears in the Prüfer code exactly a fixed number of times - its degree minus one. This can be easily checked, since the degree will get smaller every time we record its label in the code, and we remove it once the degree is 1. For the two remaining vertices this fact is also true.

Restoring the tree using the Prüfer code

To restore the tree it suffice to only focus on the property discussed in the last section. We already know the degree of all the vertices in the desired tree. Therefore we can find all leaf vertices, and also the first leaf that was removed in the first step (it has to be the smallest leaf). This leaf vertex was connected to the vertex corresponding to the number in the first cell of the Prüfer code.

Thus we found the first edge removed by when then the Prüfer code was generated. We can add this edge to the answer and reduce the degrees at both ends of the edge.

We will repeat this operation until we have used all numbers of the Prüfer code: we look for the minimum vertex with degree equal to 1, connect it with the next vertex from the Prüfer code, and reduce the degree.

In the end we only have two vertices left with degree equal to 1. These are the vertices that didn't got removed by the Prüfer code process. We connect them to get the last edge of the tree. One of them will always be the vertex $n - 1$.

This algorithm can be **implemented** easily in $O(n \log n)$: we use a data structure that supports extracting the minimum (for example `set<>` or `priority_queue<>` in C++) to store all the leaf vertices.

The following implementation returns the list of edges corresponding to the tree.

```
vector<pair<int, int>> pruefer_decode(vector<int> const& code) {
    int n = code.size() + 2;
    vector<int> degree(n, 1);
    for (int i : code)
        degree[i]++;

    set<int> leaves;
```

```

for (int i = 0; i < n; i++) {
    if (degree[i] == 1)
        leaves.insert(i);
}

vector<pair<int, int>> edges;
for (int v : code) {
    int leaf = *leaves.begin();
    leaves.erase(leaves.begin());

    edges.emplace_back(leaf, v);
    if (--degree[v] == 1)
        leaves.insert(v);
}
edges.emplace_back(*leaves.begin(), n-1);
return edges;
}

```

Restoring the tree using the Prüfer code in linear time

To obtain the tree in linear time we can apply the same technique used to obtain the Prüfer code in linear time.

We don't need a data structure to extract the minimum. Instead we can notice that, after processing the current edge, only one vertex becomes a leaf. Therefore we can either continue with this vertex, or we find a smaller one with a linear search by moving a pointer.

```

vector<pair<int, int>> pruefer_decode(vector<int> const& code) {
    int n = code.size() + 2;
    vector<int> degree(n, 1);
    for (int i : code)
        degree[i]++;

    int ptr = 0;
    while (degree[ptr] != 1)
        ptr++;
    int leaf = ptr;

    vector<pair<int, int>> edges;
    for (int v : code) {
        edges.emplace_back(leaf, v);
        if (--degree[v] == 1 && v < ptr) {
            leaf = v;
        } else {
            ptr++;
            while (degree[ptr] != 1)
                ptr++;
            leaf = ptr;
        }
    }
    edges.emplace_back(leaf, n-1);
}

```

```

    return edges;
}

```

Bijection between trees and Prüfer codes

For each tree there exists a Prüfer code corresponding to it. And for each Prüfer code we can restore the original tree.

It follows that also every Prüfer code (i.e. a sequence of $n - 2$ numbers in the range $[0; n - 1]$) corresponds to a tree.

Therefore all trees and all Prüfer codes form a bijection (a **one-to-one correspondence**).

5.5.2 Cayley's formula

Cayley's formula states that the **number of spanning trees in a complete labeled graph** with n vertices is equal to:

$$n^{n-2}$$

There are multiple proofs for this formula. Using the Prüfer code concept this statement comes without any surprise.

In fact any Prüfer code with $n - 2$ numbers from the interval $[0; n - 1]$ corresponds to some tree with n vertices. So we have n^{n-2} different such Prüfer codes. Since each such tree is a spanning tree of a complete graph with n vertices, the number of such spanning trees is also n^{n-2} .

5.5.3 Number of ways to make a graph connected

The concept of Prüfer codes are even more powerful. It allows to create a lot more general formulas than Cayley's formula.

In this problem we are given a graph with n vertices and m edges. The graph currently has k components. We want to compute the number of ways of adding $k - 1$ edges so that the graph becomes connected (obviously $k - 1$ is the minimum number necessary to make the graph connected).

Let us derive a formula for solving this problem.

We use s_1, \dots, s_k for the sizes of the connected components in the graph. We cannot add edges within a connected component. Therefore it turns out that this problem is very similar to the search for the number of spanning trees of a complete graph with k vertices. The only difference is that each vertex has actually the size s_i : each edge connecting the vertex i , actually multiplies the answer by s_i .

Thus in order to calculate the number of possible ways it is important to count how often each of the k vertices is used in the connecting tree. To obtain a formula for the problem it is necessary to sum the answer over all possible degrees.

Let d_1, \dots, d_k be the degrees of the vertices in the tree after connecting the vertices. The sum of the degrees is twice the number of edges:

$$\sum_{i=1}^k d_i = 2k - 2$$

If the vertex i has degree d_i , then it appears $d_i - 1$ times in the Prüfer code. The Prüfer code for a tree with k vertices has length $k - 2$. So the number of ways to choose a code with $k - 2$ numbers where the number i appears exactly $d_i - 1$ times is equal to the **multinomial coefficient**

$$\binom{k-2}{d_1-1, d_2-1, \dots, d_k-1} = \frac{(k-2)!}{(d_1-1)!(d_2-1)! \cdots (d_k-1)!}.$$

The fact that each edge adjacent to the vertex i multiplies the answer by s_i we receive the answer, assuming that the degrees of the vertices are d_1, \dots, d_k :

$$s_1^{d_1} \cdot s_2^{d_2} \cdots s_k^{d_k} \cdot \binom{k-2}{d_1-1, d_2-1, \dots, d_k-1}$$

To get the final answer we need to sum this for all possible ways to choose the degrees:

$$\sum_{\substack{d_i \geq 1 \\ \sum_{i=1}^k d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdots s_k^{d_k} \cdot \binom{k-2}{d_1-1, d_2-1, \dots, d_k-1}$$

Currently this looks like a really horrible answer, however we can use the **multinomial theorem**, which says:

$$(x_1 + \cdots + x_m)^p = \sum_{\substack{c_i \geq 0 \\ \sum_{i=1}^m c_i = p}} x_1^{c_1} \cdot x_2^{c_2} \cdots x_m^{c_m} \cdot \binom{p}{c_1, c_2, \dots, c_m}$$

This look already pretty similar. To use it we only need to substitute with $e_i = d_i - 1$:

$$\sum_{\substack{e_i \geq 0 \\ \sum_{i=1}^k e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdots s_k^{e_k+1} \cdot \binom{k-2}{e_1, e_2, \dots, e_k}$$

After applying the multinomial theorem we get the **answer to the problem**:

$$s_1 \cdot s_2 \cdots s_k \cdot (s_1 + s_2 + \cdots + s_k)^{k-2} = s_1 \cdot s_2 \cdots s_k \cdot n^{k-2}$$

By accident this formula also holds for $k = 1$.

5.5.4 Practice problems

- [UVA #10843 - Anne's game](#)
- [Timus #1069 - Prufer Code](#)
- [Codeforces - Clues](#)
- [Topcoder - TheCitiesAndRoadsDivTwo](#)

Chapter 6

Cycles

6.1 Checking a graph for acyclicity and finding a cycle in $O(M)$

Consider a directed or undirected graph without loops and multiple edges. We have to check whether it is acyclic, and if it is not, then find any cycle.

We can solve this problem by using [Depth First Search](#) in $O(M)$ where M is number of edges.

6.1.1 Algorithm

We will run a series of DFS in the graph. From each unvisited vertex, start the DFS, mark it gray while entering and mark it black on exit. If DFS moves to a gray vertex, then we have found a cycle (if the graph is undirected, the edge to parent is not considered). The cycle itself can be reconstructed using parent array.

6.1.2 Implementation

Here is an implementation for directed graph.

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
    color[v] = 1;
    for (int u : adj[v]) {
        if (color[u] == 0) {
            parent[u] = v;
            if (dfs(u))
                return true;
        } else if (color[u] == 1) {
            cycle_end = v;
        }
    }
    return false;
}
```

```
        cycle_start = u;
        return true;
    }
}
color[v] = 2;
return false;
}

void find_cycle() {
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int v = 0; v < n; v++) {
        if (dfs(v))
            break;
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
            cycle.push_back(v);
        cycle.push_back(cycle_start);
        reverse(cycle.begin(), cycle.end());

        cout << "Cycle found: ";
        for (int v : cycle)
            cout << v << " ";
        cout << endl;
    }
}
```

6.2 Finding a negative cycle in the graph

You are given a directed weighted graph G with N vertices and M edges. Find any cycle of negative weight in it, if such a cycle exists.

In another formulation of the problem you have to find all pairs of vertices which have a path of arbitrarily small weight between them.

It is convenient to use different algorithms to solve these two variations of the problem, so we'll discuss both of them here.

6.2.1 Using Bellman-Ford algorithm

Bellman-Ford algorithm allows you to check whether there exists a cycle of negative weight in the graph, and if it does, find one of these cycles.

The details of the algorithm are described in the article on the [Bellman-Ford](#) algorithm. Here we'll describe only its application to this problem.

Do N iterations of Bellman-Ford algorithm. If there were no changes on the last iteration, there is no cycle of negative weight in the graph. Otherwise take a vertex the distance to which has changed, and go from it via its ancestors until a cycle is found. This cycle will be the desired cycle of negative weight.

Implementation

```
struct Edge {
    int a, b, cost;
};

int n, m;
vector<Edge> edges;
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 (Edge e : edges) {
            if (d[e.a] + e.cost < d[e.b]) {
                d[e.b] = d[e.a] + e.cost;
                p[e.b] = e.a;
                x = e.b;
            }
        }
    }

    if (x == -1) {
        cout << "No negative cycle found.";
    } else {
        for (int i = 0; i < n; ++i)
```

```

        x = p[x];

        vector<int> cycle;
        for (int v = x;; v = p[v]) {
            cycle.push_back(v);
            if (v == x && cycle.size() > 1)
                break;
        }
        reverse(cycle.begin(), cycle.end());

        cout << "Negative cycle: ";
        for (int v : cycle)
            cout << v << ' ';
        cout << endl;
    }
}

```

6.2.2 Using Floyd-Warshall algorithm

The Floyd-Warshall algorithm allows to solve the second variation of the problem - finding all pairs of vertices (i, j) which don't have a shortest path between them (i.e. a path of arbitrarily small weight exists).

Again, the details can be found in the [Floyd-Warshall](#) article, and here we describe only its application.

Run Floyd-Warshall algorithm on the graph. Initially $d[v][v] = 0$ for each v . But after running the algorithm $d[v][v]$ will be smaller than 0 if there exists a negative length path from v to v . We can use this to also find all pairs of vertices that don't have a shortest path between them. We iterate over all pairs of vertices (i, j) and for each pair we check whether they have a shortest path between them. To do this try all possibilities for an intermediate vertex t . (i, j) doesn't have a shortest path, if one of the intermediate vertices t has $d[t][t] < 0$ (i.e. t is part of a cycle of negative weight), t can be reached from i and j can be reached from t . Then the path from i to j can have arbitrarily small weight. We will denote this with $-\text{INF}$.

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;
        }
    }
}

```

6.2.3 Practice Problems

- [UVA: Wormholes](#)
- [SPOJ: Alice in Amsterdam, I mean Wonderland](#)

- [SPOJ: Johnsons Algorithm](#)

6.3 Finding the Eulerian path in $O(M)$

A Eulerian path is a path in a graph that passes through all of its edges exactly once. A Eulerian cycle is a Eulerian path that is a cycle.

The problem is to find the Eulerian path in an **undirected multigraph with loops**.

6.3.1 Algorithm

First we can check if there is an Eulerian path. We can use the following theorem. An Eulerian cycle exists if and only if the degrees of all vertices are even. And an Eulerian path exists if and only if the number of vertices with odd degrees is two (or zero, in the case of the existence of a Eulerian cycle). In addition, of course, the graph must be sufficiently connected (i.e., if you remove all isolated vertices from it, you should get a connected graph).

To find the Eulerian path / Eulerian cycle we can do the following strategies: We find all simple cycles and combine them into one - this will be the Eulerian cycle. If the graph is such that the Eulerian path is not a cycle, then add the missing edge, find the Eulerian cycle, then remove the extra edge.

Looking for all cycles and combining them can be done with a simple recursive procedure:

procedure FindEulerPath(V)

1. iterate through all the edges outgoing from vertex V ;
 remove this edge from the graph,
 and call FindEulerPath from the second end of this edge;
2. add vertex V to the answer.

The complexity of this algorithm is obviously linear with respect to the number of edges.

But we can write the same algorithm in the non-recursive version:

```
stack St;
put start vertex in St;
until St is empty
  let  $V$  be the value at the top of St;
  if degree( $V$ ) = 0, then
    add  $V$  to the answer;
    remove  $V$  from the top of St;
  otherwise
    find any edge coming out of  $V$ ;
    remove it from the graph;
    put the second end of this edge in St;
```

It is easy to check the equivalence of these two forms of the algorithm. However, the second form is obviously faster, and the code will be much more.

6.3.2 The Domino problem

We give here a classical Eulerian cycle problem - the Domino problem.

There are N dominoes, as it is known, on both ends of the Domino one number is written (usually from 1 to 6, but in our case it is not important). You want to put all the dominoes in a row so that the numbers on any two adjacent dominoes, written on their common side, coincide. Dominoes are allowed to turn.

Reformulate the problem. Let the numbers written on the bottoms be the vertices of the graph, and the dominoes be the edges of this graph (each Domino with numbers (a, b) are the edges (a, b) and (b, a)). Then our problem is reduced to the problem of finding the Eulerian path in this graph.

6.3.3 Implementation

The program below searches for and outputs a Eulerian loop or path in a graph, or outputs -1 if it does not exist.

First, the program checks the degree of vertices: if there are no vertices with an odd degree, then the graph has an Euler cycle, if there are 2 vertices with an odd degree, then in the graph there is only an Euler path (but no Euler cycle), if there are more than 2 such vertices, then in the graph there is no Euler cycle or Euler path. To find the Euler path (not a cycle), let's do this: if $V1$ and $V2$ are two vertices of odd degree, then just add an edge $(V1, V2)$, in the resulting graph we find the Euler cycle (it will obviously exist), and then remove the "fictitious" edge $(V1, V2)$ from the answer. We will look for the Euler cycle exactly as described above (non-recursive version), and at the same time at the end of this algorithm we will check whether the graph was connected or not (if the graph was not connected, then at the end of the algorithm some edges will remain in the graph, and in this case we need to print -1). Finally, the program takes into account that there can be isolated vertices in the graph.

Notice that we use an adjacency matrix in this problem. Also this implementation handles finding the next with brute-force, which requires to iterate over the complete row in the matrix over and over. A better way would be to store the graph as an adjacency list, and remove edges in $O(1)$ and mark the reversed edges in separate list. This way we can archive a $O(N)$ algorithm.

```
int main() {
    int n;
    vector<vector<int>>> g(n, vector<int>(n));
    // reading the graph in the 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) {
        cout << -1;
    } else {
        for (int x : res)
            cout << x << " ";
    }
}
```

Chapter 7

Lowest common ancestor

7.1 Lowest Common Ancestor - $O(\sqrt{N})$ and $O(\log N)$ with $O(N)$ preprocessing

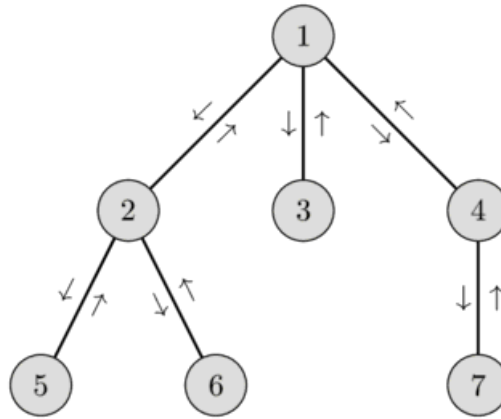
Given a tree G . Given queries of the form (v_1, v_2) , for each query you need to find the lowest common ancestor (or least common ancestor), i.e. a vertex v that lies on the path from the root to v_1 and the path from the root to v_2 , and the vertex should be the lowest. In other words, the desired vertex v is the most bottom ancestor of v_1 and v_2 . It is obvious that their lowest common ancestor lies on a shortest path from v_1 and v_2 . Also, if v_1 is the ancestor of v_2 , v_1 is their lowest common ancestor.

The Idea of the Algorithm

Before answering the queries, we need to **preprocess** the tree. We make a [DFS](#) traversal starting at the root and we build a list `euler` which stores the order of the vertices that we visit (a vertex is added to the list when we first visit it, and after the return of the DFS traversals to its children). This is also called an Euler tour of the tree. It is clear that the size of this list will be $O(N)$. We also need to build an array `first[0..N - 1]` which stores for each vertex i its first occurrence in `euler`. That is, the first position in `euler` such that `euler[first[i]] = i`. Also by using the DFS we can find the height of each node (distance from root to it) and store it in the array `height[0..N - 1]`.

So how can we answer queries using the Euler tour and the additional two arrays? Suppose the query is a pair of v_1 and v_2 . Consider the vertices that we visit in the Euler tour between the first visit of v_1 and the first visit of v_2 . It is easy to see, that the $LCA(v_1, v_2)$ is the vertex with the lowest height on this path. We already noticed, that the LCA has to be part of the shortest path between v_1 and v_2 . Clearly it also has to be the vertex with the smallest height. And in the Euler tour we essentially use the shortest path, except that we additionally visit all subtrees that we find on the path. But all vertices in these subtrees are lower in the tree than the LCA and therefore have a larger height. So the $LCA(v_1, v_2)$ can be uniquely determined by finding the vertex with the smallest height in the Euler tour between `first(v_1)` and `first(v_2)`.

Let's illustrate this idea. Consider the following graph and the Euler tour with the corresponding heights:



Vertices:	1	2	5	2	6	2	1	3	1	4	7	4	1
Heights:	1	2	3	2	3	2	1	2	1	2	3	2	1

The tour starting at vertex 6 and ending at 4 we visit the vertices $[6, 2, 1, 3, 1, 4]$. Among those vertices the vertex 1 has the lowest height, therefore $\text{LCA}(6, 4) = 1$.

To recap: to answer a query we just need **to find the vertex with smallest height** in the array euler in the range from $\text{first}[v_1]$ to $\text{first}[v_2]$. Thus, **the LCA problem is reduced to the RMQ problem** (finding the minimum in an range problem).

Using [Sqrt-Decomposition](#), it is possible to obtain a solution answering each query in $O(\sqrt{N})$ with preprocessing in $O(N)$ time.

Using a [Segment Tree](#) you can answer each query in $O(\log N)$ with preprocessing in $O(N)$ time.

Implementation

In the following implementation of the LCA algorithm a Segment Tree is used.

```
struct LCA {
    vector<int> height, euler, first, segtree;
    vector<bool> visited;
    int n;

    LCA(vector<vector<int>> &adj, int root = 0) {
        n = adj.size();
        height.resize(n);
        first.resize(n);
        euler.reserve(n * 2);
        visited.assign(n, false);
        dfs(adj, root);
        int m = euler.size();
```

```

    segtree.resize(m * 4);
    build(1, 0, m - 1);
}

void dfs(vector<vector<int>> &adj, int node, int h = 0) {
    visited[node] = true;
    height[node] = h;
    first[node] = euler.size();
    euler.push_back(node);
    for (auto to : adj[node]) {
        if (!visited[to]) {
            dfs(adj, to, h + 1);
            euler.push_back(node);
        }
    }
}

void build(int node, int b, int e) {
    if (b == e) {
        segtree[node] = euler[b];
    } else {
        int mid = (b + e) / 2;
        build(node << 1, b, mid);
        build(node << 1 | 1, mid + 1, e);
        int l = segtree[node << 1], r = segtree[node << 1 | 1];
        segtree[node] = (height[l] < height[r]) ? l : r;
    }
}

int query(int node, int b, int e, int L, int R) {
    if (b > R || e < L)
        return -1;
    if (b >= L && e <= R)
        return segtree[node];
    int mid = (b + e) >> 1;

    int left = query(node << 1, b, mid, L, R);
    int right = query(node << 1 | 1, mid + 1, e, L, R);
    if (left == -1) return right;
    if (right == -1) return left;
    return height[left] < height[right] ? left : right;
}

int lca(int u, int v) {
    int left = first[u], right = first[v];
    if (left > right)
        swap(left, right);
    return query(1, 0, euler.size() - 1, left, right);
}
};

```

7.1.1 Practice Problems

- [SPOJ: LCA](#)
- [SPOJ: DISQUERY](#)
- [TIMUS: 1471. Distance in the Tree](#)
- [CODEFORCES: Design Tutorial: Inverse the Problem](#)
- [CODECHEF: Lowest Common Ancestor](#)
- [SPOJ - Lowest Common Ancestor](#)
- [SPOJ - Ada and Orange Tree](#)
- [DevSkills - Motoku](#)
- [UVA 12655 - Trucks](#)
- [Codechef - Pishty and Tree](#)
- [UVA - 12533 - Joining Couples](#)
- [Codechef - So close yet So Far](#)
- [Codeforces - Drivers Dissatisfaction](#)
- [UVA 11354 - Bond](#)
- [SPOJ - Query on a tree II](#)
- [Codeforces - Best Edge Wiegth](#)
- [Codeforces - Misha, Grisha and Underground](#)
- [SPOJ - Nlogonian Tickets](#)
- [Codeforces - Rowena Rawenclaws Diadem](#)

7.2 Lowest Common Ancestor - Binary Lifting

Let G be a tree. For every query of the form (u, v) we want to find the lowest common ancestor of the nodes u and v , i.e. we want to find a node w that lies on the path from u to the root node, that lies on the path from v to the root node, and if there are multiple nodes we pick the one that is farthest away from the root node. In other words the desired node w is the lowest ancestor of u and v . In particular if u is an ancestor of v , then u is their lowest common ancestor.

The algorithm described in this article will need $O(N \log N)$ for preprocessing the tree, and then $O(\log N)$ for each LCA query.

7.2.1 Algorithm

For each node we will precompute its ancestor above him, its ancestor two nodes above, its ancestor four above, etc. Let's store them in the array `up`, i.e. `up[i][j]` is the 2^j -th ancestor above the node i with $i=1 \dots N$, $j=0 \dots \text{ceil}(\log(N))$. These information allow us to jump from any node to any ancestor above it in $O(\log N)$ time. We can compute this array using a [DFS](#) traversal of the tree.

