C++ contest library

Miska Kananen

May 13, 2018

Contents

1 Environment and workflow

1.1 Compilation script

```
#!/bin/bash

g++ $1 -o ${1%.*} -std=c++11 -Wall -Wextra -Wshadow -
    ftrapv -Wfloat-equal -Wconversion -Wlogical-op -
    Wshift-overflow=2 -fsanitize=address -fsanitize=
    undefined -fno-sanitize-recover
```

1.2 Stress testing

`srand(time(NULL));` changes seed only once a second and is unsuitable for stress testing. RNG seed initialization (requires x86 and g++):

```
#include <iostream>
#include <cstdlib>

using namespace std;
```

```
int main() {
	ll seed;
	asm("rdtsc" : "=A"(seed));
	srand(seed);
}
```

Shell script for stress testing with a brute force solution and a test generator:

```
for i in {1..1000}
do
	./gen $i 100000 1000000000 > test_input
	./brute < test_input > corr_output
	./tested < test_input > user_output
	diff corr_output user_output > /dev/null
	res=$?

	if [ $res -ne 0 ]; then
		echo "Wrong answer"
		echo "Test input:"
		cat test_input
		echo ""
		echo "Correct output:"
		cat corr_output
		echo ""
		echo "User output:"
		cat user_output
	fi

	rm test_input
	rm corr_output
	rm user_output

	if [ $res -ne 0 ]; then
		exit 1
	fi
done
```

2 General techniques

2.1 Bit tricks

g++ builtin functions:

- `__builtin_clz(x)`: number of zeros in the beginning
- `__builtin_ctz(x)`: number of zeros in the end
- `__builtin_popcount(x)`: number of set bits
- `__builtin_parity(x)`: parity of number of ones

There are separate functions of form `__builtin_clzll(x)` for 64-bit integers. For the compiler to utilize the native `POPCNT` instruction, `#pragma GCC target("sse4.2")` should be used.

Iterate subsets of set s:

```
int cs = 0;
do {
	// process subset cs
} while(cs=(cs-s)&s);
```

Get lowest 1-bit:

```
int lsone = x&(-x);
```

2.2 Mo's algorithm

Processes range queries on an array offline in $O(n\sqrt{n}\, f(n))$, where the array has n elements, there are n queries and addition/removal of an element to/from the active set takes $O(f(n))$ time.

The array is divided into $\sqrt{n}$ blocks of $k = \sqrt{n}$ elements. Queries are sorted such that query $[a_i, b_i]$ goes before $[a_j, b_j]$ if:

1. $\lfloor \frac{a_i}{k} \rfloor < \lfloor \frac{a_j}{k} \rfloor$ or
2. $\lfloor \frac{a_i}{k} \rfloor = \lfloor \frac{a_j}{k} \rfloor$ and $b_i < b_j$

Active range is maintained between queries and the endpoints of the range are moved accordingly. Both endpoints move $O(n\sqrt{n})$ steps in total during the algorithm.

2.3 Arbitrary precision decimals

Python 3 implements arbitrary precision decimal arithmetic in module `decimal`. All decimal numbers are represented exactly and the precision is user-definable.

```python
from decimal import *

a, b = [Decimal(x) for x in input().split(" ")]
getcontext().prec = 50 # set precision

print(a/b)
```

2.4 g++ pragmas

Pragmas optimize all functions defined afterwards. They should be located in the very beginning of the source code, even before includes in order to optimize imported standard library code.

`#pragma GCC optimize("O3")`

`#pragma GCC optimize("Ofast")`, enables more optimizations but isn't always faster.

`#pragma GCC optimize("unroll-loops")`

`#pragma GCC target("arch=skylake")`

`#pragma GCC target("mmx,sse,sse2,sse3, ssse3,sse4.2,popcnt,avx,tune=native")` for ivybridge if `arch=ivybridge` fails.

All possible target architectures are listed in compiler report if an invalid architecture is given to `arch`. Supported Intel Core generations in order: nehalem, sandybridge, ivybridge (for CF), haswell (first avx2), broadwell, skylake.

3 Data structures

3.1 Lazy segment tree

Implements range add and range sum query in $O(log(n))$. 0-indexed.

```cpp
#include <iostream>

using namespace std;
typedef long long ll;

const int N = (1<<18); // segtree max size

ll st[2*N]; // segtree values
ll lz[2*N]; // lazy updates
bool haslz[2*N]; // does a node have a lazy update
    pending

void push(int s, int l, int r) {
        if (haslz[s]) {
                st[s] += (r-l+1)*lz[s]; // change
                    operator+logic

                if (l != r) {
                        lz[2*s] += lz[s]; // change
                            operator
                        lz[2*s+1] += lz[s]; // change
                            operator
                        haslz[2*s] = true;
                        haslz[2*s+1] = true;
                }

                lz[s] = 0; // set to identity
                haslz[s] = false;
        }
}

ll kysy(int ql, int qr, int s = 1, int l = 0, int r = N
    -1) {
        push(s, l, r);
        if (l > qr || r < ql) {
                return 0; // set to identity
        }
        if (ql <= l && r <= qr) {
                return st[s];
        }

        int mid = (l+r)/2;
        ll res = 0; // set to identity
        res += kysy(ql, qr, 2*s, l, mid); // change
            operator
```

```
        res += kysy(ql, qr, 2*s+1, mid+1, r); // change
            operator
        return res;
}

void muuta(int ql, int qr, ll x, int s = 1, int l = 0,
    int r = N-1) {
        push(s, l, r);
        if (l > qr || r < ql) {
                return;
        }
        if (ql <= l && r <= qr) {
                lz[s] += x; // change operator
                haslz[s] = true;
                return;
        }

        int mid = (l+r)/2;
        muuta(ql, qr, x, 2*s, l, mid);
        muuta(ql, qr, x, 2*s+1, mid+1, r);

        st[s] = st[2*s] + st[2*s+1]; // change operator
        if (haslz[2*s]) {
                st[s] += (mid-l+1)*lz[2*s]; // change
                    operator+logic
        }
        if (haslz[2*s+1]) {
                st[s] += (r-(mid+1)+1)*lz[2*s+1]; //
                    change operator+logic
        }
}

void build(int s = 1, int l = 0, int r = N-1) {
        if (r-l > 1) {
                int mid = (l+r)/2;
                build(2*s, l, mid);
                build(2*s+1, mid+1, r);
        }
        st[s] = st[2*s]+st[2*s+1]; // change operator
}

/*
        TESTED, correct
        Allowed indices 0..N-1
        2 types of queries: range add and range sum
*/
int main() {
        for (int i = 1; i <= n; ++i) {
                cin >> st[i+N];
        }
        build();
}
```

