Universidad de Guadalajara,
CUCEI

The Empire Strikes Back

Abraham Murillo, Roberto Pino, Uriel Guzmán

Contents

Think twice, code once

Template

tem.cpp

```
#pragma GCC optimize("Ofast,unroll-loops,no-stack-
    protector")
#include <bits/stdc++.h>
using namespace std;

#ifdef LOCAL
#include "debug.h"
#else
#define debug(...)
#endif

#define df(b, e) ((b) > (e))
#define fore(i, b, e) for (auto i = (b) - df(b, e); i
    != e - df(b, e); i += 1 - 2 * df(b, e))
#define sz(x) int(x.size())
#define all(x) begin(x), end(x)
#define f first
#define s second
#define pb push_back

using lli = long long;
using ld = long double;
using ii = pair<int, int>;
using vi = vector<int>;

int main() {
  cin.tie(0)->sync_with_stdio(0), cout.tie(0);
  // solve the problem here D:
  return 0;
}
```

debug.h

```
template <class A, class B>
ostream & operator << (ostream &os, const pair<A, B> &
    p) {
  return os << "(" << p.first << ", " << p.second << "
      )";
}

template <class A, class B, class C>
basic_ostream<A, B> & operator << (basic_ostream<A, B>
     &os, const C &c) {
  os << "[";
  for (const auto &x : c)
    os << ", " + 2 * (&x == &*begin(c)) << x;
  return os << "]";
}

void print(string s) { cout << endl; }

template <class H, class... T>
void print(string s, const H &h, const T&... t) {
  const static string reset = "\033[0m", blue = "\033[
      1;34m", purple = "\033[3;95m";
  bool ok = 1;
  do {
    if (s[0] == '\"') ok = 0;
    else cout << blue << s[0] << reset;
    s = s.substr(1);
  } while (s.size() && s[0] != ',');
  if (ok) cout << ": " << purple << h << reset;
  print(s, t...);
}
```

Randoms

```
mt19937 rng(chrono::steady_clock::now().
    time_since_epoch().count());

template <class T>
T ran(T l, T r) {
  return uniform_int_distribution<T>(l, r)(rng);
}
```

Fastio

```
char gc() { return getchar_unlocked(); }

void readInt() {}
template <class H, class... T>
void readInt(H &h, T&... t) {
  char c, s = 1;
  while (isspace(c = gc()));
  if (c == '-') s = -1, c = gc();
  for (h = c - '0'; isdigit(c = gc()); h = h * 10 + c
      - '0');
  h *= s;
  readInt(t...);
}

void readFloat() {}
template <class H, class... T>
void readFloat(H &h, T&... t) {
  int c, s = 1, fp = 0, fpl = 1;
  while (isspace(c = gc()));
  if (c == '-') s = -1, c = gc();
  for (h = c - '0'; isdigit(c = gc()); h = h * 10 + c
      - '0');
  h *= s;
  if (h == '.')
    for (; isdigit(c = gc()); fp = fp * 10 + c - '0',
        fpl *= 10);
  h += (double)fp / fpl;
  readFloat(t...);
}
```

Compilation (gedit ~/.zshenv)

```
touch a_in{1..9} // make files a_in1, a_in2,..., a_in9
tee {a..m}.cpp < tem.cpp // "" with tem.cpp like base
cat > a_in1 // write on file a_in1
gedit a_in1 // open file a_in1
rm -r a.cpp // deletes file a.cpp :'(

red='\x1B[0;31m'
green='\x1B[0;32m'
noColor='\x1B[0m'
alias flags='-Wall -Wextra -Wshadow -
    D_GLIBCXX_ASSERTIONS -fmax-errors=3 -O2 -w'
go() { g++ --std=c++11 $2 ${flags} $1.cpp && ./a.out }
debug() { go $1 -DLOCAL < $2 }
run() { go $1 "" < $2 }

random() { // Make small test cases!!!
 g++ --std=c++11 $1.cpp -o prog
 g++ --std=c++11 gen.cpp -o gen
 g++ --std=c++11 brute.cpp -o brute
 for ((i = 1; i <= 200; i++)); do
  printf "Test case #$i"
  ./gen > in
  diff -uwi <(./prog < in) <(./brute < in) > $1_diff
  if [[ ! $? -eq 0 ]]; then
   printf "${red} Wrong answer ${noColor}\n"
   break
  else
   printf "${green} Accepted ${noColor}\n"
  fi
 done
}

test() {
```

```
g++ --std=c++11 $1.cpp -o prog
for ((i = 1; i <= 50; i++)); do
  [[ -f $1_in$i ]] || break
  printf "Test case #$i"
  diff -uwi <(./prog < $1_in$i) $1_out$i > $1_diff
  if [[ ! $? -eq 0 ]]; then
   printf "${red} Wrong answer ${noColor}\n"
  else
   printf "${green} Accepted ${noColor}\n"
  fi
done
}
```