For each node we will also remember the time of the first visit of this node (i.e. the time when the DFS discovers the node), and the time when we left it (i.e. after we visited all children and exit the DFS function). We can use this information to determine in constant time if a node is an ancestor of another node.

Suppose now we received a query (u, v) . We can immediately check whether one node is the ancestor of the other. In this case this node is already the LCA. If u is not the ancestor of v , and v not the ancestor of u , we climb the ancestors of u until we find the highest (i.e. closest to the root) node, which is not an ancestor of v (i.e. a node x , such that x is not an ancestor of v , but `up[x][0]` is). We can find this node x in $O(\log N)$ time using the array `up`.

We will describe this process in more detail. Let $L = \text{ceil}(\log(N))$. Suppose first that $i = L$. If `up[u][i]` is not an ancestor of v , then we can assign $u = \text{up}[u][i]$ and decrement i . If `up[u][i]` is not an ancestor, then we just decrement i . Clearly after doing this for all non-negative i the node u will be the desired node - i.e. u is still not an ancestor of v , but `up[u][0]` is.

Now, obviously, the answer to LCA will be `up[u][0]` - i.e., the smallest node among the ancestors of the node u , which is also an ancestor of v .

So answering a LCA query will iterate i from $\text{ceil}(\log(N))$ to 0 and checks in each iteration if one node is the ancestor of the other. Consequently each query can be answered in $O(\log N)$.

7.2.2 Implementation

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
```

```
vector<vector<int>> up;

void dfs(int v, int p)
{
    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 (int u : adj[v]) {
        if (u != p)
            dfs(u, v);
    }

    tout[v] = ++timer;
}

bool is_ancestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

7.3 Lowest Common Ancestor - Farach-Colton and Bender Algorithm

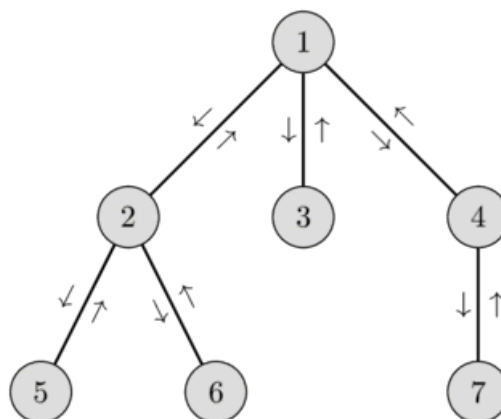
Let G be a tree. For every query of the form (u, v) we want to find the lowest common ancestor of the nodes u and v , i.e. we want to find a node w that lies on the path from u to the root node, that lies on the path from v to the root node, and if there are multiple nodes we pick the one that is farthest away from the root node. In other words the desired node w is the lowest ancestor of u and v . In particular if u is an ancestor of v , then u is their lowest common ancestor.

The algorithm which will be described in this article was developed by Farach-Colton and Bender. It is asymptotically optimal.

7.3.1 Algorithm

We use the classical reduction of the LCA problem to the RMQ problem. We traverse all nodes of the tree with [DFS](#) and keep an array with all visited nodes and the heights of these nodes. The LCA of two nodes u and v is the node between the occurrences of u and v in the tour, that has the smallest height.

In the following picture you can see a possible Euler-Tour of a graph and in the list below you can see the visited nodes and their heights.



Nodes:	1	2	5	2	6	2	1	3	1	4	7	4	1
Heights:	1	2	3	2	3	2	1	2	1	2	3	2	1

You can read more about this reduction in the article [Lowest Common Ancestor](#). In that article the minimum of a range was either found by sqrt-decomposition in $O(\sqrt{N})$ or in $O(\log N)$ using a Segment tree. In this article we look at how we can solve the given range minimum queries in $O(1)$ time, while still only taking $O(N)$ time for preprocessing.

Note that the reduced RMQ problem is very specific: any two adjacent elements in the array differ exactly by one (since the elements of the array are nothing more than the heights of the nodes visited in order of traversal, and we

either go to a descendant, in which case the next element is one bigger, or go back to the ancestor, in which case the next element is one lower). The Farach-Colton and Bender algorithm describes a solution for exactly this specialized RMQ problem.

Let's denote with A the array on which we want to perform the range minimum queries. And N will be the size of A .

There is an easy data structure that we can use for solving the RMQ problem with $O(N \log N)$ preprocessing and $O(1)$ for each query: the [Sparse Table](#). We create a table T where each element $T[i][j]$ is equal to the minimum of A in the interval $[i, i + 2^j - 1]$. Obviously $0 \leq j \leq \lceil \log N \rceil$, and therefore the size of the Sparse Table will be $O(N \log N)$. You can build the table easily in $O(N \log N)$ by noting that $T[i][j] = \min(T[i][j-1], T[i + 2^{j-1}][j-1])$.

How can we answer a query RMQ in $O(1)$ using this data structure? Let the received query be $[l, r]$, then the answer is $\min(T[l][sz], T[r - 2^{sz} + 1][sz])$, where sz is the biggest exponent such that 2^{sz} is not bigger than the range length $r - l + 1$. Indeed we can take the range $[l, r]$ and cover it two segments of length 2^{sz} - one starting in l and the other ending in r . These segments overlap, but this doesn't interfere with our computation. To really achieve the time complexity of $O(1)$ per query, we need to know the values of sz for all possible lengths from 1 to N . But this can be easily precomputed.

Now we want to improve the complexity of the preprocessing down to $O(N)$.

We divide the array A into blocks of size $K = 0.5 \log N$ with \log being the logarithm to base 2. For each block we calculate the minimum element and store them in an array B . B has the size $\frac{N}{K}$. We construct a sparse table from the array B . The size and the time complexity of it will be:

$$\begin{aligned} \frac{N}{K} \log \left(\frac{N}{K} \right) &= \frac{2N}{\log(N)} \log \left(\frac{2N}{\log(N)} \right) = \\ &= \frac{2N}{\log(N)} \left(1 + \log \left(\frac{N}{\log(N)} \right) \right) \leq \frac{2N}{\log(N)} + 2N = O(N) \end{aligned}$$

Now we only have to learn how to quickly answer range minimum queries within each block. In fact if the received range minimum query is $[l, r]$ and l and r are in different blocks then the answer is the minimum of the following three values: the minimum of the suffix of block of l starting at l , the minimum of the prefix of block of r ending at r , and the minimum of the blocks between those. The minimum of the blocks in between can be answered in $O(1)$ using the Sparse Table. So this leaves us only the range minimum queries inside blocks.

Here we will exploit the property of the array. Remember that the values in the array - which are just height values in the tree - will always differ by one. If we remove the first element of a block, and subtract it from every other item in the block, every block can be identified by a sequence of length $K - 1$ consisting of the number $+1$ and -1 . Because these blocks are so small, there are only a few different sequences that can occur. The number of possible sequences is:

$$2^{K-1} = 2^{0.5 \log(N)-1} = 0.5 \left(2^{\log(N)} \right)^{0.5} = 0.5 \sqrt{N}$$

Thus the number of different blocks is $O(\sqrt{N})$, and therefore we can precompute the results of range minimum queries inside all different blocks in $O(\sqrt{N}K^2) = O(\sqrt{N}\log^2(N)) = O(N)$ time. For the implementation we can characterize a block by a bitmask of length $K - 1$ (which will fit in a standard int) and store the index of the minimum in an array `block[mask][l][r]` of size $O(\sqrt{N}\log^2(N))$.

So we learned how to precompute range minimum queries within each block, as well as range minimum queries over a range of blocks, all in $O(N)$. With these precomputations we can answer each query in $O(1)$, by using at most four precomputed values: the minimum of the block containing `l`, the minimum of the block containing `r`, and the two minima of the overlapping segments of the blocks between them.

7.3.2 Implementation

```
int n;
vector<vector<int>> adj;

int block_size, block_cnt;
vector<int> first_visit;
vector<int> euler_tour;
vector<int> height;
vector<int> log_2;
vector<vector<int>> st;
vector<vector<vector<int>>> blocks;
vector<int> block_mask;

void dfs(int v, int p, int h) {
    first_visit[v] = euler_tour.size();
    euler_tour.push_back(v);
    height[v] = h;

    for (int u : adj[v]) {
        if (u == p)
            continue;
        dfs(u, v, h + 1);
        euler_tour.push_back(v);
    }
}

int min_by_h(int i, int j) {
    return height[euler_tour[i]] < height[euler_tour[j]] ? i : j;
}

void precompute_lca(int root) {
    // get euler tour & indices of first occurrences
    first_visit.assign(n, -1);
    height.assign(n, 0);
    euler_tour.reserve(2 * n);
    dfs(root, -1, 0);
```

```

// precompute all log values
int m = euler_tour.size();
log_2.reserve(m + 1);
log_2.push_back(-1);
for (int i = 1; i <= m; i++)
    log_2.push_back(log_2[i / 2] + 1);

block_size = max(1, log_2[m] / 2);
block_cnt = (m + block_size - 1) / block_size;

// precompute minimum of each block and build sparse table
st.assign(block_cnt, vector<int>(log_2[block_cnt] + 1));
for (int i = 0, j = 0, b = 0; i < m; i++, j++) {
    if (j == block_size)
        j = 0, b++;
    if (j == 0 || min_by_h(i, st[b][0]) == i)
        st[b][0] = i;
}
for (int l = 1; l <= log_2[block_cnt]; l++) {
    for (int i = 0; i < block_cnt; i++) {
        int ni = i + (1 << (l - 1));
        if (ni >= block_cnt)
            st[i][l] = st[i][l-1];
        else
            st[i][l] = min_by_h(st[i][l-1], st[ni][l-1]);
    }
}

// precompute mask for each block
block_mask.assign(block_cnt, 0);
for (int i = 0, j = 0, b = 0; i < m; i++, j++) {
    if (j == block_size)
        j = 0, b++;
    if (j > 0 && (i >= m || min_by_h(i - 1, i) == i - 1))
        block_mask[b] += 1 << (j - 1);
}

// precompute RMQ for each unique block
int possibilities = 1 << (block_size - 1);
blocks.resize(possibilities);
for (int b = 0; b < block_cnt; b++) {
    int mask = block_mask[b];
    if (!blocks[mask].empty())
        continue;
    blocks[mask].assign(block_size, vector<int>(block_size));
    for (int l = 0; l < block_size; l++) {
        blocks[mask][l][l] = 1;
        for (int r = l + 1; r < block_size; r++) {
            blocks[mask][l][r] = blocks[mask][l][r - 1];
            if (b * block_size + r < m)
                blocks[mask][l][r] = min_by_h(b * block_size + blocks[mask][l][r],
                    b * block_size + r) - b * block_size;
        }
    }
}

```

```

        }
    }
}

int lca_in_block(int b, int l, int r) {
    return blocks[block_mask[b]][l][r] + b * block_size;
}

int lca(int v, int u) {
    int l = first_visit[v];
    int r = first_visit[u];
    if (l > r)
        swap(l, r);
    int bl = l / block_size;
    int br = r / block_size;
    if (bl == br)
        return euler_tour[lca_in_block(bl, l % block_size, r % block_size)];
    int ans1 = lca_in_block(bl, l % block_size, block_size - 1);
    int ans2 = lca_in_block(br, 0, r % block_size);
    int ans = min_by_h(ans1, ans2);
    if (bl + 1 < br) {
        int l = log_2[br - bl - 1];
        int ans3 = st[bl+1][l];
        int ans4 = st[br - (1 << l)][l];
        ans = min_by_h(ans, min_by_h(ans3, ans4));
    }
    return euler_tour[ans];
}

```

7.4 Solve RMQ (Range Minimum Query) by finding LCA (Lowest Common Ancestor)

Given an array $A[0..N-1]$. For each query of the form $[L, R]$ we want to find the minimum in the array A starting from position L and ending with position R . We will assume that the array A doesn't change in the process, i.e. this article describes a solution to the static RMQ problem

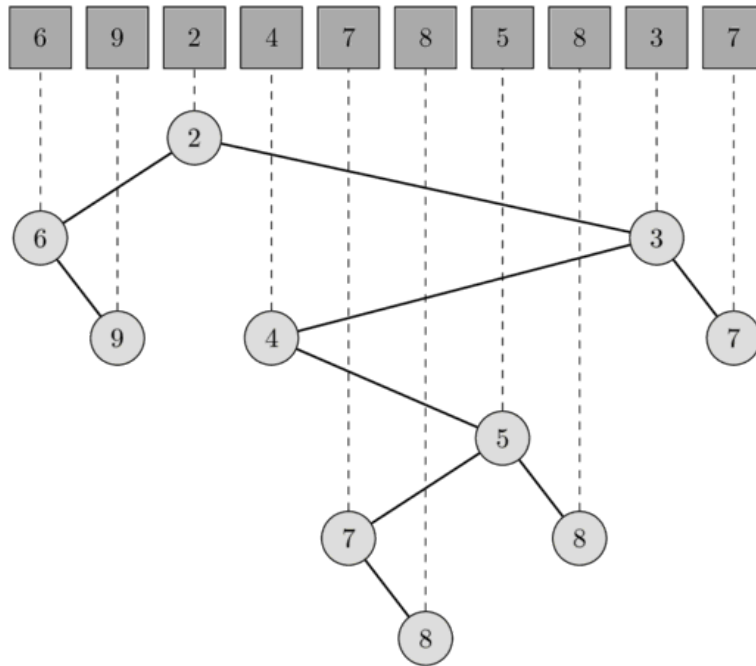
Here is a description of an asymptotically optimal solution. It stands apart from other solutions for the RMQ problem, since it is very different from them: it reduces the RMQ problem to the LCA problem, and then uses the [Farach-Colton and Bender algorithm](#), which reduces the LCA problem back to a specialized RMQ problem and solves that.

7.4.1 Algorithm

We construct a **Cartesian tree** from the array A . A Cartesian tree of an array A is a binary tree with the min-heap property (the value of parent node has to be smaller or equal than the value of its children) such that the in-order traversal of the tree visits the nodes in the same order as they are in the array A .

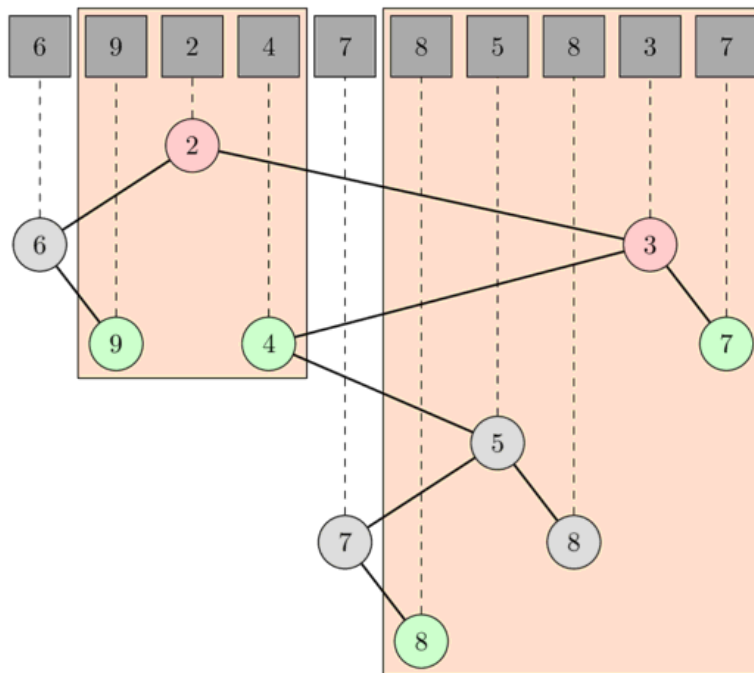
In other words, a Cartesian tree is a recursive data structure. The array A will be partitioned into 3 parts: the prefix of the array up to the minimum, the minimum, and the remaining suffix. The root of the tree will be a node corresponding to the minimum element of the array A , the left subtree will be the Cartesian tree of the prefix, and the right subtree will be a Cartesian tree of the suffix.

In the following image you can see one array of length 10 and the corresponding Cartesian tree.



The range minimum query $[l, r]$ is equivalent to the lowest common ancestor query $[l', r']$, where l' is the node corresponding to the element $A[l]$ and r' the node corresponding to the element $A[r]$. Indeed the node corresponding to the smallest element in the range has to be an ancestor of all nodes in the range, therefore also from l' and r' . This automatically follows from the min-heap property. And is also has to be the lowest ancestor, because otherwise l' and r' would be both in the left or in the right subtree, which generates a contradiction since in such a case the minimum wouldn't even be in the range.

In the following image you can see the LCA queries for the RMQ queries $[1, 3]$ and $[5, 9]$. In the first query the LCA of the nodes $A[1]$ and $A[3]$ is the node corresponding to $A[2]$ which has the value 2, and in the second query the LCA of $A[5]$ and $A[9]$ is the node corresponding to $A[8]$ which has the value 3.



Such a tree can be built in $O(N)$ time and the Farach-Colton and Benders algorithm can preprocess the tree in $O(N)$ and find the LCA in $O(1)$.

7.4.2 Construction of a Cartesian tree

We will build the Cartesian tree by adding the elements one after another. In each step we maintain a valid Cartesian tree of all the processed elements. It is easy to see, that adding an element $s[i]$ can only change the nodes in the most right path - starting at the root and repeatedly taking the right child - of the tree. The subtree of the node with the smallest, but greater or equal than $s[i]$, value becomes the left subtree of $s[i]$, and the tree with root $s[i]$ will become the new right subtree of the node with the biggest, but smaller than $s[i]$ value.

This can be implemented by using a stack to store the indices of the most right nodes.

```
vector<int> parent(n, -1);
stack<int> s;
for (int i = 0; i < n; i++) {
    int last = -1;
    while (!s.empty() && A[s.top()] >= A[i]) {
        last = s.top();
        s.pop();
    }
    if (!s.empty())
        parent[i] = s.top();
    if (last >= 0)
        parent[last] = i;
}
```

```
        s.push(i);  
    }
```


7.5 Lowest Common Ancestor - Tarjan's off-line algorithm

We have a tree G with n nodes and we have m queries of the form (u, v) . For each query (u, v) we want to find the lowest common ancestor of the vertices u and v , i.e. the node that is an ancestor of both u and v and has the greatest depth in the tree. The node v is also an ancestor of v , so the LCA can also be one of the two nodes.

In this article we will solve the problem off-line, i.e. we assume that all queries are known in advance, and we therefore answer the queries in any order we like. The following algorithm allows to answer all m queries in $O(n + m)$ total time, i.e. for sufficiently large m in $O(1)$ for each query.

7.5.1 Algorithm

The algorithm is named after Robert Tarjan, who discovered it in 1979 and also made many other contributions to the [Disjoint Set Union](#) data structure, which will be heavily used in this algorithm.

The algorithm answers all queries with one [DFS](#) traversal of the tree. Namely a query (u, v) is answered at node u , if node v has already been visited previously, or vice versa.

So let's assume we are currently at node v , we have already made recursive DFS calls, and also already visited the second node u from the query (u, v) . Let's learn how to find the LCA of these two nodes.

Note that $LCA(u, v)$ is either the node v or one of its ancestors. So we need to find the lowest node among the ancestors of v (including v), for which the node u is a descendant. Also note that for a fixed v the visited nodes of the tree split into a set of disjoint sets. Each ancestor p of node v has his own set containing this node and all subtrees with roots in those of its children who are not part of the path from v to the root of the tree. The set which contains the node u determines the $LCA(u, v)$: the LCA is the representative of the set, namely the node on lies on the path between v and the root of the tree.

We only need to learn to efficiently maintain all these sets. For this purpose we apply the data structure DSU. To be able to apply Union by rank, we store the real representative (the value on the path between v and the root of the tree) of each set in the array `ancestor`.

Let's discuss the implementation of the DFS. Let's assume we are currently visiting the node v . We place the node in a new set in the DSU, `ancestor[v] = v`. As usual we process all children of v . For this we must first recursively call DFS from that node, and then add this node with all its subtree to the set of v . This can be done with the function `union_sets` and the following assignment `ancestor[find_set(v)] = v` (this is necessary, because `union_sets` might change the representative of the set).

Finally after processing all children we can answer all queries of the form (u, v) for which u has been already visited. The answer to the query, i.e. the LCA

of u and v , will be the node `ancestor[find_set(u)]`. It is easy to see that a query will only be answered once.

Let's us determine the time complexity of this algorithm. Firstly we have $O(n)$ because of the DFS. Secondly we have the function calls of `union_sets` which happen n times, resulting also in $O(n)$. And thirdly we have the calls of `find_set` for every query, which gives $O(m)$. So in total the time complexity is $O(n + m)$, which means that for sufficiently large m this corresponds to $O(1)$ for answering one query.

7.5.2 Implementation

Here is an implementation of this algorithm. The implementation of DSU has been not included, as it can be used without any modifications.

```
vector<vector<int>> adj;
vector<vector<int>> queries;
vector<int> ancestor;
vector<bool> visited;

void dfs(int v)
{
    visited[v] = true;
    ancestor[v] = v;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u);
            union_sets(v, u);
            ancestor[find_set(v)] = v;
        }
    }
    for (int other_node : queries[v]) {
        if (visited[other_node])
            cout << "LCA of " << v << " and " << other_node
                << " is " << ancestor[find_set(other_node)] << ".\n";
    }
}

void compute_LCAs() {
    // initialize n, adj and DSU
    // for (each query (u, v)) {
    //     queries[u].push_back(v);
    //     queries[v].push_back(u);
    // }

    ancestor.resize(n);
    visited.assign(n, false);
    dfs(0);
}
```

Chapter 8

Flows and related problems

8.1 Maximum flow - Ford-Fulkerson and Edmonds-Karp

The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method for computing a maximal flow in a flow network.

8.1.1 Flow network

First let's define what a **flow network**, a **flow**, and a **maximum flow** is.

A **network** is a directed graph G with vertices V and edges E combined with a function c , which assigns each edge $e \in E$ a non-negative integer value, the **capacity** of e . Such a network is called a **flow network**, if we additionally label two vertices, one as **source** and one as **sink**.

A **flow** in a flow network is function f , that again assigns each edge e a non-negative integer value, namely the flow. The function has to fulfill the following two conditions:

The flow of an edge cannot exceed the capacity.

$$f(e) \leq c(e)$$

And the sum of the incoming flow of a vertex u has to be equal to the sum of the outgoing flow of u except in the source and sink vertices.

$$\sum_{(v,u) \in E} f((v,u)) = \sum_{(u,v) \in E} f((u,v))$$

The source vertex s only has an outgoing flow, and the sink vertex t has only incoming flow.