3.2 Sparse segment tree

Implements point update and range sum query in $O(log(n))$. Memory usage is around 40 MB with a range of $2^{30} = 10^9$ after 10^5 random operations. 0-indexed.

```
#include <iostream>

using namespace std;
typedef long long ll;

const int N = 1<<30; // max element index

struct node {
    ll s;
    int x, y;
    node *l, *r;
    node (int cs, int cx, int cy) : s(cs), x(cx), y(cy)
        {
        l = nullptr;
        r = nullptr;
    }
};

node *st = new node(0, 0, N); // segtree root node

void update(int k, ll val, node *nd = st) {
    if (nd->x == nd->y) {
        nd->s += val; // change operator
    }
    else {
        int mid = (nd->x + nd->y)/2;
        if (nd->x <= k && k <= mid) {
            if (nd->l == nullptr) nd->l = new node(0, nd
                ->x, mid);
```

```
            update(k, val, nd->l);
        }
        else if (mid < k && k <= nd->y) {
            if (nd->r == nullptr) nd->r = new node(0,
                mid+1, nd->y);
            update(k, val, nd->r);
        }
        ll ns = 0; // set to identity
        if (nd->l != nullptr) ns += (nd->l)->s; //
            change operator
        if (nd->r != nullptr) ns += (nd->r)->s; //
            change operator
        nd->s = ns;
    }
}

ll query(int ql, int qr, node *nd = st) {
    if (ql <= nd->x && nd->y <= qr) return nd->s;
    if (nd->y < ql || nd->x > qr) return 0; // set to
        identity
    ll res = 0; // set to identity
    if (nd->l != nullptr) res += query(ql, qr, nd->l);
        // change operator
    if (nd->r != nullptr) res += query(ql, qr, nd->r);
        // change operator
    return res;
}
```

3.3 2D segment tree

Implements point update and subgrid query in $O(log^2(n))$. Grid is 0-indexed.

```
#include <iostream>

using namespace std;
typedef long long ll;

const int N = 1<<11;

int n, q;

ll st[2*N][2*N];

// calculate subgrid sum from {y1, x1} to {y2, x2}
// 0-indexed
ll summa(int y1, int x1, int y2, int x2) {
    y1 += N;
    x1 += N;
    y2 += N;
    x2 += N;

    ll sum = 0;

    while (y1 <= y2) {
        if (y1%2 == 1) {
            int nx1 = x1;
            int nx2 = x2;
            while (nx1 <= nx2) {
                if (nx1%2 == 1) sum += st[y1][nx1++];
                if (nx2%2 == 0) sum += st[y1][nx2--];
                nx1 /= 2;
                nx2 /= 2;
            }
            y1++;
        }
        if (y2%2 == 0) {
            int nx1 = x1;
            int nx2 = x2;
            while (nx1 <= nx2) {
                if (nx1%2 == 1) sum += st[y2][nx1++];
                if (nx2%2 == 0) sum += st[y2][nx2--];
                nx1 /= 2;
                nx2 /= 2;
            }
            y2--;
        }
        y1 /= 2;
        y2 /= 2;
    }
    return sum;
}

// set {y, x} to u
// 0-indexed
void muuta(int y, int x, ll u) {
    y += N;
    x += N;
    st[y][x] = u;
```

```
    for (int nx = x/2; nx >= 1; nx /= 2) {
        st[y][nx] = st[y][2*nx]+st[y][2*nx+1];
    }

    for (y /= 2; y >= 1; y /= 2) {
        for (int nx = x; nx >= 1; nx /= 2) {
            st[y][nx] = st[2*y][nx]+st[2*y+1][nx];
        }
    }
}
```

3.4 Treap

Implements split, merge, kth element, range update and range reverse in $O(log(n))$. Range update adds a value to every element in a subarray. Treap is 1-indexed.

Note: Memory management tools warn of about 30 MB memory leak for 500 000 elements. This is because nodes are not deleted when exiting program and is irrelevant in a competition. Deleting nodes would slow the treap down by a factor of 3.

```
#include <iostream>
#include <cstdlib>
#include <algorithm>

using namespace std;
typedef long long ll;

struct node {
        ll val; // change data type (char, integer...)
        int prio, size;
        bool lzinv;
        ll lzupd;
        bool haslz;
        node *left, *right;

        node(ll v) {
                val = v;
                prio = rand();
                size = 1;
                lzinv = false;
                lzupd = 0;
                haslz = false;
                left = nullptr;
                right = nullptr;
        }
};

int gsize(node *s) {
        if (s == nullptr) return 0;
        return s->size;
}

void upd(node *s) {
        if (s == nullptr) return;
        s->size = gsize(s->left) + 1 + gsize(s->right);
}

void push(node *s) {
        if (s == nullptr) return;

        if (s->haslz) {
                s->val += s->lzupd; // operator
        }
        if (s->lzinv) {
                swap(s->left, s->right);
        }

        if (s->left != nullptr) {
                if (s->haslz) {
                        s->left->lzupd += s->lzupd; //
                            operator
                        s->left->haslz = true;
                }
                if (s->lzinv) {
                        s->left->lzinv = !s->left->lzinv
                            ;
                }
        }
        if (s->right != nullptr) {
                if (s->haslz) {
                        s->right->lzupd += s->lzupd; //
                            operator
                        s->right->haslz = true;
                }
                if (s->lzinv)  {
                        s->right->lzinv = !s->right->
```

```
                        lzinv;
                }
        }

        s->lzupd = 0; // operator identity value
        s->lzinv = false;
        s->haslz = false;
}

// split a treap into two treaps, size of left treap = k
void split(node *t, node *&l, node *&r, int k) {
        push(t);
        if (t == nullptr) {
                l = nullptr;
                r = nullptr;
                return;
        }
        if (k >= gsize(t->left)+1) {
                split(t->right, t->right, r, k-(gsize(t
                    ->left)+1));
                l = t;
        }
        else {
                split(t->left, l, t->left, k);
                r = t;
        }
        upd(t);
}

// merge two treaps
void merge(node *&t, node *l, node *r) {
        push(l);
        push(r);
        if (l == nullptr) t = r;
        else if (r == nullptr) t = l;
        else {
                if (l->prio >= r->prio) {
                        merge(l->right, l->right, r);
                        t = l;
                }
                else {
                        merge(r->left, l, r->left);
                        t = r;
                }
        }
        upd(t);
}

// get k:th element in array (1-indexed)
ll kthElem(node *t, int k) {
        push(t);
        int cval = gsize(t->left)+1;
        if (k == cval) return t->val;
        if (k < cval) return kthElem(t->left, k);
        return kthElem(t->right, k-cval);
}

// do a lazy update on subarray [a..b]
void rangeUpd(node *&t, int a, int b, ll x) {
        node *cl, *cur, *cr;
        int tsz = gsize(t);
        bool lsplit = false;
        bool rsplit = false;
        cur = t;
        if (a > 1) {
                split(cur, cl, cur, a-1);
                lsplit = true;
        }
        if (b < tsz) {
                split(cur, cur, cr, b-a+1);
                rsplit = true;
        }
        cur->lzupd += x; // operator
        cur->haslz = true;
        if (lsplit) {
                merge(cur, cl, cur);
        }
        if (rsplit) {
                merge(cur, cur, cr);
        }
        t = cur;
}

// reverse subarray [a..b]
void rangeInv(node *&t, int a, int b) {
        node *cl, *cur, *cr;
        int tsz = gsize(t);
        bool lsplit = false;
        bool rsplit = false;
        cur = t;
```

```
        if (a > 1) {
                split(cur, cl, cur, a-1);
                lsplit = true;
        }
        if (b < tsz) {
                split(cur, cur, cr, b-a+1);
                rsplit = true;
        }
        cur->lzinv = !cur->lzinv;
        if (lsplit) {
                merge(cur, cl, cur);
        }
        if (rsplit) {
                merge(cur, cur, cr);
        }
        t = cur;
}

int n;

// TESTED, correct
int main() {
        cin >> n;
        node *tree = nullptr;
        for (int i = 1; i <= n; ++i) {
                node *nw = new node(0);
                merge(tree, tree, nw); // treap
                    construction
        }
}
```

3.5 Sparse table

Implements range minimum/maximum query in $O(1)$ with $O(n\ log(n))$ preprocessing. 0-indexed.

```
#include <iostream>
#include <cmath>

using namespace std;
typedef long long ll;

int n, q;
ll t[100005];
ll st[18][100005];

ll rmq(int a, int b) {
        int l = b-a+1;
        int k = (int)log2(l);
        return min(t[st[k][a]], t[st[k][a+(l-(1<<k))]]);
            // change function
}

// TESTED, correct
// n elements, q queries of form rmq(a, b) (0 <= a <= b
    <= n-1)
int main() {
        cin >> n >> q;
        for (int i = 0; i < n; ++i) cin >> t[i];

        // build sparse table
        for (int i = 0; i < n; ++i) st[0][i] = i;
        for (int j = 1; (1<<j) <= n; ++j) {
                for (int i = 0; i + (1<<j) <= n; ++i) {
                        ll a = st[j-1][i];
                        ll b = st[j-1][i+(1<<(j-1))];
                        if (t[a] <= t[b]) st[j][i] = a;
                            // change operator
                        else st[j][i] = b;
                }
        }
}
```

3.6 Indexed set (policy-based data structures)

Works like `std::set` but adds support for indices. Set is 0-indexed. Requires g++. Has two additional functions:

1. `find_by_order(x)`: return an iterator to element at index x
2. `order_of_key(x)`: return the index that element x has or would have in the set, depending on if it exists

Both functions work in $O(log(n))$.

Changing `less` to `less_equal` makes the set work like multiset. However, elements can't be removed.

```
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>

using namespace std;
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> indexed_set;

indexed_set s;

int main() {
        s.insert(2);
        s.insert(4);
        s.insert(5);

        auto x = s.find_by_order(1);
        cout << *x << "\n"; // prints 4

        cout << s.order_of_key(5) << "\n"; // prints 2
        cout << s.order_of_key(3) << "\n"; // prints 1
        return 0;
}
```

3.7 Union-find

Uses path compression, `id(x)` has amortized time complexity $O(a^{-1}(n))$ where a^{-1} is inverse Ackermann function.

```
#include <iostream>
#include <algorithm>

using namespace std;

int k[100005];
int s[100005];

int id(int x) {
        int tx = x;
        while (k[x] != x) x = k[x];
        return k[tx] = x;
}

bool equal(int a, int b) {
        return id(a) == id(b);
}

void join(int a, int b) {
        a = id(a);
        b = id(b);
        if (s[b] > s[a]) swap(a, b);
        s[a] += s[b];
        k[b] = a;
}

int n;

int main() {
        for (int i = 0; i < n; ++i) {
                k[i] = i;
                s[i] = 1;
        }
}
```