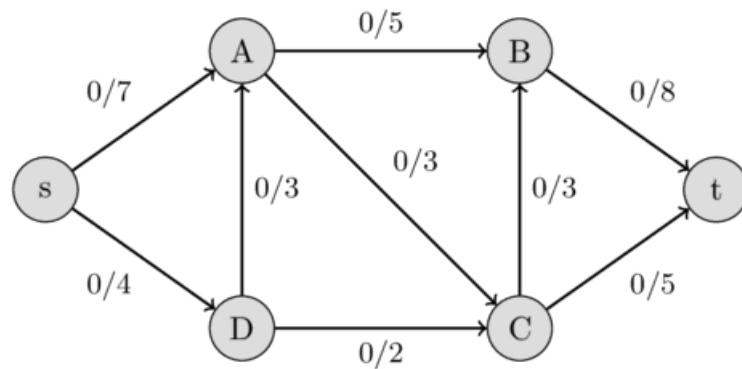
It is easy to see that the following equation holds:

$$\sum_{(s,u) \in E} f((s,u)) = \sum_{(u,t) \in E} f((u,t))$$

A good analogy for a flow network is the following visualization: We represent edges as water pipes, the capacity of an edge is the maximal amount of water

that can flow through the pipe per second, and the flow of an edge is the amount of water that currently flows through the pipe per second. This motivates the first flow condition. There cannot flow more water through a pipe than its capacity. The vertices act as junctions, where water comes out of some pipes, and distributes it in some way to other pipes. This also motivates the second flow condition. In each junction all the incoming water has to be distributed to the other pipes. It cannot magically disappear or appear. The source s is origin of all the water, and the water can only drain in the sink t .

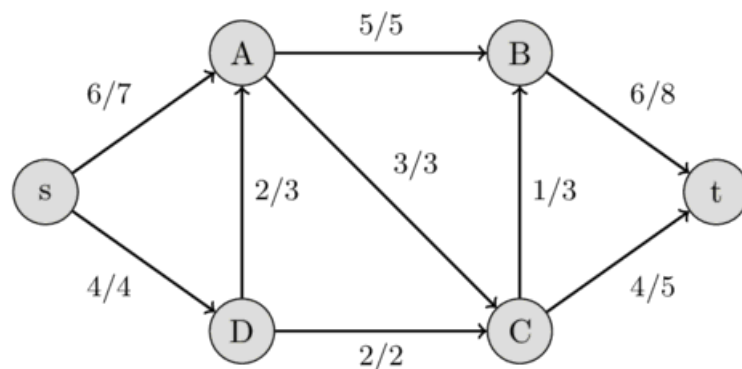
The following image show a flow network. The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.



We value of a flow of a network is the sum of all flows that gets produced in source s , or equivalent or the flows that is consumed in the sink t . A **maximal flow** is a flow with the maximal possible value. Finding this maximal flow of a flow network is the problem that we want to solve.

In the visualization with water pipes, the problem can be formulated in the following way: how much water can we push through the pipes from the source to the sink.

The following image show the maximal flow in the flow network.



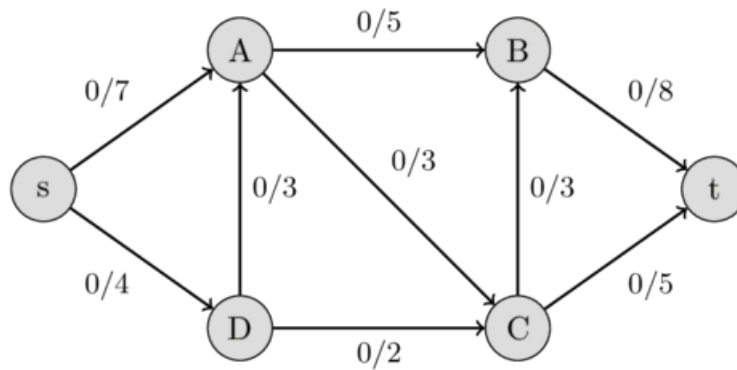
8.1.2 Ford-Fulkerson method

Let's define one more thing. A **residual capacity** of an directed edge is the capacity minus the flow. It should be noted that if there is a flow along some directed edge (u, v) , then the reversed edge has capacity 0 and we can define the flow of it as $f((v, u)) = -f((u, v))$. This also defines the residual capacity for all reversed edges. From all these edges we can create a **residual network**, which is just a network with the same vertices and same edges, but we use the residual capacities as capacities.

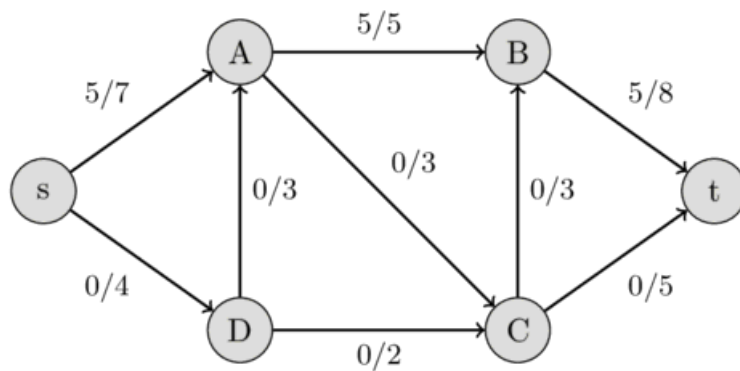
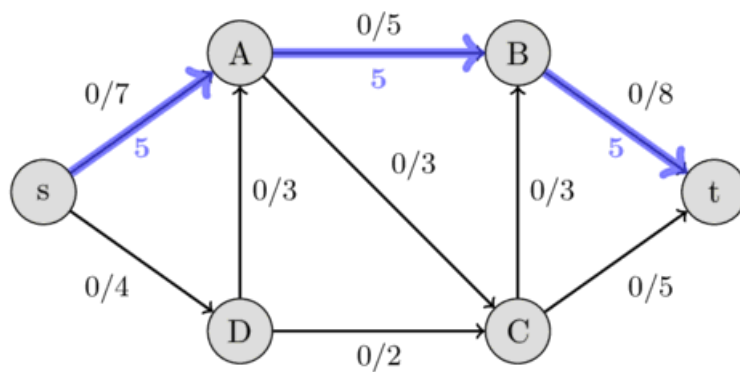
The Ford-Fulkerson method works as follows. First we set the flow of each edge to zero. Then we look for an **augmenting path** from s to t . An augmenting path is simple path in the residual graph, i.e. along the edges whose residual capacity is positive. If such a path is found, then we can add increase the flow along these edges. We keep on searching for augmenting paths and increasing the flow. Once there doesn't exist an augmenting path any more, the flow is maximal.

Let us specify in more detail, what increasing the flow along an augmenting path means. Let C be the smallest residual capacity of the edges in the path. Then we increase the flow in the following way: we update $f((u, v)) += C$ and $f((v, u)) -= C$ for every edge (u, v) in the path.

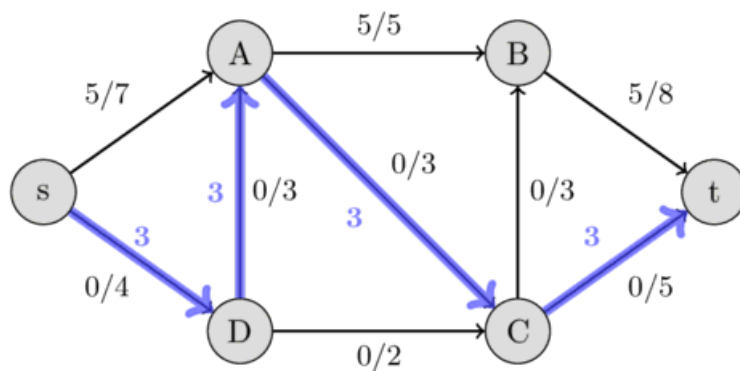
Here is an example to demonstrate the method. We use the same flow network as above. Initially we start with a flow of 0.

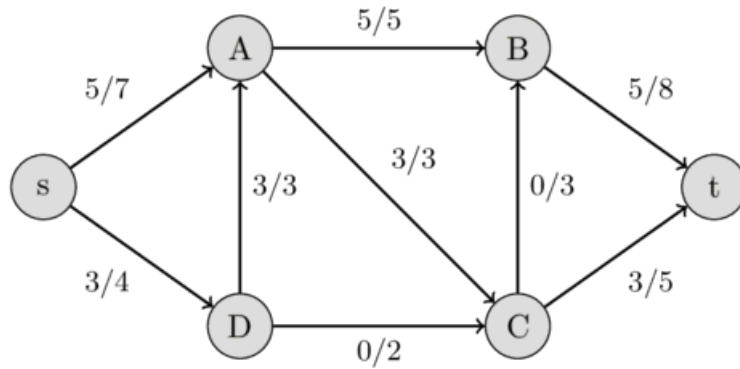


We can find the path $s - A - B - t$ with the residual capacities 7, 5 and 8. Their minimum is 5, therefore we can increase the flow along this path by 5. This gives a flow of 5 for the network.

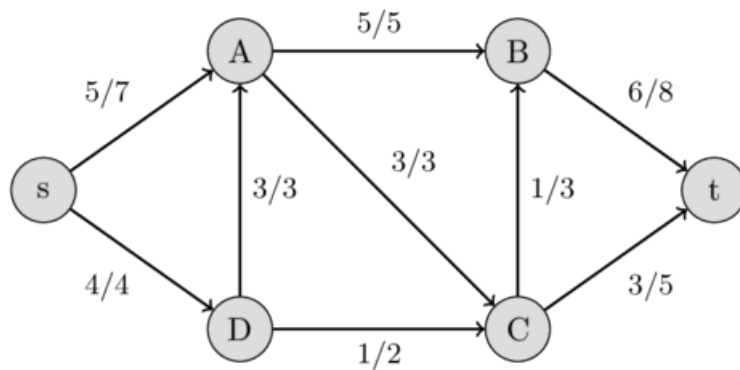
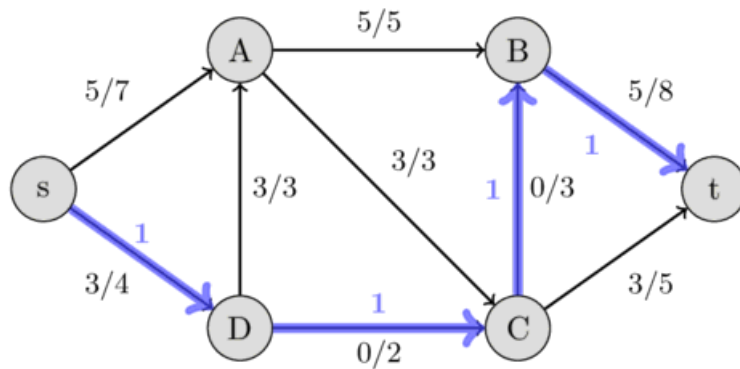


Again we look for an augmenting path, this time we find $s - D - A - C - t$ with the residual capacities 4, 3, 3 and 5. Therefore we can increase the flow by 3 and we get a flow of 8 for the network.



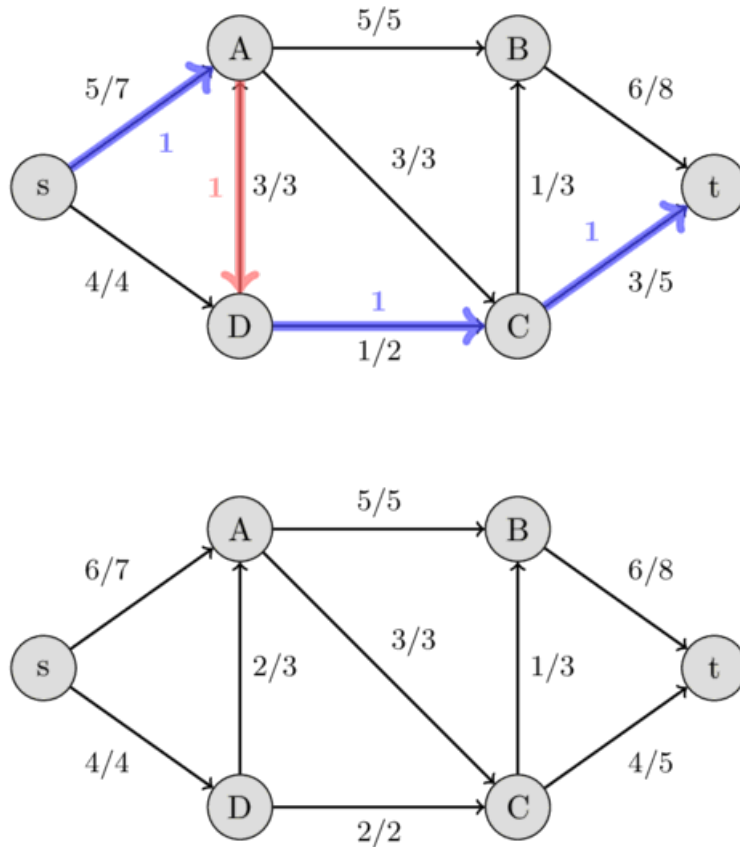


This time we find the path $s - D - C - B - t$ with the residual capacities 1, 2, 3 and 3, and we increase by 1.



This time we find the augmenting path $s - A - D - C - t$ with the residual capacities 2, 3, 1 and 2. We can increase by 1. But this path is very interesting. It includes the reversed edge (A, D) . In the original flow network we are not

allowed to send any flow from A to D . But because we already have a flow of 3 from D to A this is possible. The intuition of it is the following: Instead of sending a flow of 3 from D to A , we only send 2 and compensate this by sending an additional flow of 1 from s to A , which allows us to send an additional flow of 1 along the path $D - C - t$.



Now it is impossible to find an augmenting path between s and t , therefore this flow of 10 is the maximal possible. We have found the maximal flow.

It should be noted, that the Ford-Fulkerson method doesn't specify a method of finding the augmenting path. Possible approaches are using [DFS](#) or [BFS](#) which both work in $O(E)$. If all capacities of the network are integers, then for each augmenting path the flow of the network increases by at least 1 (for more details see [Integral flow theorem](#)). Therefore the complexity of Ford-Fulkerson is $O(EF)$, where F is the maximal flow of the network. In case of rational capacities, the algorithm will also terminate, but the complexity is not bounded. In case of irrational capacities, the algorithm might never terminate, and might not even converge to the maximal flow.

8.1.3 Edmonds-Karp algorithm

Edmonds-Karp algorithm is just an implementation of the Ford-Fulkerson method that uses [BFS](#) for finding augmenting paths. The algorithm was first published by Yefim Dinitz in 1970, and later independently published by Jack Edmonds and Richard Karp in 1972.

The complexity can be given independently of the maximal flow. The algorithm runs in $O(VE^2)$ time, even for irrational capacities. The intuition is, that every time we find an augmenting path one of the edges becomes saturated, and the distance from the edge to s will be longer, if it appears later again in an augmenting path. And the length of a simple path is bounded by V .

Implementation

The matrix `capacity` stores the capacity for every pair of vertices. `adj` is the adjacency list of the **undirected graph**, since we have also to use the reversed of directed edges when we are looking for augmenting paths.

The function `maxflow` will return the value of the maximal flow. During the algorithm the matrix `capacity` will actually store the residual capacity of the network. The value of the flow in each edge will actually not be stored, but it is easy to extend the implementation - by using an additional matrix - to also store the flow and return it.

```
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int>& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == t)
                    return new_flow;
                q.push({next, new_flow});
            }
        }
    }

    return 0;
}
```

```

}

int maxflow(int s, int t) {
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}

```

8.1.4 Integral flow theorem

The theorem simply says, that if every capacity in the network is integer, then also the flow in each edge will be integer in the maximal flow.

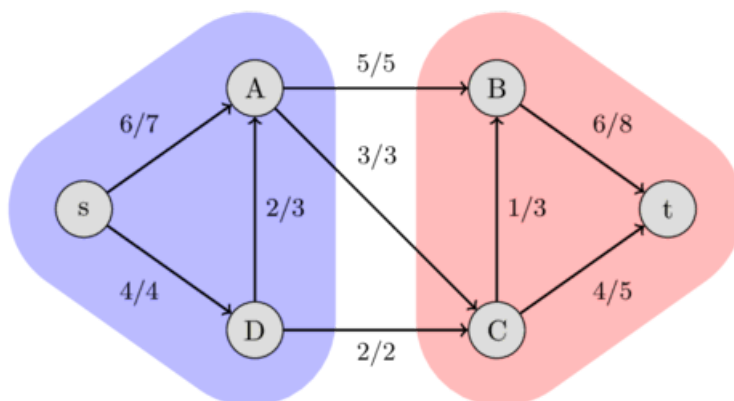
8.1.5 Max-flow min-cut theorem

A *s-t-cut* is a partition of the vertices of a flow network into two sets, such that a set includes the source s and the other one includes the sink t . The capacity of a *s-t-cut* is defined as the sum of capacities of the edges from the source side to the sink side.

Obviously we cannot send more flow from s to t than the capacity of any *s-t-cut*. Therefore the maximum flow is bounded by the minimum cut capacity.

The max-flow min-cut theorem goes even farther. It says that the capacity of the maximum flow has to be equal to the capacity of the minimum cut.

In the following image you can see the minimum cut of the flow network we used earlier. It shows that the capacity of the cut $\{s, A, D\}$ and $\{B, C, t\}$ is $5 + 3 + 2 = 10$, which is equal to the maximum flow that we found. Other cuts will have a bigger capacity, like the capacity between $\{s, A\}$ and $\{B, C, D, t\}$ is $4 + 3 + 5 = 12$.



A minimum cut can be found after performing a maximum flow computation using the Ford-Fulkerson method. One possible minimum cut is the following: the set of all vertices that can be reached from s in the residual graph (using edges with positive residual capacity), and the set of all the other vertices. This partition can be easily found using [DFS](#) starting at s .

8.2 Maximum flow - Push-relabel algorithm

The push-relabel algorithm (or also known as preflow-push algorithm) is an algorithm for computing the maximum flow of a flow network. The exact definition of the problem that we want to solve can be found in the article [Maximum flow - Ford-Fulkerson and Edmonds-Karp](#).

In this article we will consider solving the problem by pushing a preflow through the network, which will run in $O(V^4)$, or more precisely in $O(V^2E)$, time. The algorithm was designed by Andrew Goldberg and Robert Tarjan in 1985.

8.2.1 Definitions

During the algorithm we will have to handle a **preflow** - i.e. a function f that is similar to the flow function, but does not necessarily satisfies the flow conservation constraint. For it only the constraints

$$0 \leq f(e) \leq c(e)$$

and

$$\sum_{(v,u) \in E} f((v,u)) \geq \sum_{(u,v) \in E} f((u,v))$$

have to hold.

So it is possible for some vertex to receive more flow than it distributes. We say that this vertex has some excess flow, and define the amount of it with the **excess** function $x(u) = \sum_{(v,u) \in E} f((v,u)) - \sum_{(u,v) \in E} f((u,v))$.

In the same way as with the flow function, we can define the residual capacities and the residual graph with the preflow function.

The algorithm will start off with an initial preflow (some vertices having excess), and during the execution the preflow will be handled and modified. Giving away some details already, the algorithm will pick a vertex with excess, and push the excess to neighboring vertices. It will repeat this until all vertices, except the source and the sink, are free from excess. It is easy to see, that a preflow without excess is a valid flow. This makes the algorithm terminate with an actual flow.

There are still two problem, we have to deal with. First, how do we guarantee that this actually terminates? And secondly, how do we guarantee that this will actually give us a maximum flow, and not just any random flow?

To solve these problems we need the help of another function, namely the **labeling** functions h , often also called **height** function, which assigns each vertex an integer. We call a labeling is valid, if $h(s) = |V|$, $h(t) = 0$, and $h(u) \leq h(v) + 1$ if there is an edge (u,v) in the residual graph - i.e. the edge (u,v) has a positive capacity in the residual graph. In other words, if it is possible to increase the flow from u to v , then the height of v can be at most one smaller than the height of u , but it can be equal or even higher.

It is important to note, that if there exists a valid labeling function, then there doesn't exist an augmenting path from s to t in the residual graph. Because such

a path will have a length of at most $|V| - 1$ edges, and each edge can decrease the height only by at most by one, which is impossible if the first height is $h(s) = |V|$ and the last height is $h(t) = 0$.

Using this labeling function we can state the strategy of the push-relabel algorithm: We start with a valid preflow and a valid labeling function. In each step we push some excess between vertices, and update the labels of vertices. We have to make sure, that after each step the preflow and the labeling are still valid. If then the algorithm determines, the preflow is a valid flow. And because we also have a valid labeling, there doesn't exist a path between s and t in the residual graph, which means that the flow is actually a maximum flow.

If we compare the Ford-Fulkerson algorithm with the push-relabel algorithm it seems like the algorithms are the duals of each other. The Ford-Fulkerson algorithm keeps a valid flow at all time and improves it until there doesn't exist an augmenting path any more, while in the push-relabel algorithm there doesn't exist an augmenting path at any time, and we will improve the preflow until it is a valid flow.

8.2.2 Algorithm

First we have to initialize the graph with a valid preflow and labeling function.

Using the empty preflow - like it is done in the Ford-Fulkerson algorithm - is not possible, because then there will be an augmenting path and this implies that there doesn't exist a valid labeling. Therefore we will initialize each edges outgoing from s with it's maximal capacity: $f((s, u)) = c((s, u))$. And all other edges with zero. In this case there exists a valid labeling, namely $h(s) = |V|$ for the source vertex and $h(u) = 0$ for all other.

Now let's describe the two operations in more detail.

With the **push** operation we try to push as much excess flow from one vertex u to a neighboring vertex v . We have one rule: we are only allowed to push flow from u to v if $h(u) = h(v) + 1$. In layman's terms, the excess flow has to flow downwards, but not too steeply. Of course we only can push $\min(x(u), c((u, v)) - f((u, v)))$ flow.

If a vertex has excess, but it is not possible to push the excess to any adjacent vertex, then we need to increase the height of this vertex. We call this operation **relabel**. We will increase it by as much as it is possible, while still maintaining validity of the labeling.

To recap, the algorithm in a nutshell is: We initialize a valid preflow and a valid labeling. While we can perform push or relabel operations, we perform them. Afterwards the preflow is actually a flow and we return it.

8.2.3 Complexity

It is easy to show, that the maximal label of a vertex is $2|V| - 1$. At this point all remaining excess can and will be pushed back to the source. This gives at most $O(V^2)$ relabel operations.