4 Mathematics

4.1 Number theory

- Prime factorization of n: $p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$
- Number of factors: $\tau(n) = \prod_{i=1}^{k} (\alpha_i + 1)$
- Sum of factors: $\sigma(n) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1}-1}{p_i-1}$
- Product of factors: $\mu(n) = n^{\tau(n)/2}$

Euler's totient function $\varphi(n)$ $(1, 1, 2, 2, 4, 2, 6, 4, 6, 4, \dots)$: counts numbers coprime with n in range $1 \dots n$

$$\varphi(n) = \begin{cases} n-1 & \text{if } n \text{ is prime} \\ \prod_{i=1}^{k} p_i^{a_i-1}(p_i - 1) & \text{otherwise} \end{cases}$$

Fermat's theorem: $x^{m-1} \mod m = 1$ when m is prime and x and m are coprime. It follows that $x^k \mod m = x^{k \mod (m-1)} \mod m$.

Modular inverse $x^{-1} = x^{\varphi(m)-1}$. If m is prime, $x^{-1} = x^{m-2}$. Inverse exists if and only if x and m are coprime.

4.2 Combinatorics

Binomial coefficients:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

Catalan numbers $(1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796 \dots)$:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Classic examples of Catalan numbers: number of balanced pairs of parentheses, number of mountain ranges (n upstrokes and n downstrokes all staying above the original line), number of paths from upper left corner to lower right corner staying above the main diagonal in a $n \times n$ square, ways to triangulate a $n+2$ sided regular polygon, ways to shake hands between $2n$ people in a circle such that no arms cross, number of rooted binary trees with n nodes that have 2 children, number of rooted trees with n edges, number of permutations of $1 \dots n$ that don't have an increasing subsequence of length 3.

Number of derangements (no element stays in original place) of $1, 2, \dots, n$ $(1, 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961, \dots)$:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}$$

Stirling numbers of the second kind $\left\{ {n \atop k} \right\}$: number of ways to partition a set of n objects into k non-empty subsets.

$$1$$
$$0, 1$$
$$0, 1, 1$$
$$0, 1, 3, 1$$
$$0, 1, 7, 6, 1$$
$$0, 1, 15, 25, 10, 1$$
$$0, 1, 31, 90, 65, 15, 1$$

$$\left\{ {n+1 \atop k} \right\} = k\left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\} \quad (k > 0)$$

$$\left\{ {0 \atop 0} \right\} = 1, \left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} = 0 \quad (n > 0)$$

4.3 Matrices

Matrix $A = a \times n$, matrix $B = n \times b$. Matrix multiplication:

$$AB[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]$$

Let linear recurrence $f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k)$ with initial values $f(0), f(1), \dots, f(k-1)$. $c_1, c_2, \dots, c_n$ are constants.

Transition matrix X:

$$X = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ c_k & c_{k-1} & c_{k-2} & \dots & c_1 \end{pmatrix}$$

Now $f(n)$ can be calculated in $O(k^3 log(n))$:

$$\begin{pmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{pmatrix} = X^n \cdot \begin{pmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{pmatrix}$$

```
#include <iostream>
#include <cstring>

using namespace std;
typedef long long ll;

const int N = 2; // matrix size
const ll M = 1000000007; // modulo

struct matrix {
    ll m[N][N];
    matrix() {
        memset(m, 0, sizeof m);
    }
    matrix operator * (matrix b) {
        matrix c = matrix();
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                for (int k = 0; k < N; ++k) {
                    c.m[i][j] = (c.m[i][j] + m[i][k] * b
                        .m[k][j])%M;
                }
        return c;
    }
    matrix unit() {
        matrix a = matrix();
        for (int i = 0; i < N; ++i) a.m[i][i] = 1;
        return a;
    }
};

matrix p(matrix a, ll e) {
    if (e == 0) return a.unit();
    if (e%2 == 0) {
        matrix h = p(a, e/2);
        return h*h;
    }
    return (p(a, e-1)*a);
}

ll n;

// prints nth Fibonacci number mod M
int main() {
    cin >> n;
    matrix x = matrix();
    x.m[0][1] = 1;
    x.m[1][0] = 1;
    x.m[1][1] = 1;
    x = p(x, n);
    cout << x.m[0][1] << "\n";
    return 0;
}
```

4.4 Summations and progressions

- Sum of naturals: $\sum_{i=1}^{n} x = \frac{n(n+1)}{2}$
- Sum of squares: $\sum_{i=1}^{n} x^2 = \frac{n(n+1)(n+2)}{6}$
- Arithmetic progression: $a + \cdots + b = \frac{n(a+b)}{2}$, where n is the number of terms, a is the first term and b is the last term
- Geometric progression: $a + ar + ar^2 + \cdots + ar^{n-1} = a\frac{1-r^n}{1-r}$, where n is the number of terms, a is the first term and $r (r \neq 1)$ is the ratio between two successive terms
  - If $r = 1$, sum is na
  - Also $a + ar + ar^2 + \cdots + b = \frac{a-br}{1-r}$, where a is the first term, b is the last term and r is the ratio between two successive terms