It can also be showed, that there will be at most $O(VE)$ saturating pushes (a push where the total capacity of the edge is used) and at most $O(V^2E)$

non-saturating pushes (a push where the capacity of an edge is not fully used) performed. If we pick a data structure that allows us to find the next vertex with excess in $O(1)$ time, then the total complexity of the algorithm is $O(V^2E)$.

8.2.4 Implementation

```

const int inf = 1000000000;

int n;
vector<vector<int>> capacity, flow;
vector<int> height, excess, seen;
queue<int> excess_vertices;

void push(int u, int v)
{
    int d = min(excess[u], capacity[u][v] - flow[u][v]);
    flow[u][v] += d;
    flow[v][u] -= d;
    excess[u] -= d;
    excess[v] += d;
    if (d && excess[v] == d)
        excess_vertices.push(v);
}

void relabel(int u)
{
    int d = inf;
    for (int i = 0; i < n; i++) {
        if (capacity[u][i] - flow[u][i] > 0)
            d = min(d, height[i]);
    }
    if (d < inf)
        height[u] = d + 1;
}

void discharge(int u)
{
    while (excess[u] > 0) {
        if (seen[u] < n) {
            int v = seen[u];
            if (capacity[u][v] - flow[u][v] > 0 && height[u] > height[v])
                push(u, v);
            else
                seen[u]++;
        } else {
            relabel(u);
            seen[u] = 0;
        }
    }
}

int max_flow()

```

```

{
    height.assign(n, 0);
    height[0] = n;
    flow.assign(n, vector<int>(n, 0));
    excess.assign(n, 0);
    excess[0] = inf;
    for (int i = 1; i < n; i++)
        push(0, i);
    seen.assign(n, 0);

    while (!excess_vertices.empty()) {
        int u = excess_vertices.front();
        excess_vertices.pop();
        if (u != 0 && u != n - 1)
            discharge(u);
    }

    int max_flow = 0;
    for (int i = 0; i < n; i++)
        max_flow += flow[0][i];
    return max_flow;
}

```

Here we use the queue `excess_vertices` to store all vertices that currently have excess. In that way we can pick the next vertex for a push or a relabel operation in constant time.

And to make sure that we don't spend too much time finding the adjacent vertex to whom we can push, we use a data structure called **current-arc**. Basically we will iterate over the edges in a circular order and always store the last edge that we used. This way, for a certain labeling value, we will switch the current edge only $O(n)$ time. And since the relabeling already takes $O(n)$ time, we don't make the complexity worse.

8.3 Maximum flow - Push-relabel method improved

We will modify the [push-relabel method](#) to achieve a better runtime.

8.3.1 Description

The modification is extremely simple: In the previous article we chosen a vertex with excess without any particular rule. But it turns out, that if we always choose the vertices with the **greatest height**, and apply push and relabel operations on them, then the complexity will become better. Moreover, to select the vertices with the greatest height we actually don't need any data structures, we simply store the vertices with the greatest height in a list, and recalculate the list once all of them are processed (then vertices with already lower height will be added to the list), or whenever a new vertex with excess and a greater height appears (after relabeling a vertex).

Despite the simplicity, this modification reduces the complexity by a lot. To be precise, the complexity of the resulting algorithm is $O(VE + V^2\sqrt{E})$, which in the worst case is $O(V^3)$.

This modification was proposed by Cheriyan and Maheshwari in 1989.

8.3.2 Implementation

```
const int inf = 1000000000;

int n;
vector<vector<int>> capacity, flow;
vector<int> height, excess;

void push(int u, int v)
{
    int d = min(excess[u], capacity[u][v] - flow[u][v]);
    flow[u][v] += d;
    flow[v][u] -= d;
    excess[u] -= d;
    excess[v] += d;
}

void relabel(int u)
{
    int d = inf;
    for (int i = 0; i < n; i++) {
        if (capacity[u][i] - flow[u][i] > 0)
            d = min(d, height[i]);
    }
    if (d < inf)
        height[u] = d + 1;
}

vector<int> find_max_height_vertices(int s, int t) {
    vector<int> max_height;
```



```

    for (int i = 0; i < n; i++) {
        if (i != s && i != t && excess[i] > 0) {
            if (!max_height.empty() && height[i] > height[max_height[0]])
                max_height.clear();
            if (max_height.empty() || height[i] == height[max_height[0]])
                max_height.push_back(i);
        }
    }
    return max_height;
}

int max_flow(int s, int t)
{
    height.assign(n, 0);
    height[s] = n;
    flow.assign(n, vector<int>(n, 0));
    excess.assign(n, 0);
    excess[s] = inf;
    for (int i = 0; i < n; i++) {
        if (i != s)
            push(s, i);
    }

    vector<int> current;
    while (!(current = find_max_height_vertices(s, t)).empty()) {
        for (int i : current) {
            bool pushed = false;
            for (int j = 0; j < n && excess[i]; j++) {
                if (capacity[i][j] - flow[i][j] > 0 && height[i] == height[j] + 1) {
                    push(i, j);
                    pushed = true;
                }
            }
            if (!pushed) {
                relabel(i);
                break;
            }
        }
    }

    int max_flow = 0;
    for (int i = 0; i < n; i++)
        max_flow += flow[0][i];
    return max_flow;
}

```

8.4 Maximum flow - Dinic's algorithm

Dinic's algorithm solves the maximum flow problem in $O(V^2E)$. The maximum flow problem is defined in this article [Maximum flow - Ford-Fulkerson and Edmonds-Karp](#). This algorithm was discovered by Yefim Dinitz in 1970.

8.4.1 Definitions

A **residual network** G^R of network G is a network which contains two edges for each edge $(v, u) \in G$:

- (v, u) with capacity $c_{vu}^R = c_{vu} - f_{vu}$
- (u, v) with capacity $c_{uv}^R = f_{vu}$

A **blocking flow** of some network is such a flow that every path from s to t contains at least one edge which is saturated by this flow. Note that a blocking flow is not necessarily maximal.

A **layered network** of a network G is a network built in the following way. Firstly, for each vertex v we calculate $level[v]$ - the shortest path (unweighted) from s to this vertex using only edges with positive capacity. Then we keep only those edges (v, u) for which $level[v] + 1 = level[u]$. Obviously, this network is acyclic.

8.4.2 Algorithm

The algorithm consists of several phases. On each phase we construct the layered network of the residual network of G . Then we find an arbitrary blocking flow in the layered network and add it to the current flow.

8.4.3 Proof of correctness

Let's show that if the algorithm terminates, it finds the maximum flow.

If the algorithm terminated, it couldn't find a blocking flow in the layered network. It means that the layered network doesn't have any path from s to t . It means that the residual network doesn't have any path from s to t . It means that the flow is maximum.

8.4.4 Number of phases

The algorithm terminates in less than V phases. To prove this, we must firstly prove two lemmas.

Lemma 1. The distances from s to each vertex don't decrease after each iteration, i. e. $level_{i+1}[v] \geq level_i[v]$.

Proof. Fix a phase i and a vertex v . Consider any shortest path P from s to v in G_{i+1}^R . The length of P equals $level_{i+1}[v]$. Note that G_{i+1}^R can only contain edges from G_i^R and back edges for edges from G_i^R . If P has no back edges for G_i^R , then $level_{i+1}[v] \geq level_i[v]$ because P is also a path in G_i^R . Now, suppose that P has at least one back edge. Let the first such edge be (u, w) . Then $level_{i+1}[u] \geq level_i[u]$

(because of the first case). The edge (u, w) doesn't belong to G_i^R , so the edge (w, u) was affected by the blocking flow on the previous iteration. It means that $level_i[u] = level_i[w] + 1$. Also, $level_{i+1}[w] = level_{i+1}[u] + 1$. From these two equations and $level_{i+1}[u] \geq level_i[u]$ we obtain $level_{i+1}[w] \geq level_i[w] + 2$. Now we can use the same idea for the rest of the path.

Lemma 2. $level_{i+1}[t] > level_i[t]$

Proof. From the previous lemma, $level_{i+1}[t] \geq level_i[t]$. Suppose that $level_{i+1}[t] = level_i[t]$. Note that G_{i+1}^R can only contain edges from G_i^R and back edges for edges from G_i^R . It means that there is a shortest path in G_i^R which wasn't blocked by the blocking flow. It's a contradiction.

From these two lemmas we conclude that there are less than V phases because $level[t]$ increases, but it can't be greater than $V - 1$.

8.4.5 Finding blocking flow

In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS from s to t in the layered network while it can be pushed. In order to do it more quickly, we must remove the edges which can't be used to push anymore. To do this we can keep a pointer in each vertex which points to the next edge which can be used. Each pointer can be moved at most E times, so each phase works in $O(VE)$.

8.4.6 Complexity

There are less than V phases, so the total complexity is $O(V^2E)$.

8.4.7 Unit networks

A **unit network** is a network in which all the edges have unit capacity, and for any vertex except s and t either incoming or outgoing edge is unique. That's exactly the case with the network we build to solve the maximum matching problem with flows.

On unit networks Dinic's algorithm works in $O(E\sqrt{V})$. Let's prove this.

Firstly, each phase now works in $O(E)$ because each edge will be considered at most once.

Secondly, suppose there have already been \sqrt{V} phases. Then all the augmenting paths with the length $\leq \sqrt{V}$ have been found. Let f be the current flow, f' be the maximum flow. Consider their difference $f' - f$. It is a flow in G^R of value $|f'| - |f|$ and on each edge it is either 0 or 1. It can be decomposed into $|f'| - |f|$ paths from s to t and possibly cycles. As the network is unit, they can't have common vertices, so the total number of vertices is $\geq (|f'| - |f|)\sqrt{V}$, but it is also $\leq V$, so in another \sqrt{V} iterations we will definitely find the maximum flow.

8.4.8 Implementation

```
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
```

```

FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, long long cap) {
        edges.push_back(FlowEdge(v, u, cap));
        edges.push_back(FlowEdge(u, v, 0));
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
                continue;

```

```
        long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
        if (tr == 0)
            continue;
        edges[id].flow += tr;
        edges[id ^ 1].flow -= tr;
        return tr;
    }
    return 0;
}

long long flow() {
    long long f = 0;
    while (true) {
        fill(level.begin(), level.end(), -1);
        level[s] = 0;
        q.push(s);
        if (!bfs())
            break;
        fill(ptr.begin(), ptr.end(), 0);
        while (long long pushed = dfs(s, flow_inf)) {
            f += pushed;
        }
    }
    return f;
}

};
```

8.5 Maximum flow - MPM algorithm

MPM (Malhotra, Pramodh-Kumar and Maheshwari) algorithm solves the maximum flow problem in $O(V^3)$. This algorithm is similar to [Dinic's algorithm](#).

8.5.1 Algorithm

Like Dinic's algorithm, MPM runs in phases, during each phase we find the blocking flow in the layered network of the residual network of G . The main difference from Dinic's is how we find the blocking flow. Consider the layered network L . For each node we define its' *inner potential* and *outer potential* as:

$$p_{in}(v) = \sum_{(u,v) \in L} (c(u,v) - f(u,v))$$

$$p_{out}(v) = \sum_{(v,u) \in L} (c(v,u) - f(v,u))$$

Also we set $p_{in}(s) = p_{out}(t) = \infty$. Given p_{in} and p_{out} we define the *potential* as $p(v) = \min(p_{in}(v), p_{out}(v))$. We call a node r a *reference node* if $p(r) = \min\{p(v)\}$. Consider a reference node r . We claim that the flow can be increased by $p(r)$ in such a way that $p(r)$ becomes 0. It is true because L is acyclic, so we can push the flow out of r by outgoing edges and it will reach t because each node has enough outer potential to push the flow out when it reaches it. Similarly, we can pull the flow from s . The construction of the blocked flow is based on this fact. On each iteration we find a reference node and push the flow from s to t through r . This process can be simulated by BFS. All completely saturated arcs can be deleted from L as they won't be used later in this phase anyway. Likewise, all the nodes different from s and t without outgoing or incoming arcs can be deleted.

Each phase works in $O(V^2)$ because there are at most V iterations (because at least the chosen reference node is deleted), and on each iteration we delete all the edges we passed through except at most V . Summing, we get $O(V^2 + E) = O(V^2)$. Since there are less than V phases (see the proof [here](#)), MPM works in $O(V^3)$ total.

8.5.2 Implementation

```
struct MPM{
    struct FlowEdge{
        int v, u;
        long long cap, flow;
        FlowEdge(){}
        FlowEdge(int _v, int _u, long long _cap, long long _flow)
            : v(_v), u(_u), cap(_cap), flow(_flow){}
        FlowEdge(int _v, int _u, long long _cap)
            : v(_v), u(_u), cap(_cap), flow(0){}
    };
};
```

```

const long long flow_inf = 1e18;
vector<FlowEdge> edges;
vector<char> alive;
vector<long long> pin, pout;
vector<list<int> > in, out;
vector<vector<int> > adj;
vector<long long> ex;
int n, m = 0;
int s, t;
vector<int> level;
vector<int> q;
int qh, qt;
void resize(int _n){
    n = _n;
    ex.resize(n);
    q.resize(n);
    pin.resize(n);
    pout.resize(n);
    adj.resize(n);
    level.resize(n);
    in.resize(n);
    out.resize(n);
}
MPM(){}
MPM(int _n, int _s, int _t){resize(_n); s = _s; t = _t;}
void add_edge(int v, int u, long long cap){
    edges.push_back(FlowEdge(v, u, cap));
    edges.push_back(FlowEdge(u, v, 0));
    adj[v].push_back(m);
    adj[u].push_back(m + 1);
    m += 2;
}
bool bfs(){
    while(qh < qt){
        int v = q[qh++];
        for(int id : adj[v]){
            if(edges[id].cap - edges[id].flow < 1)continue;
            if(level[edges[id].u] != -1)continue;
            level[edges[id].u] = level[v] + 1;
            q[qt++] = edges[id].u;
        }
    }
    return level[t] != -1;
}
long long pot(int v){
    return min(pin[v], pout[v]);
}
void remove_node(int v){
    for(int i : in[v]){
        int u = edges[i].v;
        auto it = find(out[u].begin(), out[u].end(), i);
        out[u].erase(it);
    }
}

```

```

        pout[u] -= edges[i].cap - edges[i].flow;
    }
    for(int i : out[v]){
        int u = edges[i].u;
        auto it = find(in[u].begin(), in[u].end(), i);
        in[u].erase(it);
        pin[u] -= edges[i].cap - edges[i].flow;
    }
}

void push(int from, int to, long long f, bool forw){
    qh = qt = 0;
    ex.assign(n, 0);
    ex[from] = f;
    q[qt++] = from;
    while(qh < qt){
        int v = q[qh++];
        if(v == to)
            break;
        long long must = ex[v];
        auto it = forw ? out[v].begin() : in[v].begin();
        while(true){
            int u = forw ? edges[*it].u : edges[*it].v;
            long long pushed = min(must, edges[*it].cap - edges[*it].flow);
            if(pushed == 0) break;
            if(forw){
                pout[v] -= pushed;
                pin[u] -= pushed;
            }
            else{
                pin[v] -= pushed;
                pout[u] -= pushed;
            }
            if(ex[u] == 0)
                q[qt++] = u;
            ex[u] += pushed;
            edges[*it].flow += pushed;
            edges[(*it)^1].flow -= pushed;
            must -= pushed;
            if(edges[*it].cap - edges[*it].flow == 0){
                auto jt = it;
                ++jt;
                if(forw){
                    in[u].erase(find(in[u].begin(), in[u].end(), *it));
                    out[v].erase(it);
                }
                else{
                    out[u].erase(find(out[u].begin(), out[u].end(), *it));
                    in[v].erase(it);
                }
                it = jt;
            }
        }
        else break;
    }
}

```



```

        if(!must)break;
    }
}
}
long long flow(){
    long long ans = 0;
    while(true){
        pin.assign(n, 0);
        pout.assign(n, 0);
        level.assign(n, -1);
        alive.assign(n, true);
        level[s] = 0;
        qh = 0; qt = 1;
        q[0] = s;
        if(!bfs())
            break;
        for(int i = 0; i < n; i++){
            out[i].clear();
            in[i].clear();
        }
        for(int i = 0; i < m; i++){
            if(edges[i].cap - edges[i].flow == 0)
                continue;
            int v = edges[i].v, u = edges[i].u;
            if(level[v] + 1 == level[u] && (level[u] < level[t] || u == t)){
                in[u].push_back(i);
                out[v].push_back(i);
                pin[u] += edges[i].cap - edges[i].flow;
                pout[v] += edges[i].cap - edges[i].flow;
            }
        }
        pin[s] = pout[t] = flow_inf;
        while(true){
            int v = -1;
            for(int i = 0; i < n; i++){
                if(!alive[i])continue;
                if(v == -1 || pot(i) < pot(v))
                    v = i;
            }
            if(v == -1)
                break;
            if(pot(v) == 0){
                alive[v] = false;
                remove_node(v);
                continue;
            }
            long long f = pot(v);
            ans += f;
            push(v, s, f, false);
            push(v, t, f, true);
            alive[v] = false;
            remove_node(v);
        }
    }
}

```

```
        }  
    }  
    return ans;  
}  
};
```

8.6 Flows with demands

In a normal flow network the flow of an edge is only limited by the capacity $c(e)$ from above and by 0 from below. In this article we will discuss flow networks, where we additionally require the flow of each edge to have a certain amount, i.e. we bound the flow from below by a **demand** function $d(e)$:

$$d(e) \leq f(e) \leq c(e)$$

So next each edge has a minimal flow value, that we have to pass along the edge.

This is a generalization of the normal flow problem, since setting $d(e) = 0$ for all edges e gives a normal flow network. Notice, that in the normal flow network it is extremely trivial to find a valid flow, just setting $f(e) = 0$ is already a valid one. However if the flow of each edge has to satisfy a demand, than suddenly finding a valid flow is already pretty complicated.

We will consider two problems:

1. finding an arbitrary flow that satisfies all constraints
2. finding a minimal flow that satisfies all constraints

8.6.1 Finding an arbitrary flow

We make the following changes in the network. We add a new source s' and a new sink t' , a new edge from the source s' to every other vertex, a new edge for every vertex to the sink t' , and one edge from t to s . Additionally we define the new capacity function c' as:

- $c'((s', v)) = \sum_{u \in V} d((u, v))$ for each edge (s', v) .
- $c'((v, t')) = \sum_{w \in V} d((v, w))$ for each edge (v, t') .
- $c'((u, v)) = c((u, v)) - d((u, v))$ for each edge (u, v) in the old network.
- $c'((t, s)) = \infty$

If the new network has a saturating flow (a flow where each edge outgoing from s' is completely filled, which is equivalent to every edge incoming to t' is completely filled), then the network with demands has a valid flow, and the actual flow can be easily reconstructed from the new network. Otherwise there doesn't exist a flow that satisfies all conditions. Since a saturating flow has to be a maximum flow, it can be found by any maximum flow algorithm, like the [Edmonds-Karp algorithm](#) or the [Push-relabel algorithm](#).

The correctness of these transformations is more difficult to understand. We can think of it in the following way: Each edge $e = (u, v)$ with $d(e) > 0$ is originally replaced by two edges: one with the capacity $d(i)$, and the other with $c(i) - d(i)$. We want to find a flow that saturates the first edge (i.e. the flow along this edge must be equal to its capacity). The second edge is less important - the flow along it can be anything, assuming that it doesn't exceed its capacity. Consider each edge that has to be saturated, and we perform the following operation: we draw the edge from the new source s' to its end v , draw the edge from its start u to the new sink t' , remove the edge itself, and from

the old sink t to the old source s we draw an edge of infinite capacity. By these actions we simulate the fact that this edge is saturated - from v there will be an additionally $d(e)$ flow outgoing (we simulate it with a new source that feeds the right amount of flow to v), and u will also push $d(e)$ additional flow (but instead along the old edge, this flow will go directly to the new sink t'). A flow with the value $d(e)$, that originally flowed along the path $s - \dots - u - v - \dots - t$ can now take the new path $s' - v - \dots - t - s - \dots - u - t'$. The only thing that got simplified in the definition of the new network, is that if procedure created multiple edges between the same pair of vertices, then they are combined to one single edge with the summed capacity.

8.6.2 Minimal flow

Note that along the edge (t, s) (from the old sink to the old source) with the capacity ∞ flows the entire flow of the corresponding old network. I.e. the capacity of this edge effects the flow value of the old network. By giving this edge a sufficient large capacity (i.e. ∞), the flow of the old network is unlimited. By limiting this edge by smaller capacities, the flow value will decrease. However if we limit this edge by a too small value, than the network will not have a saturated solution, e.g. the corresponding solution for the original network will not satisfy the demand of the edges. Obviously here can use a binary search to find the lowest value with which all constraints are still satisfied. This gives the minimal flow of the original network.

8.7 Minimum-cost flow - Successive shortest path algorithm

Given a network G consisting of n vertices and m edges. For each edge (generally speaking, oriented edges, but see below), the capacity (a non-negative integer) and the cost per unit of flow along this edge (some integer) are given. Also the source s and the sink t are marked.

For a given value K , we have to find a flow of this quantity, and among all flows of this quantity we have to choose the flow with the lowest cost. This task is called **minimum-cost flow problem**.

Sometimes the task is given a little differently: you want to find the maximum flow, and among all maximal flows we want to find the one with the least cost. This is called the **minimum-cost maximum-flow problem**.

Both these problems can be solved effectively with the algorithm of successive shortest paths.

8.7.1 Algorithm

This algorithm is very similar to the [Edmonds-Karp](#) for computing the maximum flow.

Simplest case

First we only consider the simplest case, where the graph is oriented, and there is at most one edge between any pair of vertices (e.g. if (i, j) is an edge in the graph, then (j, i) cannot be part in it as well).

Let U_{ij} be the capacity of an edge (i, j) if this edge exists. And let C_{ij} be the cost per unit of flow along this edge (i, j) . And finally let $F_{i,j}$ be the flow along the edge (i, j) . Initially all flow values are zero.

We **modify** the network as follows: for each edge (i, j) we add the **reverse edge** (j, i) to the network with the capacity $U_{ji} = 0$ and the cost $C_{ji} = -C_{ij}$. Since, according to our restrictions, the edge (j, i) was not in the network before, we still have a network that is not a multigraph (graph with multiple edges). In addition we will always keep the condition $F_{ji} = -F_{ij}$ true during the steps of the algorithm.

We define the **residual network** for some fixed flow F as follow (just like in the Ford-Fulkerson algorithm): the residual network contains only unsaturated edges (i.e. edges in which $F_{ij} < U_{ij}$), and the residual capacity of each such edge is $R_{ij} = U_{ij} - F_{ij}$.

Now we can talk about the **algorithms** to compute the minimum-cost flow. At each iteration of the algorithm we find the shortest path in the residual graph from s to t . In contrary to Edmonds-Karp we look for the shortest path in terms of the cost of the path, instead of the number of edges. If there doesn't exist a path anymore, then the algorithm terminates, and the stream F is the desired one. If a path was found, we increase the flow along it as much as possible (i.e. we find the minimal residual capacity R of the path, and increase the flow by it, and

reduce the back edges by the same amount). If at some point the flow reaches the value K , then we stop the algorithm (note that in the last iteration of the algorithm it is necessary to increase the flow by only such an amount so that the final flow value doesn't surpass K).

It is not difficult to see, that if we set K to infinity, then the algorithm will find the minimum-cost maximum-flow. So both variations of the problem can be solved by the same algorithm.

Undirected graphs / multigraphs

The case of an undirected graph or a multigraph doesn't differ conceptually from the algorithm above. The algorithm will also work on these graphs. However it becomes a little more difficult to implement it.

An **undirected edge** (i, j) is actually the same as two oriented edges (i, j) and (j, i) with the same capacity and values. Since the above-described minimum-cost flow algorithm generates a back edge for each directed edge, so it splits the undirected edge into 4 directed edges, and we actually get a **multigraph**.

How do we deal with **multiple edges**? First the flow for each of the multiple edges must be kept separately. Secondly, when searching for the shortest path, it is necessary to take into account that it is important which of the multiple edges is used in the path. Thus instead of the usual ancestor array we additionally must store the edge number from which we came from along with the ancestor. Thirdly, as the flow increases along a certain edge, it is necessary to reduce the flow along the back edge. Since we have multiple edges, we have to store the edge number for the reversed edge for each edge.

There are no other obstructions with undirected graphs or multigraphs.

Complexity

Analog to the analysis of the Edmonds-Karp algorithm we get the following estimation: $O(nm) \cdot T(n, m)$, where $T(n, m)$ is the time required to find the shortest path in a graph with n vertices and m edges

If this search is done with the [Dijkstra algorithm](#), then the complexity for the minimum-cost algorithm would become $O(n^3m)$. However we deal with edges with negative cost. So Dijkstra is not applicable, at least not unmodified.

Instead we can use the [Bellman-Ford algorithm](#). With it the complexity becomes $O(n^2m^2)$.

Instead, we can also use the [D'Esopo-Pape algorithm](#), which has a much worse complexity, but will run very quickly in practice (in about the same time as Dijkstra's algorithm).

8.7.2 Implementation

Here is an implementation using the [D'Esopo-Pape algorithm](#) for the simplest case.

```
struct Edge
{
```

```

    int from, to, capacity, cost;
};

vector<vector<int>> adj, cost, capacity;

const int INF = 1e9;

void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p) {
    d.assign(n, INF);
    d[v0] = 0;
    vector<int> m(n, 2);
    deque<int> q;
    q.push_back(v0);
    p.assign(n, -1);

    while (!q.empty()) {
        int u = q.front();
        q.pop_front();
        m[u] = 0;
        for (int v : adj[u]) {
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (m[v] == 2) {
                    m[v] = 1;
                    q.push_back(v);
                } else if (m[v] == 0) {
                    m[v] = 1;
                    q.push_front(v);
                }
            }
        }
    }
}

int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }

    int flow = 0;
    int cost = 0;
    vector<int> d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
    }
}

```

```
    if (d[t] == INF)
        break;

    // find max flow on that path
    int f = K - flow;
    int cur = t;
    while (cur != s) {
        f = min(f, capacity[p[cur]][cur]);
        cur = p[cur];
    }

    // apply flow
    flow += f;
    cost += f * d[t];
    cur = t;
    while (cur != s) {
        capacity[p[cur]][cur] -= f;
        capacity[cur][p[cur]] += f;
        cur = p[cur];
    }
}

if (flow < K)
    return -1;
else
    return cost;
}
```


8.8 Solving assignment problem using min-cost-flow

The **assignment problem** has two equivalent statements:

- Given a square matrix $A[1..N, 1..N]$, you need to select N elements in it so that exactly one element is selected in each row and column, and the sum of the values of these elements is the smallest.
- There are N orders and N machines. The cost of manufacturing on each machine is known for each order. Only one order can be performed on each machine. It is required to assign all orders to the machines so that the total cost is minimized.

Here we will consider the solution of the problem based on the algorithm for finding the **minimum cost flow (min-cost-flow)**, solving the assignment problem in $\mathcal{O}(N^5)$.

8.8.1 Description

Let's build a bipartite network: there is a source S , a drain T , in the first part there are N vertices (corresponding to rows of the matrix, or orders), in the second there are also N vertices (corresponding to the columns of the matrix, or machines). Between each vertex i of the first set and each vertex j of the second set, we draw an edge with bandwidth 1 and cost A_{ij} . From the source S we draw edges to all vertices i of the first set with bandwidth 1 and cost 0. We draw an edge with bandwidth 1 and cost 0 from each vertex of the second set j to the drain T .

We find in the resulting network the maximum flow of the minimum cost. Obviously, the value of the flow will be N . Further, for each vertex i of the first segment there is exactly one vertex j of the second segment, such that the flow $F_{ij} = 1$. Finally, this is a one-to-one correspondence between the vertices of the first segment and the vertices of the second part, which is the solution to the problem (since the found flow has a minimal cost, then the sum of the costs of the selected edges will be the lowest possible, which is the optimality criterion).

The complexity of this solution of the assignment problem depends on the algorithm by which the search for the maximum flow of the minimum cost is performed. The complexity will be $\mathcal{O}(N^5)$ using **Dijkstra** or $\mathcal{O}(N^6)$ using **Bellman-Ford**.

8.8.2 Implementation

The implementation given here is long, it can probably be significantly reduced. It uses the **D'Esopo-Pape algorithm** for finding shortest paths.

```
const int INF = 1000 * 1000 * 1000;

vector<int> assignment(vector<vector<int>>> a) {
    int n = a.size();
    int m = n * 2 + 2;
```

```

vector<vector<int>> f(m, vector<int>(m));
int s = m - 2, t = m - 1;
int cost = 0;
while (true) {
    vector<int> dist(m, INF);
    vector<int> p(m);
    vector<int> type(m, 2);
    deque<int> q;
    dist[s] = 0;
    p[s] = -1;
    type[s] = 1;
    q.push_back(s);
    while (!q.empty()) {
        int v = q.front();
        q.pop_front();
        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 cur = t; cur != -1; cur = p[cur]) {
        int prev = p[cur];
        if (prev != -1)
            f[cur][prev] = -(f[prev][cur] = 1);
    }
}

vector<int> answer(n);
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (f[i][j + n] == 1)
            answer[i] = j;
    }
}
return answer;
}
```

Chapter 9

Matchings and related problems

9.1 Check whether a graph is bipartite

A bipartite graph is a graph whose vertices can be divided into two disjoint sets so that every edge connects two vertices from different sets (i.e. there are no edges which connect vertices from the same set). These sets are usually called sides.

You are given an undirected graph. Check whether it is bipartite, and if it is, output its sides.

9.1.1 Algorithm

There exists a theorem which claims that a graph is bipartite if and only if all its cycles have even length. However, in practice it's more convenient to use a different formulation of the definition: a graph is bipartite if and only if it is two-colorable.

Let's use a series of [breadth-first searches](#), starting from each vertex which hasn't been visited yet. In each search, assign the vertex from which we start to side 1. Each time we visit a yet unvisited neighbor of a vertex assigned to one side, we assign it to the other side. When we try to go to a neighbor of a vertex assigned to one side which has already been visited, we check that it has been assigned to the other side; if it has been assigned to the same side, we conclude that the graph is not bipartite. Once we've visited all vertices and successfully assigned them to sides, we know that the graph is bipartite and we have constructed its partitioning.

9.1.2 Implementation

```
int n;
vector<vector<int>> adj;

vector<int> side(n, -1);
bool is_bipartite = true;
```

```
queue<int> q;
for (int st = 0; st < n; ++st) {
    if (side[st] == -1) {
        q.push(st);
        side[st] = 0;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int u : adj[v]) {
                if (side[u] == -1) {
                    side[u] = side[v] ^ 1;
                    q.push(u);
                } else {
                    is_bipartite &= side[u] != side[v];
                }
            }
        }
    }
}

cout << (is_bipartite ? "YES" : "NO") << endl;
```

Practice problems:

- [SPOJ - BUGLIFE](#)

Chapter 10

Miscellaneous

10.1 Topological Sorting

You are given a directed graph with n vertices and m edges. You have to **number the vertices** so that every edge leads from the vertex with a smaller number assigned to the vertex with a larger one.

In other words, you want to find a permutation of the vertices (**topological order**) which corresponds to the order defined by all edges of the graph.

Topological order can be **non-unique** (for example, if the graph is empty; or if there exist three vertices a, b, c for which there exist paths from a to b and from a to c but not paths from b to c or from c to b).

Topological order may **not exist** at all if the graph contains cycles (because there is a contradiction: there is a path from a to b and vice versa).

A common problem in which topological sorting occurs is the following. There are n variables with unknown values. For some variables we know that one of them is less than the other. You have to check whether these constraints are contradictory, and if not, output the variables in ascending order (if several answers are possible, output any of them). It is easy to notice that this is exactly the problem of finding topological order of a graph with n vertices.

10.1.1 The Algorithm

To solve this problem we will use [depth-first search](#).

Let's assume that the graph is acyclic, i.e. there is a solution. What does the depth-first search do? When started from some vertex v , it tries to run along all edges outgoing from v . It fails to run along the edges for which the opposite ends have been visited previously, and runs along the rest of the edges and starts from their ends.

Thus, by the time of the call $dfs(v)$ is ended, all vertices that are reachable from v either directly (via one edge) or indirectly are already visited by the search. Therefore, if at the time of exit from $dfs(v)$ we add vertex v to the beginning of a certain list, in the end this list will store a topological ordering of all vertices.

These explanations can also be presented in terms of time of exit from DFS routine. Exit time for vertex v is the time at which $dfs(v)$ finished work (the

times can be numbered from 1 to n). It is easy to understand that exit time of any vertex v is always greater than exit time of any vertex reachable from it (since they were visited either before the call $dfs(v)$ or during it). Thus, the desired topological ordering is sorting vertices in descending order of their exit times.

10.1.2 Implementation

Here is an implementation which assumes that the graph is acyclic, i.e. the desired topological ordering exists. If necessary, you can easily check that the graph is acyclic, as described in the article on [depth-first search](#).

C++ implementation

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
    reverse(ans.begin(), ans.end());
}
```

The main function of the solution is `topological_sort`, which initializes DFS variables, launches DFS and receives the answer in the vector `ans`.

10.1.3 Practice Problems

- [SPOJ TOPOSORT](#) “Topological Sorting” [difficulty: easy]
- [UVA #10305](#) “Ordering Tasks” [difficulty: easy]
- [UVA #124](#) “Following Orders” [difficulty: easy]
- [UVA #200](#) “Rare Order” [difficulty: easy]
- [Codeforces 510C](#) “Fox and Names” [difficulty: easy]

10.2 Edge connectivity / Vertex connectivity

10.2.1 Definition

Given an undirected graph G with n vertices and m edges. Both the edge connectivity and the vertex connectivity are characteristics describing the graph.

Edge connectivity

The **edge connectivity** λ of the graph G is the minimum number of edges that need to be deleted, such that the graph G gets disconnected.

For example an already disconnected graph has an edge connectivity of 0, a connected graph with at least one bridge has an edge connectivity of 1, and a connected graph with no bridges has an edge connectivity of at least 2.

We say that a set S of edges **separates** the vertices s and t , if, after removing all edges in S from the graph G , the vertices s and t end up in different connected components.

It is clear, that the edge connectivity of a graph is equal to the minimum size of such a set separating two vertices s and t , taken among all possible pairs (s, t) .

Vertex connectivity

The **vertex connectivity** κ of the graph G is the minimum number of vertices that need to be deleted, such that the graph G gets disconnected.

For example an already disconnected graph has the vertex connectivity 0, and a connected graph with an articulation point has the vertex connectivity 1. We define that a complete graph has the vertex connectivity $n - 1$. For all other graphs the vertex connectivity doesn't exceed $n - 2$, because you can find a pair of vertices which are not connected by an edge, and remove all other $n - 2$ vertices.

We say that a set T of vertices **separates** the vertices s and t , if, after removing all vertices in T from the graph G , the vertices end up in different connected components.

It is clear, that the vertex connectivity of a graph is equal to the minimal size of such a set separating two vertices s and t , taken among all possible pairs (s, t) .

10.2.2 Properties

The Whitney inequalities

The **Whitney inequalities** (1932) gives a relation between the edge connectivity λ , the vertex connectivity κ and the smallest degree of the vertices δ :

$$\kappa \leq \lambda \leq \delta$$

Intuitively if we have a set of edges of size λ , which make the graph disconnected, we can choose one of each end point, and create a set of vertices, that also disconnect the graph. And this set has size $\leq \lambda$.

And if we pick the vertex and the minimal degree δ , and remove all edges connected to it, then we also end up with a disconnected graph. Therefore the second inequality $\lambda \leq \delta$.

It is interesting to note, that the Whitney inequalities cannot be improved: i.e. for any triple of numbers satisfying this inequality there exists at least one corresponding graph. One such graph can be constructed in the following way: The graph will consist of $2(\delta + 1)$ vertices, the first $\delta + 1$ vertices form a clique (all pairs of vertices are connected via an edge), and the second $\delta + 1$ vertices form a second clique. In addition we connect the two cliques with λ edges, such that it uses λ different vertices in the first clique, and only κ vertices in the second clique. The resulting graph will have the three characteristics.

The Ford-Fulkerson theorem

The **Ford-Fulkerson theorem** implies, that the biggest number of edge-disjoint paths connecting two vertices, is equal to the smallest number of edges separating these vertices.

10.2.3 Computing the values

Edge connectivity using maximum flow

This method is based on the Ford-Fulkerson theorem.

We iterate over all pairs of vertices (s, t) and between each pair we find the largest number of disjoint paths between them. This value can be found using a maximum flow algorithm: we use s as the source, t as the sink, and assign each edge a capacity of 1. Then the maximum flow is the number of disjoint paths.

The complexity for the algorithm using [Edmonds-Karp](#) is $O(V^2VE^2) = O(V^3E^2)$. But we should note, that this includes a hidden factor, since it is practically impossible to create a graph such that the maximum flow algorithm will be slow for all sources and sinks. Especially the algorithm will run pretty fast for random graphs.

Special algorithm for edge connectivity

The task of finding the edge connectivity is equal to the task of finding the **global minimum cut**.

Special algorithms have been developed for this task. One of them is the Stoer-Wagner algorithm, which works in $O(V^3)$ or $O(VE)$ time.

Vertex connectivity

Again we iterate over all pairs of vertices s and t , and for each pair we find the minimum number of vertices that separates s and t .

By doing this, we can apply the same maximum flow approach as described in the previous sections.

We split each vertex x with $x \neq s$ and $x \neq t$ into two vertices x_1 and x_2 . We connect these to vertices with a directed edge (x_1, x_2) with the capacity 1, and

replace all edges (u, v) by the two directed edges (u, v) and (v, u) , both with the capacity of 1. The by the construction the value of the maximum flow will be equal to the minimum number of vertices that are needed to separate s and t .

This approach has the same complexity as the flow approach for finding the edge connectivity.

10.3 Paint the edges of the tree

This is a fairly common task. Given a tree G with N vertices. There are two types of queries: the first one is to paint an edge, the second one is to query the number of colored edges between two vertices.

Here we will describe a fairly simple solution (using a [segment tree](#)) that will answer each query in $O(\log N)$ time. The preprocessing step will take $O(N)$ time.

10.3.1 Algorithm

First, we need to find the [LCA](#) to reduce each query of the second kind (i, j) into two queries (l, i) and (l, j) , where l is the LCA of i and j . The answer of the query (i, j) will be the sum of both subqueries. Both these queries have a special structure, the first vertex is an ancestor of the second one. For the rest of the article we will only talk about these special kind of queries.

We will start by describing the **preprocessing** step. Run a depth-first search from the root of the tree and record the Euler tour of this depth-first search (each vertex is added to the list when the search visits it first and every time we return from one of its children). The same technique can be used in the LCA preprocessing.

This list will contain each edge (in the sense that if i and j are the ends of the edge, then there will be a place in the list where i and j are neighbors in the list), and it appear exactly two times: in the forward direction (from i to j , where vertex i is closer to the root than vertex j) and in the opposite direction (from j to i).

We will build two lists for these edges. The first one will store the color of all edges in the forward direction, and the second one the color of all edges in the opposite direction. We will use 1 if the edge is colored, and 0 otherwise. Over these two lists we will build each a segment tree (for sum with a single modification), let's call them $T1$ and $T2$.

Let us answer a query of the form (i, j) , where i is the ancestor of j . We need to determine how many edges are painted on the path between i and j . Let's find i and j in the Euler tour for the first time, let it be the positions p and q (this can be done in $O(1)$ if we calculate these positions in advance during preprocessing). Then the **answer** to the query is the sum $T1[p..q - 1]$ minus the sum $T2[p..q - 1]$.

Why? Consider the segment $[p; q]$ in the Euler tour. It contains all edges of the path we need from i to j but also contains a set of edges that lie on other paths from i . However there is one big difference between the edges we need and the rest of the edges: the edges we need will be listed only once in the forward direction, and all the other edges appear twice: once in the forward and once in the opposite direction. Hence, the difference $T1[p..q - 1] - T2[p..q - 1]$ will give us the correct answer (minus one is necessary because otherwise, we will capture an extra edge going out from vertex j). The sum query in the segment tree is executed in $O(\log N)$.

Answering the **first type of query** (painting an edge) is even easier - we just need to update $T1$ and $T2$, namely to perform a single update of the element that corresponds to our edge (finding the edge in the list, again, is possible in $O(1)$, if you perform this search during preprocessing). A single modification in the segment tree is performed in $O(\log N)$.

10.3.2 Implementation

Here is the full implementation of the solution, including LCA computation:

```
const int INF = 1000 * 1000 * 1000;

typedef vector<vector<int>> graph;

vector<int> dfs_list;
vector<int> edges_list;
vector<int> h;

void dfs(int v, const graph& g, const graph& edge_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) {
            edges_list.push_back(edge_ids[v][i]);
            dfs(g[v][i], g, edge_ids, cur_h + 1);
            edges_list.push_back(edge_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> edge_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)edges_list.size(); ++i) {
        int j = edges_list[i];
        if (first1[j] == -1)
            first1[j] = i;
        else
            first2[j] = i;
    }

    edge_used.resize(n - 1);
    tree1.resize(edges_list.size() * 8);
    tree2.resize(edges_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)edges_list.size() - 1, first[v1], first[v2] - 1) +
        sum_tree_query(tree2, 1, 0, (int)edges_list.size() - 1, first[v1], first[v2] - 1);
}

int main() {
    // reading the graph
    int n;
    scanf("%d", &n);
    graph g(n), edge_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);
        edge_ids[v1].push_back(i);
        edge_ids[v2].push_back(i);
    }

    h.assign(n, -1);
    dfs(0, g, edge_ids);
    lca_prepare(n);
    query_prepare(n);

    for (;;) {
        if () {
            // request for painting edge x;
            // if start = true, then the edge is painted, otherwise the painting
            // is removed
            edge_used[x] = start;
            sum_tree_update(tree1, 1, 0, (int)edges_list.size() - 1, first1[x],
                start ? 1 : -1);
            sum_tree_update(tree2, 1, 0, (int)edges_list.size() - 1, first2[x],

```