Terms of sum $S = \sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor$ get at most $O(\sqrt{n})$ distinct values. All terms and their counts can be found as follows in $O(\sqrt{n})$:

```cpp
#include <iostream>
#include <vector>

using namespace std;
typedef long long ll;

ll n;

int main() {
        cin >> n;
        vector<ll> v;
        ll x = 0;
        for (ll i = 1; i <= n; i = x+1) {
                x = n/(n/i); // iterate all possible
                    values of floor(n/i) in increasing
                    order
                v.push_back(x);
        }
        for (int i = 0; i < v.size(); ++i) {
                // current value of floor(n/i)
                ll cx = v[i];
                // smallest i for which floor(n/i) == cx
                ll imin = (i == v.size()-1 ? 1 : n/v[i
                    +1] + 1);
                // largest i for which floor(n/i) == cx
                ll imax = n/cx;
        }
        return 0;
}
```

4.5 Miller-Rabin

Deterministic primality test for all 64-bit integers. Requires `__int128` support to test over 32-bit integers.

```cpp
#include <iostream>

using namespace std;
typedef long long ll;
typedef __int128 lll;

// required bases to make test deterministic for 64-bit
    integers
ll mrb[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    37};

lll modpow(lll k, lll e, lll m) {
        if (e == 0) return 1;
        if (e == 1) return k;
        if (e%2 == 0) {
                lll h = modpow(k, e/2, m)%m;
                return (h*h)%m;
        }
        return (k*modpow(k, e-1, m))%m;
}

bool witness(ll a, ll x, ll u, ll t) {
        lll cx = modpow(a, u, x);
        for (int i = 1; i <= t; ++i) {
                lll nx = (cx*cx)%x;
                if (nx == 1 && cx != 1 && cx != (x-1))
                    return true;
                cx = nx;
        }
        return (cx != 1);
}

// TESTED, correct
// determines if x is prime
// deterministic for all 64-bit integers
bool miller_rabin(ll x) {
        if (x == 2) return true;
        if (x < 2 || x%2 == 0) return false;

        ll u = x-1;
        ll t = 0;
        while (u%2 == 0) {
                u /= 2;
                t++;
        }

        for (int i = 0; i < 12; ++i) {
                if (mrb[i] >= x-1) break;
                if (witness(mrb[i], x, u, t)) return
                    false;
        }
        return true;
}
```

4.6 Pollard-Rho

Finds a prime factor of x in $O(\sqrt[4]{x})$. Requires `__int128` support to factor over 32-bit integers.

If x is prime, algorithm might not terminate or it might return 1. Primality must be checked separately.

```
#include <iostream>
#include <cstdlib>
#include <algorithm>

using namespace std;

typedef long long ll;
typedef __int128 lll;

ll n;

ll f(lll x) {
    return (x*x+1)%n;
}

ll gcd(ll a, ll b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}

// return a prime factor of a
// st is a starting seed for pseudorandom numbers, start
     with 2, if algorithm fails (returns -1), increment
    seed
ll pollardrho(ll a, ll st) {
    if (n%2 == 0) return 2;

    ll x = st, y = st, d = 1;
    while (d == 1) {
        x = f(x);
        y = f(f(y));
        d = gcd(abs(x-y), a);
        if (d == a) return -1;
    }
    return d;
}

/*
        TESTED, correct.
    Finds a prime factor of n in O(root_4(n))
    If n is prime, alg might not terminate or it might
        return 1. Check for primality.
*/
int main() {
    cin >> n;
    ll fa = -1;
    ll st = 2;
    while (fa == -1) {
        fa = pollardrho(n, st++);
    }
    cout << min(fa, n/fa) << "␣" << max(fa, n/fa) << "\n
        ";
    return 0;
}
```