```
                start ? 1 : -1);
    } else {
        // query the number of colored edges on the path between v1 and v2
        int l = lca(v1, v2);
        int result = query(l, v1) + query(l, v2);
        // result - the answer to the request
    }
}
}
```

10.4 2 - SAT

SAT (Boolean satisfiability problem) is the problem of assigning Boolean values to variables to satisfy a given Boolean formula. The Boolean formula will usually be given in CNF (conjunctive normal form), which is a conjunction of multiple clauses, where each clause is a disjunction of literals (variables or negation of variables). 2-SAT (2-satisfiability) is a restriction of the SAT problem, in 2-SAT every clause has exactly two literals. Here is an example of such a 2-SAT problem. Find an assignment of a, b, c such that the following formula is true:

$$(a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$

SAT is NP-complete, there is no known efficient solution known for it. However 2SAT can be solved efficiently in $O(n + m)$ where n is the number of variables and m is the number of clauses.

10.4.1 Algorithm:

First we need to convert the problem to a different form, the so-called implicative normal form. Note that the expression $a \vee b$ is equivalent to $\neg a \Rightarrow b \wedge \neg b \Rightarrow a$ (if one of the two variables is false, then the other one must be true).

We now construct a directed graph of these implications: for each variable x there will be two vertices v_x and $v_{\neg x}$. The edges will correspond to the implications.

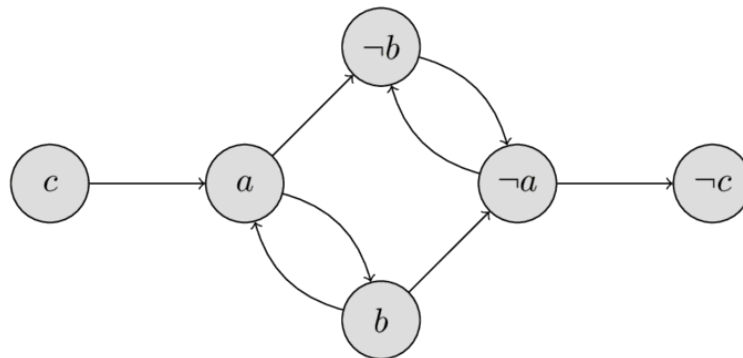
Lets look at the example in 2-CNF form:

$$(a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$

The oriented graph will contain the following vertices and edges:

$$\begin{array}{cccc} \neg a \Rightarrow \neg b & a \Rightarrow b & a \Rightarrow \neg b & \neg a \Rightarrow \neg c \\ b \Rightarrow a & \neg b \Rightarrow \neg a & b \Rightarrow \neg a & c \Rightarrow a \end{array}$$

You can see the implication graph in the following image:



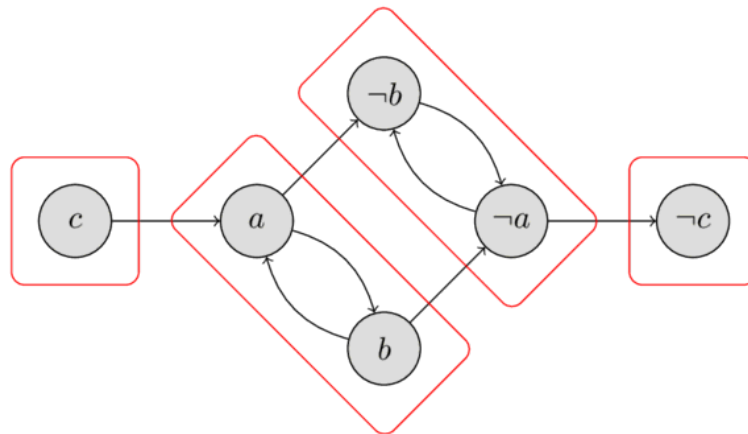
It is worth paying attention to the property of the implication graph: if there is an edge $a \Rightarrow b$, then there also is an edge $\neg b \Rightarrow \neg a$.

Also note, that if x is reachable from $\neg x$, and $\neg x$ is reachable from x , then the problem has no solution. Whatever value we choose for the variable x , it will always end in a contradiction - if x will be assigned true then the implication tell us that $\neg x$ should also be true and visa versa. It turns out, that this condition is not only necessary, but also sufficient. We will prove this in a few paragraphs below. First recall, if a vertex is reachable from a second one, and the second one is reachable from the first one, then these two vertices are in the same strongly connected component. Therefore we can formulate the criterion for the existence of a solution as follows:

In order for this 2-SAT problem to have a solution, it is necessary and sufficient that for any variable x the vertices x and $\neg x$ are in different strongly connected components of the strong connection of the implication graph.

This criterion can be verified in $O(n+m)$ time by finding all strongly connected components.

The following image shows all strongly connected components for the example. As we can check easily, neither of the four components contain a vertex x and its negation $\neg x$, therefore the example has a solution. We will learn in the next paragraphs how to compute a valid assignment, but just for demonstration purposes the solution $a = \text{false}$, $b = \text{false}$, $c = \text{false}$ is given.



Now we construct the algorithm for finding the solution of the 2-SAT problem on the assumption that the solution exists.

Note that, in spite of the fact that the solution exists, it can happen that $\neg x$ is reachable from x in the implication graph, or that (but not simultaneously) x is reachable from $\neg x$. In that case the choice of either true or false for x will lead to a contradiction, while the choice of the other one will not. Let's learn how to choose a value, such that we don't generate a contradiction.

Let us sort the strongly connected components in topological order (i.e. $\text{comp}[v] \leq \text{comp}[u]$ if there is a path from v to u) and let $\text{comp}[v]$ denote the

index of strongly connected component to which the vertex v belongs. Then, if $\text{comp}[x] < \text{comp}[\neg x]$ we assign x with false and true otherwise.

Let us prove that with this assignment of the variables we do not arrive at a contradiction. Suppose x is assigned with true. The other case can be proven in a similar way.

First we prove that the vertex x cannot reach the vertex $\neg x$. Because we assigned true it has to hold that the index of strongly connected component of x is greater than the index of the component of $\neg x$. This means that $\neg x$ is located on the left of the component containing x , and the later vertex cannot reach the first.

Secondly we prove that there doesn't exist a variable y , such that the vertices y and $\neg y$ are both reachable from x in the implication graph. This would cause a contradiction, because $x = \text{true}$ implies that $y = \text{true}$ and $\neg y = \text{true}$. Let us prove this by contradiction. Suppose that y and $\neg y$ are both reachable from x , then by the property of the implication graph $\neg x$ is reachable from both y and $\neg y$. By transitivity this results that $\neg x$ is reachable by x , which contradicts the assumption.

So we have constructed an algorithm that finds the required values of variables under the assumption that for any variable x the vertices x and $\neg x$ are in different strongly connected components. Above showed the correctness of this algorithm. Consequently we simultaneously proved the above criterion for the existence of a solution.

10.4.2 Implementation:

Now we can implement the entire algorithm. First we construct the graph of implications and find all strongly connected components. This can be accomplished with Kosaraju's algorithm in $O(n + m)$ time. In the second traversal of the graph Kosaraju's algorithm visits the strongly connected components in topological order, therefore it is easy to compute $\text{comp}[v]$ for each vertex v .

Afterwards we can choose the assignment of x by comparing $\text{comp}[x]$ and $\text{comp}[\neg x]$. If $\text{comp}[x] = \text{comp}[\neg x]$ we return false to indicate that there doesn't exist a valid assignment that satisfies the 2-SAT problem.

Below is the implementation of the solution of the 2-SAT problem for the already constructed graph of implication g and the transpose graph g^T (in which the direction of each edge is reversed). In the graph the vertices with indices $2k$ and $2k + 1$ are the two vertices corresponding to variable k with $2k + 1$ corresponding to the negated variable.

```
int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;

void dfs1(int v) {
    used[v] = true;
    for (int u : g[v]) {
```

```

        if (!used[u])
            dfs1(u);
    }
    order.push_back(v);
}

void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : gt[v]) {
        if (comp[u] == -1)
            dfs2(u, cl);
    }
}

bool solve_2SAT() {
    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[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }

    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])
            return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true;
}

```

10.4.3 Practice Problems

- [UVA: Rectangles](#)
- [Codeforces: The Door Problem](#)

10.5 Heavy-light decomposition

Heavy-light decomposition is a fairly general technique that allows us to effectively solve many problems that come down to **queries on a tree** .

10.5.1 Description

Let there be a tree G of n vertices, with an arbitrary root.

The essence of this tree decomposition is to **split the tree into several paths** so that we can reach the root vertex from any v by traversing at most $\log n$ paths. In addition, none of these paths should intersect with another.

It is clear that if we find such a decomposition for any tree, it will allow us to reduce certain single queries of the form “*calculate something on the path from a to b* ” to several queries of the type “*calculate something on the segment $[l; r]$ of the k^{th} path*”.

Construction algorithm

We calculate for each vertex v the size of its subtree $s(v)$, i.e. the number of vertices in the subtree of the vertex v including itself.

Next, consider all the edges leading to the children of a vertex v . We call an edge **heavy** if it leads to a vertex c such that:

$$s(c) \geq \frac{s(v)}{2} \iff \text{edge } (v, c) \text{ is heavy}$$

All other edges are labeled **light**.

It is obvious that at most one heavy edge can emanate from one vertex downward, because otherwise the vertex v would have at least two children of size $\geq \frac{s(v)}{2}$, but size of subtree of v , $s(v) < 1 + 2\frac{s(v)}{2}$ which leads to a contradiction.

Now we will decompose the tree into disjoint paths. Consider all the vertices from which no heavy edges come down. We will go up from each such vertex until we reach the root of the tree or go through a light edge. As a result, we will get several paths which are made up of zero or more heavy edges plus one light edge. The path which has an end at the root is an exception to this and will not have a light edge. Let these be called **heavy paths** - these are the desired paths of heavy-light decomposition.

Proof of correctness

First, we note that the heavy paths obtained by the algorithm will be **disjoint** . In fact, if two such paths have a common edge, it would imply that there are two heavy edges coming out of one vertex, which is impossible.

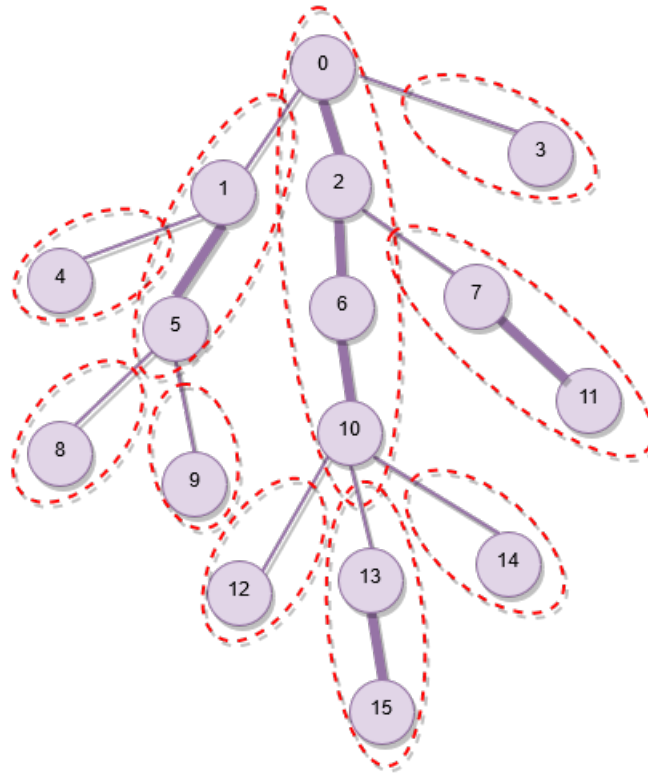
Secondly, we will show that going down from the root of the tree to an arbitrary vertex, we will **change no more than $\log n$ heavy paths along the way** . Moving down a light edge reduces the size of the current subtree to half or lower:

$$s(c) < \frac{s(v)}{2} \iff \text{edge } (v, c) \text{ is light}$$

Thus, we can go through at most $\log n$ light edges before subtree size reduces to one.

Since we can move from one heavy path to another only through a light edge (each heavy path, except the one starting at the root, has one light edge), we cannot change heavy paths more than $\log n$ times along the path from the root to any vertex, as required.

The following image illustrates the decomposition of a sample tree. The heavy edges are thicker than the light edges. The heavy paths are marked by dotted boundaries.



10.5.2 Sample problems

When solving problems, it is sometimes more convenient to consider the heavy-light decomposition as a set of **vertex disjoint** paths (rather than edge disjoint paths). To do this, it suffices to exclude the last edge from each heavy path if it is a light edge, then no properties are violated, but now each vertex belongs to exactly one heavy path.

Below we will look at some typical tasks that can be solved with the help of heavy-light decomposition.

Separately, it is worth paying attention to the problem of the **sum of numbers on the path**, since this is an example of a problem that can be solved by simpler techniques.

Maximum value on the path between two vertices

Given a tree, each vertex is assigned a value. There are queries of the form (a, b) , where a and b are two vertices in the tree, and it is required to find the maximum value on the path between the vertices a and b .

We construct in advance a heavy-light decomposition of the tree. Over each heavy path we will construct a [segment tree](#), which will allow us to search for a vertex with the maximum assigned value in the specified segment of the specified heavy path in $\mathcal{O}(\log n)$. Although the number of heavy paths in heavy-light decomposition can reach $n - 1$, the total size of all paths is bounded by $\mathcal{O}(n)$, therefore the total size of the segment trees will also be linear.

In order to answer a query (a, b) , we find the [lowest common ancestor](#) of a and b as l , by any preferred method. Now the task has been reduced to two queries (a, l) and (b, l) , for each of which we can do the following: find the heavy path that the lower vertex lies in, make a query on this path, move to the top of this path, again determine which heavy path we are on and make a query on it, and so on, until we get to the path containing l .

One should be careful with the case when, for example, a and l are on the same heavy path - then the maximum query on this path should be done not on any prefix, but on the internal section between a and l .

Responding to the subqueries (a, l) and (b, l) each requires going through $\mathcal{O}(\log n)$ heavy paths and for each path a maximum query is made on some section of the path, which again requires $\mathcal{O}(\log n)$ operations in the segment tree. Hence, one query (a, b) takes $\mathcal{O}(\log^2 n)$ time.

If you additionally calculate and store maximums of all prefixes for each heavy path, then you get a $\mathcal{O}(\log n)$ solution because all maximum queries are on prefixes except at most once when we reach the ancestor l .

Sum of the numbers on the path between two vertices

Given a tree, each vertex is assigned a value. There are queries of the form (a, b) , where a and b are two vertices in the tree, and it is required to find the sum of the values on the path between the vertices a and b . A variant of this task is possible where additionally there are update operations that change the number assigned to one or more vertices.

This task can be solved similar to the previous problem of maximums with the help of heavy-light decomposition by building segment trees on heavy paths. Prefix sums can be used instead if there are no updates. However, this problem can be solved by simpler techniques too.

If there are no updates, then it is possible to find out the sum on the path between two vertices in parallel with the LCA search of two vertices by [binary lifting](#) — for this, along with the 2^k -th ancestors of each vertex it is also necessary to store the sum on the paths up to those ancestors during the preprocessing.

There is a fundamentally different approach to this problem - to consider the [Euler tour](#) of the tree, and build a segment tree on it. This algorithm is considered in an [article about a similar problem](#). Again, if there are no updates, storing prefix sums is enough and a segment tree is not required.

Both of these methods provide relatively simple solutions taking $\mathcal{O}(\log n)$ for one query.

Repainting the edges of the path between two vertices

Given a tree, each edge is initially painted white. There are updates of the form (a, b, c) , where a and b are two vertices and c is a color, which instructs that all the edges on the path from a to b must be repainted with color c . After all repaintings, it is required to report how many edges of each color were obtained.

Similar to the above problems, the solution is to simply apply heavy-light decomposition and make a [segment tree](#) over each heavy path.

Each repainting on the path (a, b) will turn into two updates (a, l) and (b, l) , where l is the lowest common ancestor of the vertices a and b . $\mathcal{O}(\log n)$ per path for $\mathcal{O}(\log n)$ paths leads to a complexity of $\mathcal{O}(\log^2 n)$ per update.

10.5.3 Implementation

Certain parts of the above discussed approach can be modified to make implementation easier without losing efficiency.

- The definition of **heavy edge** can be changed to **the edge leading to the child with largest subtree**, with ties broken arbitrarily. This may result in some light edges being converted to heavy, which means some heavy paths will combine to form a single path, but all heavy paths will remain disjoint. It is also still guaranteed that going down a light edge reduces subtree size to half or less.
- Instead of building a segment tree over every heavy path, a single segment tree can be used with disjoint segments allocated to each heavy path.
- It has been mentioned that answering queries requires calculation of the LCA. While LCA can be calculated separately, it is also possible to integrate LCA calculation in the process of answering queries.

To perform heavy-light decomposition:

```
vector<int> parent, depth, heavy, head, pos;
int cur_pos;

int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1;
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
}
```

```

    }
    return size;
}

int decompose(int v, int h, vector<vector<int>> const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
            decompose(c, c, adj);
    }
}

void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    cur_pos = 0;

    dfs(0, adj);
    decompose(0, 0, adj);
}

```

The adjacency list of the tree must be passed to the `init` function, and decomposition is performed assuming vertex 0 as root.

The `dfs` function is used to calculate `heavy[v]`, the child at the other end of the heavy edge from `v`, for every vertex `v`. Additionally `dfs` also stores the parent and depth of each vertex, which will be useful later during queries.

The `decompose` function assigns for each vertex `v` the values `head[v]` and `pos[v]`, which are respectively the head of the heavy path `v` belongs to and the position of `v` on the single segment tree that covers all vertices.

To answer queries on paths, for example the maximum query discussed, we can do something like this:

```

int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
        int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
        res = max(res, cur_heavy_path_max);
    }
    if (depth[a] > depth[b])
        swap(a, b);
    int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
    res = max(res, last_heavy_path_max);
    return res;
}

```


Part X

Miscellaneous

Chapter 1

Sequences

1.1 Range Minimum Query

You are given an array $A[1..N]$. You have to answer incoming queries of the form (L, R) , which ask to find the minimum element in array A between positions L and R inclusive.

RMQ can appear in problems directly or can be applied in some other tasks, e.g. the [Lowest Common Ancestor](#) problem.

1.1.1 Solution

There are lots of possible approaches and data structures that you can use to solve the RMQ task.

The ones that are explained on this site are listed below.

First the approaches that allow modifications to the array between answering queries.

- [Sqrt-decomposition](#) - answers each query in $O(\sqrt{N})$, preprocessing done in $O(N)$. Pros: a very simple data structure. Cons: worse complexity.
- [Segment tree](#) - answers each query in $O(\log N)$, preprocessing done in $O(N)$. Pros: good time complexity. Cons: larger amount of code compared to the other data structures.
- [Fenwick tree](#) - answers each query in $O(\log N)$, preprocessing done in $O(N \log N)$. Pros: the shortest code, good time complexity. Cons: Fenwick tree can only be used for queries with $L = 1$, so it is not applicable to many problems.

And here are the approaches that only work on static arrays, i.e. it is not possible to change a value in the array without recomputing the complete data structure.

- [Sparse Table](#) - answers each query in $O(1)$, preprocessing done in $O(N \log N)$. Pros: simple data structure, excellent time complexity.
- [Sqrt Tree](#) - answers queries in $O(1)$, preprocessing done in $O(N \log \log N)$. Pros: fast. Cons: Complicated to implement.

- [Disjoint Set Union / Arpa's Trick](#) - answers queries in $O(1)$, preprocessing in $O(n)$. Pros: short, fast. Cons: only works if all queries are known in advance, i.e. only supports off-line processing of the queries.
- [Cartesian Tree](#) and [Farach-Colton and Bender algorithm](#) - answers queries in $O(1)$, preprocessing in $O(n)$. Pros: optimal complexity. Cons: large amount of code.

Note: Preprocessing is the preliminary processing of the given array by building the corresponding data structure for it.

1.1.2 Practice Problems

- [SPOJ: Range Minimum Query](#)
- [CODECHEF: Chef And Array](#)

1.2 Longest increasing subsequence

We are given an array with n numbers: $a[0 \dots n-1]$. The task is to find the longest, strictly increasing, subsequence in a .

Formally we look for the longest sequence of indices i_1, \dots, i_k such that

$$i_1 < i_2 < \dots < i_k, a[i_1] < a[i_2] < \dots < a[i_k]$$

In this article we discuss multiple algorithms for solving this task. Also we will discuss some other problems, that can be reduced to this problem.

1.2.1 Solution in $O(n^2)$ with dynamic programming

Dynamic programming is a very general technique that allows to solve a huge class of problems. Here we apply the technique for our specific task.

First we will search only for the **length** of the longest increasing subsequence, and only later learn how to restore the subsequence itself.

Finding the length

To accomplish this task, we define an array $d[0 \dots n-1]$, where $d[i]$ is the length of the longest increasing subsequence that ends in the element at index i . We will compute this array gradually: first $d[0]$, then $d[1]$, and so on. After this array is computed, the answer to the problem will be the maximum value in the array $d[]$.

So let the current index be i . I.e. we want to compute the value $d[i]$ and all previous values $d[0], \dots, d[i-1]$ are already known. Then there are two options:

- $d[i] = 1$: the required subsequence consists of only the element $a[i]$.
- $d[i] > 1$: then in the required subsequence is another number before the number $a[i]$. Let's focus on that number: it can be any element $a[j]$ with $j = 0 \dots i-1$ and $a[j] < a[i]$. In this fashion we can compute $d[i]$ using the following formula: If we fixate the index j , then the longest increasing subsequence ending in the two elements $a[j]$ and $a[i]$ has the length $d[j] + 1$. All of these values $d[j]$ are already known, so we can directly compute $d[i]$ with:

$$d[i] = \max_{\substack{j=0 \dots i-1 \\ a[j] < a[i]}} (d[j] + 1)$$

If we combine these two cases we get the final answer for $d[i]$:

$$d[i] = \max \left(1, \max_{\substack{j=0 \dots i-1 \\ a[j] < a[i]}} (d[j] + 1) \right)$$

Implementation

Here is an implementation of the algorithm described above, which computes the length of the longest increasing subsequence.

```
int lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i])
                d[i] = max(d[i], d[j] + 1);
        }
    }

    int ans = d[0];
    for (int i = 1; i < n; i++) {
        ans = max(ans, d[i]);
    }
    return ans;
}
```

Restoring the subsequence

So far we only learned how to find the length of the subsequence, but not how to find the subsequence itself.

To be able to restore the subsequence we generate an additional auxiliary array $p[0 \dots n-1]$ that we will compute alongside the array $d[]$. $p[i]$ will be the index j of the second last element in the longest increasing subsequence ending in i . In other words the index $p[i]$ is the same index j at which the highest value $d[j]$ was obtained. This auxiliary array $p[]$ points in some sense to the ancestors.

Then to derive the subsequence, we just start at the index i with the maximal $d[i]$, and follow the ancestors until we deduced the entire subsequence, i.e. until we reach the element with $d[i] = 1$.

Implementation of restoring

We will change the code from the previous sections a little bit. We will compute the array $p[]$ alongside $d[]$, and afterwards compute the subsequence.

For convenience we originally assign the ancestors with $p[i] = -1$. For elements with $d[i] = 1$, the ancestors value will remain -1 , which will be slightly more convenient for restoring the subsequence.

```
vector<int> lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1), p(n, -1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i] && d[i] < d[j] + 1) {
                d[i] = d[j] + 1;
                p[i] = j;
            }
        }
    }

    int ans = d[0];
    for (int i = 1; i < n; i++) {
        ans = max(ans, d[i]);
    }

    vector<int> subsequence;
    int i = n - 1;
    while (i >= 0) {
        subsequence.push_back(a[i]);
        i = p[i];
    }
    reverse(subsequence.begin(), subsequence.end());
    return subsequence;
}
```

```

        p[i] = j;
    }
}

int ans = d[0], pos = 0;
for (int i = 1; i < n; i++) {
    if (d[i] > ans) {
        ans = d[i];
        pos = i;
    }
}

vector<int> subseq;
while (pos != -1) {
    subseq.push_back(a[pos]);
    pos = p[pos];
}
reverse(subseq.begin(), subseq.end());
return subseq;
}

```

Alternative way of restoring the subsequence

It is also possible to restore the subsequence without the auxiliary array $p[]$. We can simply recalculate the current value of $d[i]$ and also see how the maximum was reached.

This method leads to a slightly longer code, but in return we save some memory.

1.2.2 Solution in $O(n \log n)$ with dynamic programming and binary search

In order to obtain a faster solution for the problem, we construct a different dynamic programming solution that runs in $O(n^2)$, and then later improve it to $O(n \log n)$.

We will use the dynamic programming array $d[0 \dots n]$. This time $d[i]$ will be the element at which a subsequence of length i terminates. If there are multiple such sequences, then we take the one that ends in the smallest element.

Initially we assume $d[0] = -\infty$ and for all other elements $d[i] = \infty$.

We will again gradually process the numbers, first $a[0]$, then $a[1]$, etc, and in each step maintain the array $d[]$ so that it is up to date.

After processing all the elements of $a[]$ the length of the desired subsequence is the largest l with $d[l] < \infty$.

```

int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -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];
        }
    }

    int ans = 0;
    for (int i = 0; i <= n; i++) {
        if (d[i] < INF)
            ans = i;
    }
    return ans;
}

```

We now make two important observations.

The array d will always be sorted: $d[i-1] \leq d[i]$ for all $i = 1 \dots n$. And also the element $a[i]$ will only update at most one value $d[j]$.

Thus we can find this element in the array $d[]$ using binary search in $O(\log n)$. In fact we are simply looking in the array $d[]$ for the first number that is strictly greater than $a[i]$, and we try to update this element in the same way as the above implementation.

Implementation

```

int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int j = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[j-1] < a[i] && a[i] < d[j])
            d[j] = a[i];
    }

    int ans = 0;
    for (int i = 0; i <= n; i++) {
        if (d[i] < INF)
            ans = i;
    }
    return ans;
}

```

Restoring the subsequence

It is also possible to restore the subsequence using this approach. This time we have to maintain two auxiliary arrays. One that tells us the index of the elements

in $d[]$. And again we have to create an array of “ancestors” $p[]$. $p[i]$ will be the index of the previous element for the optimal subsequence ending in element i .

It’s easy to maintain these two arrays in the course of iteration over the array $a[]$ alongside the computations of $d[]$. And at the end it is not difficult to restore the desired subsequence using these arrays.

1.2.3 Solution in $O(n \log n)$ with data structures

Instead of the above method for computing the longest increasing subsequence in $O(n \log n)$ we can also solve the problem in a different way: using some simple data structures.

Let’s go back to the first method. Remember that $d[i]$ is the value $d[j] + 1$ with $j < i$ and $a[j] < a[i]$.

Thus if we define an additional array $t[]$ such that

$$t[a[i]] = d[i],$$

then the problem of computing the value $d[i]$ is equivalent to finding the **maximum value in a prefix** of the array $t[]$:

$$d[i] = \max(t[0 \dots a[i] - 1] + 1)$$

The problem of finding the maximum of a prefix of an array (which changes) is a standard problem that can be solved by many different data structures. For instance we can use a [Segment tree](#) or a [Fenwick tree](#).

This method has obviously some **shortcomings**: in terms of length and complexity of the implementation this approach will be worse than the method using binary search. In addition if the input numbers $a[i]$ are especially large, then we would have to use some tricks, like compressing the numbers (i.e. renumber them from 0 to $n - 1$), or use an implicit Segment tree (only generate the branches of the tree that are important). Otherwise the memory consumption will be too high.

On the other hand this method has also some **advantages**: with this method you don’t have to think about any tricky properties in the dynamic programming solution. And this approach allows us to generalize the problem very easily (see below).

1.2.4 Related tasks

Here are several problems that are closely related to the problem of finding the longest increasing subsequence.

Longest non-decreasing subsequence

This is in fact nearly the same problem. Only now it is allowed to use identical numbers in the subsequence.

The solution is essentially also nearly the same. We just have to change the inequality signs, and make a slightly modification to the binary search.

Number of longest increasing subsequences

We can use the first discussed method, either the $O(n^2)$ version or the version using data structures. We only have to additionally store in how many ways we can obtain longest increasing subsequences ending in the values $d[i]$.

The number of ways to form a longest increasing subsequences ending in $a[j]$ is the sum of all ways for all longest increasing subsequences ending in j where $d[j]$ is maximal. There can be multiple such j , so we need to sum all of them.

Using a Segment tree this approach can also be implemented in $O(n \log n)$.

It is not possible to use the binary search approach for this task.

Smallest number of non-increasing subsequences covering a sequence

For a given array with n numbers $a[0 \dots n - 1]$ we have to colorize the numbers in the smallest number of colors, so that each color forms a non-increasing subsequence.

To solve this, we notice that the minimum number of required colors is equal to the length of the longest increasing subsequence.

Proof: We need to prove the **duality** of these two problems.

Let's denote by x the length of the longest increasing subsequence and by y the least number of non-increasing subsequences that form a cover. We need to prove that $x = y$.

It is clear that $y < x$ is not possible, because if we have x strictly increasing elements, than no two can be part of the same non-increasing subsequence. Therefore we have $y \geq x$.

We now show that $y > x$ is not possible by contradiction. Suppose that $y > x$. Then we consider any optimal set of y non-increasing subsequences. We transform this in set in the following way: as long as there are two such subsequences such that the first begins before the second subsequence, and the first sequence start with a number greater than or equal to the second, then we unhook this starting number and attach it to the beginning of second. After a finite number of steps we have y subsequences, and their starting numbers will form an increasing subsequence of length y . Since we assumed that $y > x$ we reached a contradiction.

Thus it follows that $y = x$.

Restoring the sequences: The desired partition of the sequence into subsequences can be done greedily. I.e. go from left to right and assign the current number or that subsequence ending with the minimal number which is greater than or equal to the current one.

1.2.5 Practice Problems

- [Topcoder - IntegerSequence](#)
- [Topcoder - AutoMarket](#)
- [Codeforces - Tourist](#)
- [Codeforces - LCIS](#)
- [SPOJ - SUPPER](#)

-
- [Topcoder - BridgeArrangement](#)
 - [ACMSGURU - “North-East”](#)

1.3 Search the subsegment with the maximum/minimum sum

Here we consider the problem of finding a subsegment of an array with maximum sum, as well as some of its variations (including the algorithm for solving this problem in the online mode).

1.3.1 Problem statement

Given an array of numbers $a[1 \dots n]$. It is required to find such the subsegment $a[l \dots r]$ with the maximal sum:

$$\max_{1 \leq l \leq r \leq n} \sum_{i=l}^r a[i].$$

For example, if all integers in array $a[]$ would be a non-negative, then the answer would be to take the whole array. However the solution is nontrivial when the array can contain both positive and negative numbers.

It is clear that the problem of finding the **minimum** subsegment is essentially the same, you just need to change the signs of all numbers.

1.3.2 Algorithm 1

Here we consider an almost obvious algorithm. (Next, we'll look at another algorithm, which is a little harder to come up with, but its implementation is even shorter.)

Algorithm description

The algorithm is very simple.

We introduce for convenience the **notation**: $s[i] = \sum_{j=1}^i a[j]$. That is, the array $s[i]$ is an array of partial sums of array $a[]$. Also put $s[0] = 0$.

Let us now iterate over the index $r = 1 \dots n$, and learn how to quickly find the optimal l for each current value r , at which the maximum sum is reached on the subsegment $[l; r]$.

Formally, this means that for the current r we need to find such l (not exceeding r), so that the value of $s[r] - s[l - 1]$ is maximal. After a trivial transformation, we get that we need to find in the array $s[]$ a minimum on the segment $[0; r - 1]$.

From here, we immediately obtain a solution: we simply store where the current minimum is in the array $s[]$. Using this minimum, we find the current optimal index l in $O(1)$, and when moving from the current index r to the next one, we simply update this minimum.

Obviously, this algorithm works in $O(n)$ and is asymptotically optimal.

Implementation

To implement it, we don't even need to explicitly store an array of partial sums $s[]$ — we will only need the current element from it.

The implementation is given in 0-indexed arrays, not in 1-numbering as described above.

We first give a solution that finds a simple numerical answer without finding the indices of the desired 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 give a full version of the solution, which additionally also finds the boundaries of the desired segment:

```
int ans = a[0], ans_l = 0, ans_r = 0;
int 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;
    }
}
```

1.3.3 Algorithm 2

Here we consider a different algorithm. It is a little more difficult to understand, but it is more elegant than the above, and its implementation is a little bit shorter. This algorithm was proposed by Jay Kadane in 1984.

Algorithm description

The algorithm itself is as follows. Let's go through the array and accumulate the current partial sum in some variable s . If at some point s is negative, we just assign $s = 0$. It is argued that the maximum all the values, that the variable s is assigned to during the algorithm, will be the answer to the problem.

Proof:

In fact, consider the first moment in time when the sum of s became negative. This means that starting with a zero partial sum, we eventually came to a negative partial sum — so this whole prefix of the array, as well as any suffix has a negative sum. Therefore, from all this prefix of the array in the future there cannot benefit: it can only give a negative impact to the answer.

However, this is not enough to prove the algorithm. In the algorithm, we are actually limited in finding the answer only to such segments that begin immediately after the places when $s < 0$ happened.

But, in fact, consider an arbitrary segment $[l; r]$, and l is not in such a “critical” position (i.e. $l > p + 1$, where p is the last such position, in which $s < 0$). Since the last critical position is strictly earlier than in $l - 1$, it turns out that the sum of $a[p + 1 \dots l - 1]$ is non-negative. This means that by moving l to position $p + 1$, we will increase the answer or, in extreme cases, we will not change it.

One way or another, it turns out that really when searching for an answer, you can limit yourself to only segments that begin immediately after the positions in which $s < 0$ appeared. This proves that the algorithm is correct.

Implementation

As in algorithm 1, we first gave a simplified implementation that looks for only a numerical answer without 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);
}
```

A complete solution, maintaining the indexes of the boundaries of the corresponding segment:

```
int ans = a[0], ans_l = 0, ans_r = 0;
int 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;
    }
}
```

1.3.4 Related tasks

Finding the maximum / minimum subsegment with constraints

If the problem condition imposes additional restrictions on the required segment $[l; r]$ (for example, that the length $r - l + 1$ of the segment must be within the specified limits), then the described algorithm is likely to be easily generalized to these cases — anyway, the problem will still be to find the minimum in the array $s[]$ with the specified additional restrictions.

Two-dimensional case of the problem: search for maximum / minimum submatrix

The problem described in this article is naturally generalized to large dimensions. For example, in a two-dimensional case, it turns into a search for such a submatrix $[l_1 \dots r_1; l_2 \dots r_2]$ of a given matrix, which has the maximum sum of numbers in it.

Using the solution for the one-dimensional case, it is easy to obtain a solution in $O(n^3)$ for the two-dimensions case: we iterate over all possible values of l_1 and r_1 , and calculate the sums from l_1 to r_1 in each row of the matrix. Now we have the one-dimensional problem of finding the indices l_2 and r_2 in this array, which can already be solved in linear time.

Faster algorithms for solving this problem are known, but they are not much faster than $O(n^3)$, and are very complex (so complex that many of them are inferior to the trivial algorithm for all reasonable constraints by the hidden constant). Currently, the best known algorithm works in $O\left(n^3 \frac{\log^3 \log n}{\log^2 n}\right)$ time (T. Chan 2007 “More algorithms for all-pairs shortest paths in weighted graphs”)

This algorithm by Chan, as well as many other results in this area, actually describe **fast matrix multiplication** (where matrix multiplication means modified multiplication: minimum is used instead of addition, and addition is used instead of multiplication). The problem of finding the submatrix with the largest sum can be reduced to the problem of finding the shortest paths between all pairs of vertices, and this problem, in turn, can be reduced to such a multiplication of matrices.

Search for a subsegment with a maximum / minimum average amount

This problem lies in finding such a segment $a[l; r]$, such that the average value is maximal:

$$\max_{l \leq r} \frac{1}{r - l + 1} \sum_{i=l}^r a[i].$$

Of course, if no other conditions are imposed on the required segment $[l; r]$, then the solution will always be a segment of length 1 at the maximum element of the array. The problem only makes sense, if there are additional restrictions (for example, the length of the desired segment is bounded below).

In this case, we apply the **standard technique** when working with the problems of the average value: we will select the desired maximum average value by **binary search**.

To do this, we need to learn how to solve this subproblem: given the number x , and we need to check whether there is a subarray of array $a[]$ (of course, satisfying all additional constraints of the problem), where the average value is greater than x .

To solve this subproblem, subtract x from each element of array $a[]$. Then our subproblem actually turns into this one: whether or not there are positive sum subsegments in this array. And we already know how to solve this problem.

Thus, we obtained the solution for the asymptotic $O(T(n) \log W)$, where W is the required accuracy, $T(n)$ is the time of solving the subtask for an array of length n (which may vary depending on the specific additional restrictions imposed).

Solving the online problem

The condition of the problem is as follows: given an array of n numbers, and a number L . There are queries of the form (l, r) , and in response to each query, it is required to find a subsegment of the segment $[l; r]$ of length not less than L with the maximum possible arithmetic mean.

The algorithm for solving this problem is quite complex. KADR (Yaroslav Tverdokhlebov) described his algorithm on the [Russian forum](#).

1.4 K-th order statistic in $O(N)$

Given an array **A** of size **N** and a number **K**. The challenge is to find **K**-th largest number in the array, i.e., **K**-th order statistic.

The basic idea - to use the idea of quick sort algorithm. Actually, the algorithm is simple, it is more difficult to prove that it runs in an average of $O(N)$, in contrast to the quick sort.

1.4.1 Implementation (not recursive):

```
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)
        {
            // the current part size is either 1 or 2, so it is easy to find the answer
            if (r == l+1 && a[r] < a[l])
                swap (a[l], a[r]);
            return a[k];
        }

        // ordering 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]);

        // performing 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]);
        }

        // inserting the barrier
```



```
    a[l+1] = a[j];
    a[j] = cur;

    // we continue to work in that part, which must contain the required element
    if (j >= k)
        r = j-1;
    if (j <= k)
        l = i;
}
}
```

To note, in the standard C ++ library, this algorithm has already been implemented - it is called `nth_element`.

1.4.2 Practice Problems

- [CODECHEF: Median](#)

Chapter 2

Game Theory

2.1 Games on arbitrary graphs

Let a game be played by two players on an arbitrary graph G . I.e. the current state of the game is a certain vertex. The player perform moves by turns, and move from the current vertex to an adjacent vertex using a connecting edge. Depending on the game, the person that is unable to move will either lose or win the game.

We consider the most general case, the case of an arbitrary directed graph with cycles. It is our task to determine, given an initial state, who will win the game if both players play with optimal strategies or determine that the result of the game will be a draw.

We will solve this problem very efficiently. We will find the solution for all possible starting vertices of the graph in linear time with respect to the number of edges: $O(m)$.

2.1.1 Description of the algorithm

We will call a vertex a winning vertex, if the player starting at this state will win the game, if he plays optimal (regardless what turns the other player makes). And similarly we will call a vertex a losing vertex, if the player starting at this vertex will lose the game, if the opponent plays optimal.

For some of the vertices of the graph we already know in advance that they are winning or losing vertices: namely all vertices that have no outgoing edges.

Also we have the following **rules**:

- if a vertex has an outgoing edge that leads to a losing vertex, then the vertex itself is a winning vertex.
- if all outgoing edges of a certain vertex lead to winning vertices, then the vertex itself is a losing vertex.
- if at some point there are still undefined vertices, and neither will fit the first or the second rule, then each of these vertices, when used as a starting vertex, will lead to a draw if both player play optimal.

Thus we can define an algorithm which runs in $O(nm)$ time immediately. We go through all vertices and try to apply the first or second rule, and repeat.

However we can accelerate this procedure, and get the complexity down to $O(m)$.

We will go over all the vertices, for which we initially know if they are winning or losing states. For each of them we start a [depth first search](#). This DFS will move back over the reversed edges. First of all, it will not enter vertices which already are defined as winning or losing vertices. And further, if the search goes from a losing vertex to an undefined vertex, then we mark this one as a winning vertex, and continue the DFS using this new vertex. If we go from a winning vertex to an undefined vertex, then we must check whether all edges from this one leads to winning vertices. We can perform this test in $O(1)$ by storing the number of edges that lead to a winning vertex for each vertex. So if we go from a winning vertex to an undefined one, then we increase the counter, and check if this number is equal to the number of outgoing edges. If this is the case, we can mark this vertex as a losing vertex, and continue the DFS from this vertex. Otherwise we don't know yet, if this vertex is a winning or losing vertex, and therefore it doesn't make sense to keep continuing the DFS using it.

In total we visit every winning and every losing vertex exactly once (undefined vertices are not visited), and we go over each edge also at most one time. Hence the complexity is $O(m)$.

2.1.2 Implementation

Here is the implementation of such a DFS. We assume that the variable `adj_rev` stores the adjacency list for the graph in **reversed** form, i.e. instead of storing the edge (i, j) of the graph, we store (j, i) . Also for each vertex we assume that the outgoing degree is already computed.

```
vector<vector<int>> adj_rev;

vector<bool> winning;
vector<bool> losing;
vector<bool> visited;
vector<int> degree;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj_rev[v]) {
        if (!visited[u]) {
            if (losing[v])
                winning[u] = true;
            else if (--degree[u] == 0)
                losing[u] = true;
            else
                continue;
            dfs(u);
        }
    }
}
```

```
    }
}
```

2.1.3 Example: “Policeman and thief”

Here is a concrete example of such a game.

There is $m \times n$ board. Some of the cells cannot be entered. The initial coordinates of the police officer and of the thief are known. One of the cells is the exit. If the policeman and the thief are located at the same cell at any moment, the policeman wins. If the thief is at the exit cell (without the policeman also being on the cell), then the thief wins. The policeman can walk in all 8 directions, the thief only in 4 (along the coordinate axis). Both the policeman and the thief will take turns moving. However they also can skip a turn if they want to. The first move is made by the policeman.

We will now **construct the graph**. For this we must formalize the rules of the game. The current state of the game is determined by the coordinates of the police offices P , the coordinates of the thief T , and also by whose turn it is, let's call this variable P_{turn} (which is true when it is the policeman's turn). Therefore a vertex of the graph is determined by the triple (P, T, P_{turn}) . The graph then can be easily constructed, simply by following the rules of the game.

Next we need to determine which vertices are winning and which are losing ones initially. There is a **subtle point** here. The winning / losing vertices depend, in addition to the coordinates, also on P_{turn} - whose turn it is. If it is the policeman's turn, then the vertex is a winning vertex, if the coordinates of the policeman and the thief coincide, and the vertex is a losing one if it is not a winning one and the thief is on the exit vertex. If it is the thief's turn, then a vertex is a losing vertex, if the coordinates of the two players coincide, and it is a winning vertex if it is not a losing one, and the thief is at the exit vertex.

The only point before implementing is not, that you need to decide if you want to build the graph **explicitly** or just construct it **on the fly**. On one hand, building the graph explicitly will be a lot easier and there is less chance of making mistakes. On the other hand, it will increase the amount of code and the running time will be slower than if you build the graph on the fly.

The following implementation will construct the graph explicitly:

```
struct State {
    int P, T;
    bool Pstep;
};

vector<State> adj_rev[100][100][2]; // [P][T][Pstep]
bool winning[100][100][2];
bool losing[100][100][2];
bool visited[100][100][2];
int degree[100][100][2];

void dfs(State v) {
    visited[v.P][v.T][v.Pstep] = true;
```

```

    for (State u : adj_rev[v.P][v.T][v.Pstep]) {
        if (!visited[u.P][u.T][u.Pstep]) {
            if (losing[v.P][v.T][v.Pstep])
                winning[u.P][u.T][u.Pstep] = true;
            else if (--degree[u.P][u.T][u.Pstep] == 0)
                losing[u.P][u.T][u.Pstep] = true;
            else
                continue;
            dfs(u);
        }
    }
}

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 (int 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& win = winning[P][T][Pstep];
                bool& lose = losing[P][T][Pstep];
                if (Pstep) {
                    win = Px==Tx && Py==Ty;
                    lose = !win && a[Tx][Ty] == 'E';
                } else {
                    lose = Px==Tx && Py==Ty;
                    win = !lose && a[Tx][Ty] == 'E';
                }
                if (win || lose)
                    continue;

                State st = {P,T,!Pstep};
                adj_rev[P][T][Pstep].push_back(st);
                st.Pstep = Pstep;
                degree[P][T][Pstep]++;

                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] != '*')
    {
        adj_rev[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 (int Pstep = 0; Pstep <= 1; Pstep++) {
            if ((winning[P][T][Pstep] || losing[P][T][Pstep]) && !visited[P][T][Pstep])
                dfs({P, T, (bool)Pstep});
        }
    }
}

int P_st, T_st;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (a[i][j] == 'P')
            P_st = i*m+j;
        else if (a[i][j] == 'T')
            T_st = i*m+j;
    }
}

if (winning[P_st][T_st][true]) {
    cout << "Police catches the thief" << endl;
} else if (losing[P_st][T_st][true]) {
    cout << "The thief escapes" << endl;
} else {
    cout << "Draw" << endl;
}
}

```

Chapter 3

Schedules

3.1 Scheduling jobs on one machine

This task is about finding an optimal schedule for n jobs on a single machine, if the job i can be processed in t_i time, but for the t seconds waiting before processing the job a penalty of $f_i(t)$ has to be paid.

Thus the task asks to find such a permutation of the jobs, so that the total penalty is minimal. If we denote by π the permutation of the jobs (π_1 is the first processed item, π_2 the second, etc.), then the total penalty is equal to:

$$F(\pi) = f_{\pi_1}(0) + f_{\pi_2}(t_{\pi_1}) + f_{\pi_3}(t_{\pi_1} + t_{\pi_2}) + \cdots + f_{\pi_n}\left(\sum_{i=1}^{n-1} t_{\pi_i}\right)$$

3.1.1 Solutions for special cases

Linear penalty functions

First we will solve the problem in the case that all penalty functions $f_i(t)$ are linear, i.e. they have the form $f_i(t) = c_i \cdot t$, where c_i is a non-negative number. Note that these functions don't have a constant term. Otherwise we can sum up all constant term, and resolve the problem without them.

Let us fixate some permutation π , and take an index $i = 1 \dots n-1$. Let the permutation π' be equal to the permutation π with the elements i and $i+1$ switched. Let's see how much the penalty changed.

$$F(\pi') - F(\pi) =$$

It is easy to see that the changes only occur in the i -th and $(i+1)$ -th summands:

$$\begin{aligned} &= c_{\pi'_i} \cdot \sum_{k=1}^{i-1} t_{\pi'_k} + c_{\pi'_{i+1}} \cdot \sum_{k=1}^i t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=1}^i t_{\pi_k} \\ &= c_{\pi_{i+1}} \cdot \sum_{k=1}^{i-1} t_{\pi'_k} + c_{\pi_i} \cdot \sum_{k=1}^i t_{\pi'_k} - c_{\pi_i} \cdot \sum_{k=1}^{i-1} t_{\pi_k} - c_{\pi_{i+1}} \cdot \sum_{k=1}^i t_{\pi_k} \\ &= c_{\pi_i} \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i} \end{aligned}$$

It is easy to see, that if the schedule π is optimal, than any change in it leads to an increased penalty (or to the identical penalty), therefore for the optimal schedule we can write down the following condition:

$$c \cdot t_{\pi_{i+1}} - c_{\pi_{i+1}} \cdot t_{\pi_i} \geq 0 \quad \forall i = 1 \dots n-1$$

And after rearranging we get:

$$\frac{c_{\pi_i}}{t_{\pi_i}} \geq \frac{c_{\pi_{i+1}}}{t_{\pi_{i+1}}} \quad \forall i = 1 \dots n-1$$

Thus we obtain the **optimal schedule** by simply **sorting** the jobs by the fraction $\frac{c_i}{t_i}$ in non-ascending order.