5 Geometry

```
#include <iostream>
#include <complex>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;
typedef long double ct; // coordinate type
typedef complex<ct> point;
#define X real()
#define Y imag()
#define F first
#define S second

const ct EPS = 0.000001; // 1e-6
const ct PI = 3.14159265359;

// floating-point equality comparison
bool equal(ct a, ct b) {
        return abs(a-b) < EPS;
}

// point equality comparison
```

```
bool equal(point a, point b) {
        return (equal(a.X, b.X) && equal(a.Y, b.Y));
}

// comparer for sorting points
// check if a < b
bool point_comp(point a, point b) {
        if (equal(a.X, b.X)) {
                return a.Y < b.Y;
        }
        return a.X < b.X;
}

struct line {
        point first, second;

        line(point a, point b) {
                if (point_comp(b, a)) swap(a, b);
                first = a;
                second = b;
        }

        // construct line from point and angle of
            elevation
        line(point a, ct ang) : line(a, a+polar((ct)1.0,
             ang)) {}

        // construct line from standard equation
            coefficients
        // assume that a != 0 or b != 0
        // TESTED
        line(ct a, ct b, ct c) {
                if (equal(b, 0.0)) {
                        // vertical line
                        ct cx = c/(-a);
                        first = {cx, 0};
                        second = {cx, 1};
                }
                else {
                        first = {0, c/(-b)};
                        second = {1, (a+c)/(-b)};
                }
                if (point_comp(second, first)) swap(
                    first, second);
        }
};

struct line_segment {
        point first, second;

        // implicit conversion
        operator line() {
                return line(first, second);
        }

        line_segment(point a, point b) {
                if (point_comp(b, a)) swap(a, b);
                first = a;
                second = b;
        }

        line_segment(point a, ct ang, ct len) :
            line_segment(a, a+polar(len, ang)) {};
};

// assume that the first and last vertices are the same
typedef vector<point> polygon;

// radians to degrees
ct rad_to_deg(ct arad) {
        return (arad*((ct)180.0/PI));
}

// degrees to radians
ct deg_to_rad(ct adeg) {
        return (adeg*(PI/(ct)180.0));
}

// dot product, > 0 if a, b point to same direction, 0
    if perpendicular, < 0 if pointing to opposite
    directions
ct dot(point a, point b) {
        return (conj(a)*b).X;
}

// 2D cross product, > 0 if a+b turns left, 0 if
    collinear, < 0 if turns right
ct cross(point a, point b) {
        return (conj(a)*b).Y;
}
```

```
// euclidean distance
// TESTED
ct dist(point a, point b) {
	return abs(a-b);
}

// squared distance
ct sq_dist(point a, point b) {
	return norm(a-b);
}

// angle from a to b
// [0, 2*pi[
// TESTED
ct angle(point a, point b) {
	ct cres = arg(b-a);
	if (cres < 0) cres = 2*PI+cres;
	return cres;
}

// angle of elevation
// [-pi/2, pi/2]
ct elev_ang(point a, point b) {
	if (point_comp(b, a)) swap(a, b);
	return arg(b-a);
}

// angle of elevation
ct elev_ang(line l) {
	return elev_ang(l.F, l.S);
}

// slope of line
ct slope(point a, point b) {
	return tan(elev_ang(a, b));
}

// slope of line
ct slope(line l) {
	return tan(elev_ang(l));
}

// length of line segment
ct segment_len(line_segment ls) {
	return dist(ls.F, ls.S);
}

// rotate a around origin by ang
point rot_origin(point a, ct ang) {
	return (a*polar((ct)1.0, ang));
}

// rotate a around ps by ang
point rot_pivot(point a, point ps, ct ang) {
	return ((a-ps)*polar((ct)1.0, ang)+ps);
}

// translate a by dist to the direction of ang
point translate(point a, ct dist, ct ang) {
	return a+polar(dist, ang);
}

// check if a -> b -> c turns counterclockwise
bool ccw(point a, point b, point c) {
	return cross({b.X-a.X, b.Y-a.Y}, {c.X-a.X, c.Y-a
	    .Y}) > 0;
}

// < 0 if point is left, ~0 if on line, > 0 if right
// TESTED
ct point_line_side(point a, line l) {
	return cross(a-l.F, a-l.S);
}

// check if point is on line
// TESTED
bool point_on_line(point a, line l) {
	return equal(point_line_side(a, l), (ct)0.0);
}

// check if point is on line segment
// TESTED
bool point_on_seg(point a, line_segment ls) {
	if (!point_on_line(a, ls)) return false;
	if (equal(a, ls.F) || equal(a, ls.S)) return
	    true;
	return (point_comp(ls.F, a) && point_comp(a, ls.
	    S));
}
```

```
// get projection of a on l
// TESTED
point point_line_proj(point a, line l) {
        return (l.F+(l.S-l.F)*dot(a-l.F, l.S-l.F)/norm(l
            .S-l.F));
}

// reflect a across l
point point_line_refl(point a, line l) {
        return (l.F+conj((a-l.F)/(l.S-l.F))*(l.S-l.F));
}

// angle a-b-c
// [0, PI]
// TESTED
ct ang_abc(point a, point b, point c) {
        return abs(remainder(arg(a-b)-arg(c-b), (ct)2.0*
            PI));
}

// shortest distance between point a and line l
// TESTED
ct point_line_dist(point a, line l) {
        point proj = point_line_proj(a, l);
        return dist(a, proj);
}

// shortest distance between point a and line segment ls
// TESTED
ct point_segment_dist(point a, line_segment ls) {
        point proj = point_line_proj(a, ls);
        if (point_on_seg(proj, ls)) {
                return dist(a, proj);
        }
        return min(dist(a, ls.F), dist(a, ls.S));
}

// get intersection point of two lines
// first return val 0 = no intersection, 1 = single
    point, 2 = infinitely many
// second return val = intersection point if first
    return val = 1, otherwise undefined
// TESTED (only non-degenerate cases, single
    intersection point)
pair<int, point> intersect(line a, line b) {
        ct c1 = cross(b.F-a.F, a.S-a.F);
        ct c2 = cross(b.S-a.F, a.S-a.F);
        if (equal(c1, c2)) {
                if (point_on_line(b.F, a)) {
                        return {2, a.F};
                }
                return {0, a.F};
        }
        return {1, (c1*b.S-c2*b.F)/(c1-c2)};
}

// sort comparer for seg_intersect
bool pi_comp(pair<point, int> p1, pair<point, int> p2) {
        if (equal(p1.F, p2.F)) return p1.S < p2.S;
        return point_comp(p1.F, p2.F);
}

// get intersection point of two line segments
// first return val 0 = no intersection, 1 = single
    point, 2 = infinitely many
// second return val = intersection point if first
    return val = 1, otherwise undefined
// might miss an intersection due to precision issues if
     coordinates are too large, increasing epsilon works
pair<int, point> seg_intersect(line_segment a,
    line_segment b) {
        ct alen = segment_len(a);
        ct blen = segment_len(b);

        if (equal(alen, (ct)0) && equal(blen, (ct)0)) {
                return (equal(a.F, b.F) ? make_pair(1, a
                    .F) : make_pair(0, a.F));
        }
        else if (equal(alen, (ct)0)) {
                return (point_on_seg(a.F, b) ? make_pair
                    (1, a.F) : make_pair(0, a.F));
        }
        else if (equal(blen, (ct)0)) {
                return (point_on_seg(b.F, a) ? make_pair
                    (1, b.F) : make_pair(0, b.F));
        }

        auto tres = intersect(a, b);
        if (tres.F == 0) {
```

```
            return tres;
        }
        else if (tres.F == 2) {
            vector<pair<point, int>> v = {{a.F, 1},
                {a.S, 1}, {b.F, 2}, {b.S, 2}};
            sort(v.begin(), v.end(), pi_comp);
            if (v[0].S != v[1].S) return {2, a.F};
                // overlapping segments

            // common vertex
            if (equal(a.S, b.F)) return {1, a.S};
            if (equal(a.F, b.S)) return {1, a.F};

            // not intersecting but on the same line
            return {0, a.F};
        }
        if (point_on_seg(tres.S, a) && point_on_seg(tres
            .S, b)) {
            return tres;
        }
        return {0, a.F};
}

// get polygon area
// O(n)
// TESTED
ct pgon_area(polygon pg) {
        ct cres = 0;
        for (int i = 0; i < pg.size()-1; ++i) {
            cres += cross(pg[i], pg[i+1]);
        }
        return (abs(cres)/(ct)2.0);
}

// check if point is inside polygon
// 0 = outside, 1 = inside, 2 = on polygon edge
// O(n)
// TESTED
int point_in_pgon(point a, polygon pg) {
        for (int i = 0; i < pg.size()-1; ++i) {
            if (point_on_seg(a, line_segment(pg[i],
                pg[i+1]))) {
                    return 2;
            }
        }
        // arbitrary angle, try to avoid polygon
            vertices (likely lattice points)
        line_segment tl = line_segment(a, {(ct)1092854,
            (ct)1085417});
        int icnt = 0;
        for (int i = 0; i < pg.size()-1; ++i) {
            auto cur = seg_intersect(tl,
                line_segment(pg[i], pg[i+1]));
            if (cur.F == 1) {
                icnt++;
            }
        }
        return (icnt%2 == 1);
}

// return the points that form given point set's convex
    hull
// O(n log n)
vector<point> convex_hull(vector<point> ps) {
        vector<point> ch;
        sort(ps.begin(), ps.end(), point_comp);
    for (int cv = 0; cv < 2; ++cv) {
        for (int i = 0; i < ps.size(); ++i) {
            int cs = ch.size();
            while (cs >= 2 && ccw(ch[cs-2], ch[cs-1], ps
                [i])) {
                ch.pop_back();
                --cs;
            }
            ch.push_back(ps[i]);
        }
        ch.pop_back();
        reverse(ps.begin(), ps.end());
    }
    return ch;
}
```

6 Graph algorithms

6.1 Kosaraju's algorithm

Finds strongly connected components in a directed graph in $O(n+m)$.

1. Create an inverse graph where all edges are reversed.
2. Do a DFS traversal on original graph and add all nodes in post-order to a vector.
3. Reverse the obtained vector.
4. Iterate the vector. If a node doesn't belong to a component, create new component and assign current node to it, and do a DFS **in inverse graph** from current node and add all reachable nodes to the component that was just created.

6.2 Bridges

An edge $u - v$ is a bridge if there is no edge from the subtree of v to any node with lower depth than u in DFS tree. $O(n+m)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int n, m;
vector<int> g[200010];

int v[200010];
int d[200010];

// found bridges
vector<pair<int, int>> res;

// find bridges
int bdfs(int s, int cd, int p) {
    if (v[s]) return d[s];
    v[s] = 1;
    d[s] = cd;

    int minh = cd;

    for (int a : g[s]) {
        if (a == p) continue;
        minh = min(minh, bdfs(a, cd+1, s));
    }

    if (p != -1) {
        if (minh == cd) {
            res.push_back({s, p});
        }
    }
    return minh;
}

int main() {
    for (int i = 1; i <= n; ++i) {
        if (!v[i]) bdfs(i, 1, -1);
    }
}
```

6.3 Articulation points

A vertex u is an articulation point if there is no edge from the subtree of u to any parent of u in DFS tree, or if u is the root of DFS tree and has at least 2 children. $O(n+m)$ if removing duplicates doesn't count.

Set `res` can be replaced with a vector if duplicates are removed afterwards.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>

using namespace std;

int n, m;
vector<int> g[200010];
int v[200010];
int dt[200010];
int low[200010];

// found articulation points
// can be replaced with vector, but duplicates must be
    removed
set<int> res;
```

```
int curt = 1;

void adfs(int s, int p) {
    if (v[s]) return;
    v[s] = 1;
    dt[s] = curt++;
    low[s] = dt[s];

    int ccount = 0;

    for (int a : g[s]) {
        if (!v[a]) {
            ++ccount;
            adfs(a, s);
            low[s] = min(low[s], low[a]);

            if (low[a] >= dt[s] && p != -1) res.insert(s
                );
        }
        else if (a != p) {
            low[s] = min(low[s], dt[a]);
        }

        if (p == -1 && ccount > 1) {
            res.insert(s);
        }
    }
}


int main() {
    for (int i = 1; i <= n; ++i) {
        if (!v[i]) adfs(i, -1);
    }
}
```

6.4 Maximum flow (scaling algorithm)

Scaling algorithm, uses DFS to find an augmenting path where each edge weight is larger than or equal to a certain threshold. Time complexity $O(m^2\ log(c))$, where c is the starting threshold (sum of all edge weights in the graph).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
typedef long long ll;

const int N = 105; // vertex count
const ll LINF = 1000000000000000005;

int n, m;
vector<int> g[N];
ll d[N][N]; // edge weights

int v[N];
vector<int> cp; // current augmenting path

ll res = 0;

// find augmenting path using scaling
// prerequisities: clear current path, divide threshold
    by 2, increment cvis
ll dfs(int s, int t, ll thresh, int cvis, ll cmin) {
    if (v[s] == cvis) return -1;
    v[s] = cvis;
    cp.push_back(s);
    if (s == t) return cmin;

    for (int a : g[s]) {
        if (d[s][a] < thresh) continue; // scaling
        ll cres = dfs(a, t, thresh, cvis, min(cmin, d[s
            ][a]));
        if (cres != -1) return cres;
    }

    cp.pop_back();
    return -1;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    ll cthresh = 0;
    for (int i = 0; i < m; ++i) {
```