It should be noted, that we constructed this algorithm by the so-called **permutation method**: we tried to swap two adjacent elements, calculated how much the penalty changed, and then derived the algorithm for finding the optimal method.

Exponential penalty function

Let the penalty function look like this:

$$f_i(t) = c_i \cdot e^{\alpha t},$$

where all numbers c_i are non-negative and the constant α is positive.

By applying the permutation method, it is easy to determine that the jobs must be sorted in non-ascending order of the value:

$$v_i = \frac{1 - e^{\alpha t_i}}{c_i}$$

Identical monotone penalty function

In this case we consider the case that all $f_i(t)$ are equal, and this function is monotone increasing.

It is obvious that in this case the optimal permutation is to arrange the jobs by non-ascending processing time t_i .

3.1.2 The Livshits-Kladov theorem

The Livshits-Kladov theorem establishes, that the permutation method is only applicable for the above mentioned three cases, i.e.:

- Linear case: $f_i(t) = c_i(t) + d_i$, where c_i are non-negative constants,
- Exponential case: $f_i(t) = c_i \cdot e^{\alpha t} + d_i$, where c_i and α are positive constants,
- Identical case: $f_i(t) = \phi(t)$, where ϕ is a monotone increasing function.

In all other cases the method cannot be applied.

The theorem is proven under the assumption that the penalty functions are sufficiently smooth (the third derivatives exists).

In all three case we apply the permutation method, through which the desired optimal schedule can be found by sorting, hence in $O(n \log n)$ time.

3.2 Scheduling jobs on two machines

This task is about finding an optimal schedule for n jobs on two machines. Every item must first be processed on the first machine, and afterwards on the second one. The i -th job takes a_i time on the first machine, and b_i time on the second machine. Each machine can only process one job at a time.

We want to find the optimal order of the jobs, so that the final processing time is the minimum possible.

This solution that is discussed here is called Johnson's rule (named after S. M. Johnson).

It is worth noting, that the task becomes NP-complete, if we have more than two machines.

3.2.1 Construction of the algorithm

Note first, that we can assume that the order of jobs for the first and the second machine have to coincide. In fact, since the jobs for the second machine become available after processing them at the first, and if there are several jobs available for the second machine, than the processing time will be equal to the sum of their b_i , regardless of their order. Therefore it is only advantageous to send the jobs to the second machine in the same order as we sent them to the first machine.

Consider the order of the jobs, which coincides with their input order $1, 2, \dots, n$.

We denote by x_i the **idle time** of the second machine immediately before processing i . Our goal is to **minimize the total idle time**:

$$F(x) = \sum x_i \rightarrow \min$$

For the first job we have $x_1 = a_1$. For the second job, since it gets sent to the machine at the time $a_1 + a_2$, and the second machine gets free at $x_1 + b_1$, we have $x_2 = \max((a_1 + a_2) - (x_1 + b_1), 0)$. In general we get the equation:

$$x_k = \max\left(\sum_{i=1}^k a_i - \sum_{i=1}^{k-1} b_i - \sum_{i=1}^{k-1} x_i, 0\right)$$

We can now calculate the **total idle time** $F(x)$. It is claimed that that is has the form

$$F(x) = \max_{k=1 \dots n} K_k,$$

where

$$K_k = \sum_{i=1}^k a_i - \sum_{i=1}^{k-1} b_i.$$

This can be easily verified using induction.

We now use the **permutation method**: we will exchange two neighboring jobs j and $j + 1$ and see how this will change the total idle time.

By the form of the expression of K_i , it is clear that only K_j and K_{j+1} change, we denote their new values with K'_j and K'_{j+1} .

If this change from of the jobs j and $j + 1$ increased the total idle time, it has to be the case that:

$$\max(K_j, K_{j+1}) \leq \max(K'_j, K'_{j+1})$$

(Switching two jobs might also have no impact at all. The above condition is only a sufficient one, but not a necessary one.)

After removing $\sum_{i=1}^{j+1} a_i - \sum_{i=1}^{j-1} b_i$ from both sides of the inequality, we get:

$$\max(-a_{j+1}, -b_j) \leq \max(-b_{j+1}, -a_j)$$

And after getting rid of the negative signs:

$$\min(a_j, b_{j+1}) \leq \min(b_j, a_{j+1})$$

Thus we obtained a **comparator**: by sorting the jobs on it, we obtain an optimal order of the jobs, in which no two jobs can be switched with an improvement of the final time.

However you can further **simplify** the sorting, if you look at the comparator from a different angle. The comparator can be interpreted in the following way: If we have the four times $(a_j, a_{j+1}, b_j, b_{j+1})$, and the minimum of them is a time corresponding to the first machine, then the corresponding job should be done first. If the minimum time is a time from the second machine, then it should go later. Thus we can sort the jobs by $\min(a_i, b_i)$, and if the processing time of the current job on the first machine is less then the processing time on the second machine, then this job must be done before all the remaining jobs, and otherwise after all remaining tasks.

One way or another, it turns out that by Johnson's rule we can solve the problem by sorting the jobs, and thus receive a time complexity of $O(n \log n)$.

3.2.2 Implementation

Here we implement the second variation of the described algorithm.

```
struct Job {
    int a, b, idx;

    bool operator<(Job o) const {
        return min(a, b) < min(o.a, o.b);
    }
};

vector<Job> johnsons_rule(vector<Job> jobs) {
    sort(jobs.begin(), jobs.end());
    vector<Job> a, b;
    for (Job j : jobs) {
        if (j.a < j.b)
            a.push_back(j);
        else
            b.push_back(j);
    }
}
```

```
    }
    a.insert(a.end(), b.rbegin(), b.rend());
    return a;
}

pair<int, int> finish_times(vector<Job> const& jobs) {
    int t1 = 0, t2 = 0;
    for (Job j : jobs) {
        t1 += j.a;
        t2 = max(t2, t1) + j.b;
    }
    return make_pair(t1, t2);
}
```

All the information about each job is store in struct. The first function sorts all jobs and computes the optimal schedule. The second function computes the finish times of both machines given a schedule.

3.3 Optimal schedule of jobs given their deadlines and durations

Suppose, we have a set of jobs, and we are aware of every job's deadline and its duration. The execution of a job cannot be interrupted prior to its ending. It is required to create such a schedule to accomplish the biggest number of jobs.

3.3.1 Solving

The algorithm of the solving is **greedy**. Let's sort all the jobs by their deadlines and look at them in descending order. Also, let's create a queue q , in which we'll gradually put the jobs and extract one with the least run-time (for instance, we can use set or priority_queue). Initially, q is empty.

Suppose, we're looking at the i -th job. First of all, let's put it into q . Let's consider the period of time between the deadline of i -th job and the deadline of $i - 1$ -th job. That is the segment of some length T . We will extract jobs from q (in their left duration ascending order) and execute them until the whole segment T is filled. Important: if at any moment of time the extracted job can only be partly executed until segment T is filled, then we execute this job partly just as far as possible, i.e., during the T -time, and we put the remaining part of a job back into q .

On the algorithm's completion we'll choose the optimal solution (or, at least, one of several solutions). The running time of algorithm is $O(n \log n)$.

3.3.2 Implementation

The following function takes a vector of jobs (consisting of a deadline, a duration, and the job's index) and computes a vector containing all indices of the used jobs in the optimal schedule. Notice that you still need to sort these jobs by their deadline, if you want to write down the plan explicitly.

```
struct Job {
    int deadline, duration, idx;

    bool operator<(Job o) const {
        return deadline < o.deadline;
    }
};

vector<int> compute_schedule(vector<Job> jobs) {
    sort(jobs.begin(), jobs.end());

    set<pair<int,int>> s;
    vector<int> schedule;
    for (int i = jobs.size()-1; i >= 0; i--) {
        int t = jobs[i].deadline - (i ? jobs[i-1].deadline : 0);
        s.insert(make_pair(jobs[i].duration, jobs[i].idx));
        while (t && !s.empty()) {
            auto it = s.begin();
```

```
        if (it->first <= t) {
            t -= it->first;
            schedule.push_back(it->second);
        } else {
            s.insert(make_pair(it->first - t, it->second));
            t = 0;
        }
        s.erase(it);
    }
}
return schedule;
}
```

Chapter 4

Miscellaneous

4.1 Josephus Problem

Problem situation - Given the natural numbers n and k . All natural numbers from 1 to n are written in a circle. First count the k -th number starting from the first one and delete it. Then k numbers are counted starting from the next one and the k -th one is removed again, and so on. The process stops when one number remains. It is required to find the last number.

This task was set by **Flavius Josephus** in the 1st century (though in a somewhat narrower formulation: for $k = 2$).

This problem can be solved by modeling the procedure. Brute force modeling will work $O(n^2)$. Using a [segment tree](#) we can improve it to $O(n \log n)$. We want something better though.

4.1.1 Modeling a $O(n)$ solution

We will try to find a pattern expressing the answer for the problem $J_{n,k}$ through the solution of the previous problems.

Using brute force modeling we can construct a table of values, for example, the following:

$n \setminus k$	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 we can clearly see the following **pattern**:

$$J_{n,k} = (J_{(n-1),k} + k - 1) \bmod n + 1$$

$$J_{1,k} = 1$$

Here, 1-indexing somewhat spoils the elegance of the formula; if you number the positions from 0, you get a very clear formula:

$$J_{n,k} = (J_{(n-1),k} + k) \bmod n$$

So, we found a solution to the problem of Joseph, working in $O(n)$ operations. Simple **recursive implementation** (in 1-indexing)

```
int josephus(int n, int k) {
    return n > 1 ? (josephus(n-1, k) + k - 1) % n + 1 : 1;
}
```

Non-recursive form :

```
int josephus(int n, int k) {
    int res = 0;
    for (int i = 1; i <= n; ++i)
        res = (res + k) % i;
    return res + 1;
}
```

This formula can also be found analytically. Again here we assume 0-indexed. After we killed the first person, we have $n - 1$ people left. And when we repeat the procedure then we will start with the person that had originally the index $k \bmod m$. $J_{(n-1),k}$ would be the answer for the remaining circle, if we start counting at 0, but because we actually start with k we have $J_{n,k} = (J_{(n-1),k} + k) \bmod n$.

4.1.2 Modeling a $O(k \log n)$ solution

For relatively small k we can come up with a better solution than the above recursive solution in $O(n)$. If k is a lot smaller than n , then we can kill multiple people ($\lfloor \frac{n}{k} \rfloor$) in one run without looping over. Afterwards we have $n - \lfloor \frac{n}{k} \rfloor$ people left, and we start with the $(\lfloor \frac{n}{k} \rfloor \cdot n)$ -th person. So we have to shift by that many. We can notice that $\lfloor \frac{n}{k} \rfloor \cdot n$ is simply $n \bmod k$. And since we removed every k -th person, we have to add the number of people that we removed before the result index.

Also, we need to handle the case when n becomes less than k - in this case, the above optimization would degenerate into an infinite loop.

Implementation (for convenience in 0-indexing):

```
int josephus(int n, int k) {
    if (n == 1)
        return 0;
    if (k == 1)
        return n - 1;
    return (josephus(n - \frac{n}{k}, k) + n % k) % n;
}
```

```

        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;
}

```

Let us estimate the **complexity** of this algorithm. Immediately note that the case $n < k$ is analyzed by the old solution, which will work in this case for $O(k)$. Now consider the algorithm itself. In fact, on each iteration of it, instead of 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 roughly from the following equation:

$$n \left(1 - \frac{1}{k}\right)^x = 1,$$

on taking logarithm, we obtain:

$$\ln n + x \ln \left(1 - \frac{1}{k}\right) = 0,$$

$$x = -\frac{\ln n}{\ln \left(1 - \frac{1}{k}\right)},$$

using the decomposition of the logarithm into Taylor series, we obtain an approximate estimate:

$$x \approx k \ln n$$

Thus, the complexity of the algorithm is actually $O(k \log n)$.

4.1.3 Analytical solution for $k = 2$

In this particular case (in which this task was set by Josephus Flavius) the problem is solved much easier.

In the case of even n we get that all even numbers will be crossed out, and then there will be a problem remaining for $\frac{n}{2}$, then the answer for n will be obtained from the answer for $\frac{n}{2}$ by multiplying by two and subtracting one (by shifting positions):

$$J_{2n,2} = 2J_{n,2} - 1$$

Similarly, in the case of an odd n , all even numbers will be crossed out, then the first number, and the problem for $\frac{n-1}{2}$ will remain, and taking into account the shift of positions, we obtain the second formula:

$$J_{2n+1,2} = 2J_{n,2} + 1$$

We can use this recurrent dependency directly in our implementation. This pattern can be translated into another form: $J_{n,2}$ represents a sequence of all odd numbers, “restarting” from one whenever n turns out to be a power of two. This can be written as a single formula:

$$J_{n,2} = 1 + 2 \left(n - 2^{\lfloor \log_2 n \rfloor} \right)$$

4.1.4 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 analytical representation of the solution of Joseph’s problem has not yet been found. For small k , some formulas are derived, but apparently they are all difficult to apply in practice (for example, see Halbeisen, Hungerbuhler “the Josephus Problem” and Odlyzko, Wilf “Functional iteration and the Josephus problem”).

4.2 15 Puzzle Game: Existence Of The Solution

This game is played on a 4×4 board. On this board there are 15 playing tiles numbered from 1 to 15. One cell is left empty (denoted by 0). You need to get the board to the position presented below by repeatedly moving one of the tiles to the free space:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

The game “15 Puzzle” was created by Noyes Chapman in 1880.

4.2.1 Existence Of The Solution

Let’s consider this problem: given position on the board, determine whether a sequence of moves which leads to a solution exists.

Suppose we have some position on the board:

a_1	a_2	a_3	a_4
a_5	a_6	a_7	a_8
a_9	a_{10}	a_{11}	a_{12}
a_{13}	a_{14}	a_{15}	a_{16}

where one of the elements equals zero and indicates an empty cell $a_z = 0$

Let’s consider the permutation:

$$a_1 a_2 \dots a_{z-1} a_{z+1} \dots a_{15} a_{16}$$

(i.e. the permutation of numbers corresponding to the position on the board without a zero element)

Let N be the number of inversions in this permutation (i.e. the number of such elements a_i and a_j that $i < j$, but $a_i > a_j$).

Suppose K is an index of a row where the empty element is located (i.e. in our indications $K = (z - 1) \text{ div } 4 + 1$).

Then, **the solution exists iff $N + K$ is even.**

4.2.2 Implementation

The algorithm above can be illustrated with the following program 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");
```

4.2.3 Proof

In 1879 Johnson proved that if $N + K$ is odd, then the solution doesn't exist, and in the same year Story proved that all positions when $N + K$ is even have a solution.

However, all these proofs were quite complex.

In 1999 Archer proposed a much simpler proof (you can download his article [here](#)).

4.2.4 Practice Problems

- [Hackerrank - N-puzzle](#)

4.3 The Stern-Brocot tree and Farey sequences

4.3.1 Stern-Brocot tree

The Stern-Brocot tree is an elegant construction to represent the set of all positive fractions. It was independently discovered by German mathematician Moritz Stern in 1858 and by French watchmaker Achille Brocot in 1861. However, some sources attribute the discovery to ancient Greek mathematician Eratosthenes.

The construction starts at the zeroth iteration with the two fractions

$$\frac{0}{1}, \frac{1}{0}$$

where it should be noted that the second quantity is not strictly a fraction, but it can be interpreted as an irreducible fraction representing infinity.

At every subsequent iteration, consider all adjacent fractions $\frac{a}{b}$ and $\frac{c}{d}$ and insert their mediant $\frac{a+c}{b+d}$ between them.

The first few iterations look like this:

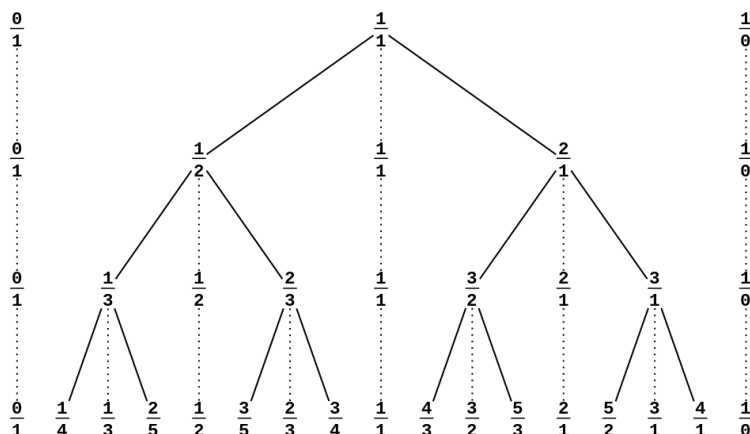
$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{2}{2}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

Continuing this process to infinity this covers *all* positive fractions. Additionally, all fractions will be *unique* and *irreducible*. Finally, the fractions will also appear in ascending order.

Before proving these properties, let us actually show a visualization of the Stern-Brocot tree, rather than the list representation. Every fraction in the tree has two children. Each child is the mediant of the closest ancestor on the left and closest ancestor to the right.



4.3.2 Proofs

Ordering. Proving ordering is simple. We note that the mediant of two fractions is always in-between the fractions

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$$

given that

$$\frac{a}{b} \leq \frac{c}{d}.$$

The two inequalities can be easily shown by rewriting the fractions with common denominators.

As the ordering is ascending in the zeroth iteration, it will be maintained at every subsequent iteration.

Irreducibility. To prove this we will show that for any two adjacent fractions $\frac{a}{b}$ and $\frac{c}{d}$ we have that

$$bc - ad = 1.$$

Recall that a Diophantine equation with two variables $ax + by = c$ has a solution iff c is a multiple of $\gcd(a, b)$. In our case this implies that $\gcd(a, b) = \gcd(c, d) = 1$, which is what we want to show.

Clearly at the zeroth iteration $bc - ad = 1$. What remains to be shown is that mediants retain this property.

Assume our two adjacent fractions uphold $bc - ad = 1$, after the mediant is added to the list

$$\frac{a}{b}, \frac{a+c}{b+d}, \frac{c}{d}$$

the new expressions become

$$\begin{aligned} b(a+c) - a(b+d) &= 1 \\ c(b+d) - d(a+c) &= 1 \end{aligned}$$

which, using that $bc - ad = 1$, can be easily shown to be true.

From this we see that the property is always maintained and thus all fractions are irreducible.

The presence of all fractions. This proof is closely related to locating a fraction in the Stern-Brocot tree. From the ordering property we have that left subtree of a fraction contains only fractions smaller than the parent fraction, and the right subtree contains only fractions larger than the parent fraction. This means we can search for a fraction by traversing the tree from the root, going left if the target is smaller than the fraction and going right if the target is larger.

Pick an arbitrary positive target fraction $\frac{x}{y}$. It is obviously between $\frac{0}{1}$ and $\frac{1}{0}$, so the only way for the fraction to not be in the tree is if it takes an infinite number of steps to get to it.

If that is the case we would at all iterations have

$$\frac{a}{b} < \frac{x}{y} < \frac{c}{d}$$

which (using the fact that an integer $z > 0 \iff z \geq 1$) can be rewritten as

$$\begin{aligned} bx - ay &\geq 1 \\ cy - dx &\geq 1. \end{aligned}$$

Now multiply the first inequality by $c + d$ and the second with $a + b$ and add them to get

$$(c + d)(bx - ay) + (a + b)(cy - dx) \geq a + b + c + d.$$

Expanding this and using the previously shown property $bc - ad = 1$ we get that

$$x + y \geq a + b + c + d.$$

And given that at every iteration at least one of a, b, c, d will increase, the fraction searching process will contain no more than $x + y$ iterations. This contradicts the assumption that the path to $\frac{x}{y}$ was infinite and hence $\frac{x}{y}$ must be part of the tree.

4.3.3 Tree Building Algorithm

To build any subtree of the Stern-Brocot tree it suffices to know the left and right ancestor. On the first level the left and right ancestor is $\frac{0}{1}$ and $\frac{1}{0}$ respectively. Using these we calculate the mediant and proceed one level deeper, with the median replacing the right ancestor in the left subtree, and vice versa.

This pseudocode tries to build the entire 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;

    ... output the current fraction x/y at the current level in the tree

    build(a, b, x, y, level + 1);
    build(x, y, c, d, level + 1);
}
```

4.3.4 Fraction Search Algorithm

The search algorithm was already described in the proof that all fractions appear in the tree, but we will repeat it here. The algorithm is a binary search algorithm. Initially we stand at the root of the tree and we compare our target with the current fraction. If they are the same we are done and stop the process. If our target is smaller we move to the left child, otherwise we move to the right child.

Here is an implementation that returns the path to a given fraction $\frac{x}{y}$ as a sequence of 'L' and 'R' characters, meaning traversal to the left and right child respectively. This sequence of characters uniquely defines all positive fractions and is called the Stern-Brocot number system.

```
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 the Stern-Brocot number system corresponds to infinite sequences of characters. Along the endless path towards the irrational number the algorithm will find reduced fractions with gradually increasing denominators that provides increasingly better approximations of the irrational number. So by taking a prefix of the infinite sequence approximations with any desired precision can be achieved. This application is important in watch-making, which explains why the tree was discovered in that domain.

4.3.5 Farey Sequence

The Farey sequence or order n is the sorted sequence of fractions between 0 and 1 whose denominators do not exceed n .

The sequences are named after English geologist John Farey, who in 1816 conjectured that any fraction in a Farey sequence is the mediant of its neighbors. This was proven some time later by Cauchy, but independent of both of them the mathematician Haros had come to almost the same conclusion in 1802.

The Farey sequences has many interesting properties on their own, but the connection to Stern-Brocot tree is the most obvious. In fact, the Farey sequence can be obtained by trimming branches from the tree.

From the algorithm for building the Stern-Brocot tree we get an algorithm for Farey sequences. Start with the list of fractions $\frac{0}{1}, \frac{1}{0}$. At every subsequent iteration insert the mediant only if the denominator does not exceed n . At some point the list will stop changing and the desired Farey sequence has been found.

Length of a Farey Sequence

A Farey sequence of order n contains all elements of the Farey sequence of order $n - 1$ as well as all irreducible fractions with denominator n , but the latter is just the totient $\varphi(n)$. So the length L_n of the Farey sequence of order n is

$$L_n = L_{n-1} + \varphi(n)$$

or equivalently by unraveling the recursion we get

$$L_n = 1 + \sum_{k=1}^n \varphi(k).$$