```
        int a, b;
        ll c;
        cin >> a >> b >> c;
        g[a].push_back(b);
        g[b].push_back(a);
        d[a][b] += c;
        d[b][a] = 0;
        cthresh += c;
    }
    int cvis = 0;
    while (true) {
        cvis++;
        cp.clear();
        ll minw = dfs(1, n, cthresh, cvis, LINF);
        if (minw != -1) {
            res += minw;
            for (int i = 0; i < cp.size()-1; ++i) {
                d[cp[i]][cp[i+1]] -= minw;
                d[cp[i+1]][cp[i]] += minw;
            }
        }
        else {
            if (cthresh == 1) break;
            cthresh /= 2;
        }
    }
    cout << res << "\n";
    return 0;
}
```

6.5 Theorems on flows and cuts

Maximum flow is always equal to minimum cut. Minimum cut can be found by running a maximum flow algorithm and dividing the resulting flow graph into two sets of vertices. Set A contains all vertices that can be reached from source using positive-weight edges. Set B contains all other vertices. Minimum cut consists of the edges between these two sets.

Number of edge-disjoint (= each edge can be used at most once) paths in a graph is equal to maximum flow on graph where capacity of each edge is 1.

Number of vertex-disjoint paths can be found the same way as edge-disjoint paths, but each vertex is duplicated and an edge is added between the two vertices. All incoming edges go to the first vertex and all outgoing edges start from the second vertex.

Maximum matching of a bipartite graph can be found by adding a source and a sink to the graph and connecting source to all left vertices and sink to all right vertices. Maximum matching equals maximum flow on this graph.

König's theorem: sizes of a minimum vertex cover (= minimum set of vertices such that each edge has at least one endpoint in the set) and a maximum matching are always equal in a bipartite graph. Maximum independent set (= maximum set of vertices such that no two vertices in the set are connected with an edge) consists of the vertices not in a minimum vertex cover.

6.6 Heavy-light decomposition

Supports updates and queries on path between two vertices a and b in $O(log^2(n))$.

Doesn't explicitly look for LCA, instead climbs upwards from the lower chain until both vertices are in the same chain.

Requires a segment tree implementation that corresponds to the queries. Lazy segtree, for example, can be pasted directly in.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
typedef long long ll;

const int S = 100005; // vertex count
const int N = (1<<18); // segtree size, must be >= S

vector<int> g[S];

int sz[S], de[S], pa[S];
int cind[S], chead[S], cpos[S];
int cchain, cstind, stind[S];
```

```cpp
// IMPLEMENT SEGMENT TREE HERE
// st_update() and st_query() should call segtree
    functions
ll st[2*N];

void hdfs(int s, int p, int cd) {
    de[s] = cd;
    pa[s] = p;
    sz[s] = 1;
    for (int a : g[s]) {
        if (a == p) continue;
        hdfs(a, s, cd+1);
        sz[s] += sz[a];
    }
}

void hld(int s) {
    if (chead[cchain] == 0) {
        chead[cchain] = s;
        cpos[s] = 0;
    }
    else {
        cpos[s] = cpos[pa[s]]+1;
    }
    cind[s] = cchain;

    stind[s] = cstind;
    cstind++;

    int cmx = 0, cmi = -1;
    for (int i = 0; i < g[s].size(); ++i) {
        if (g[s][i] == pa[s]) continue;
        if (sz[g[s][i]] > cmx) {
            sz[g[s][i]] = cmx;
            cmi = i;
        }
    }

    if (cmi != -1) {
        hld(g[s][cmi]);
    }

    for (int i = 0; i < g[s].size(); ++i) {
        if (i == cmi) continue;
        if (g[s][i] == pa[s]) continue;
        cchain++;
        cstind++;
        hld(g[s][i]);
    }
}

// do a range update on underlying segtree
// sa and sb are segtree indices
void st_update(int sa, int sb, ll x) {

}

// do a range query on underlying segtree
// sa and sb are segtree indices
ll st_query(int sa, int sb) {

}

// update all vertices on path from vertex a to b
// a and b are vertex numbers
void path_update(int a, int b, ll x) {
    while (cind[a] != cind[b]) {
        if (de[chead[cind[b]]] > de[chead[cind[a]]])
            swap(a, b);
        st_update(stind[chead[cind[a]]], stind[a], x);
        a = pa[chead[cind[a]]];
    }
    if (stind[b] < stind[a]) swap(a, b);
    st_update(stind[a], stind[b], x);
}

// query all vertices on path from vertex a to b
// a and b are vertex numbers
ll path_query(int a, int b) {
        ll cres = 0; // set to identity
        while (cind[a] != cind[b]) {
        if (de[chead[cind[b]]] > de[chead[cind[a]]])
            swap(a, b);
        cres += st_query(stind[chead[cind[a]]], stind[a
            ]); // change operator
        a = pa[chead[cind[a]]];
    }
    if (stind[b] < stind[a]) swap(a, b);
    cres += st_query(stind[a], stind[b]); // change
        operator
```

```
    return cres;
}

// TESTED, correct
// do updates and queries on paths between two nodes in
    a tree
// interface: path_update() and path_query()
int main() {
    // init hld
    hdfs(1, -1, 0);
    hld(1);

    // handle queries
}
```

7 String algorithms

7.1 Polynomial hashing

If hash collisions are likely, compute two hashes with two distinct pairs of constants of magnitude 10^9 and use their product as the actual hash.

```
#include <iostream>

using namespace std;

const ll A = 957262683;
const ll B = 998735246;

string s;
ll h[1000005];
ll p[1000005];

ll ghash(int a, int b) {
        if (a == 0) return h[b];
        ll cres = (h[b]-h[a-1]*p[b-a+1])%B;
        if (cres < 0) cres += B;
        return cres;
}

int main() {
        cin >> s;

        h[0] = s[0];
        p[0] = 1;

        for (int i = 1; i < s.length(); ++i) {
                h[i] = (h[i-1]*A+s[i])%B;
                p[i] = (p[i-1]*A)%B;
        }
        return 0;
}
```