Bump allocator

```
static char buf[450 << 20];
void* operator new(size_t s) {
  static size_t i = sizeof buf; assert(s < i);
  return (void *) &buf[i -= s];
}
void operator delete(void *) {}
```

1 Data structures

1.1 Disjoint set with rollback

```
struct Dsu {
  vi par, tot;
  stack<ii> mem;

  Dsu(int n = 1) : par(n + 1), tot(n + 1, 1) {
    iota(all(par), 0);
  }

  int find(int u) {
    return par[u] == u ? u : find(par[u]);
  }

  void unite(int u, int v) {
    u = find(u), v = find(v);
    if (u != v) {
      if (tot[u] < tot[v])
        swap(u, v);
      mem.emplace(u, v);
      tot[u] += tot[v];
      par[v] = u;
    }
  }

  void rollback() {
    auto [u, v] = mem.top();
    mem.pop();
    if (u != -1) {
      tot[u] -= tot[v];
      par[v] = v;
    }
  }
};
```

1.2 Min-Max queue

```
template <class T>
struct MinQueue : deque< pair<T, int> > {
  // add a element to the right {val, pos}
  void add(T val, int pos) {
    while (!empty() && back().f >= val)
      pop_back();
    emplace_back(val, pos);
  }
  // remove all less than pos
  void rem(int pos) {
    while (front().s < pos)
      pop_front();
  }
  T qmin() { return front().f; }
};
```

1.3 Sparse table

```
template <class T, class F = function<T(const T&,
    const T&)>>
struct Sparse {
  int n;
  vector<vector<T>> sp;
  F f;

  Sparse(vector<T> &a, const F &f) : n(sz(a)), sp(1 +
      __lg(n)), f(f) {
    sp[0] = a;
    for (int k = 1; (1 << k) <= n; k++) {
      sp[k].resize(n - (1 << k) + 1);
      fore (l, 0, sz(sp[k])) {
        int r = l + (1 << (k - 1));
        sp[k][l] = f(sp[k - 1][l], sp[k - 1][r]);
      }
    }
  }

  T query(int l, int r) {
    int k = __lg(r - l + 1);
    return f(sp[k][l], sp[k][r - (1 << k) + 1]);
  }
};
```

1.4 Squirtle decomposition

The perfect block size is *squirtle* of N

```
int blo[N], cnt[N][B], a[N];

void update(int i, int x) {
  cnt[blo[i]][x]--;
  a[i] = x;
  cnt[blo[i]][x]++;
}

int query(int l, int r, int x) {
  int tot = 0;
  while (l <= r)
    if (l % B == 0 && l + B - 1 <= r) {
      tot += cnt[blo[l]][x];
      l += B;
    } else {
      tot += (a[l] == x);
      l++;
    }
  return tot;
}
```

1.5 In-Out trick

```
vector<int> in[N], out[N];
vector<Query> queries;

fore (x, 0, N) {
  for (int i : in[x])
    add(queries[i]);
  // solve
  for (int i : out[x])
    rem(queries[i]);
}
```

1.6 Parallel binary search

```
int lo[Q], hi[Q];
queue<int> solve[N];
vector<Query> queries;
```

```
fore (it, 0, 1 + __lg(N)) {
  fore (i, 0, sz(queries))
    if (lo[i] != hi[i]) {
      int mid = (lo[i] + hi[i]) / 2;
      solve[mid].emplace(i);
    }
  fore (x, 0, n) {
    // simulate
    while (!solve[x].empty()) {
      int i = solve[x].front();
      solve[x].pop();
      if (can(queries[i]))
        hi[i] = x;
      else
        lo[i] = x + 1;
    }
  }
}
```

1.7 Mo's algorithm

```
vector<Query> queries;
// N = 1e6, so aprox. sqrt(N) +/- C
uniform_int_distribution<int> dis(970, 1030);
const int blo = dis(rng);
sort(all(queries), [&](Query a, Query b) {
  const int ga = a.l / blo, gb = b.l / blo;
  if (ga == gb)
    return (ga & 1) ? a.r < b.r : a.r > b.r;
  return a.l < b.l;
});
int l = queries[0].l, r = l - 1;
for (Query &q : queries) {
  while (r < q.r)
    add(++r);
  while (r > q.r)
    rem(r--);
  while (l < q.l)
    rem(l++);
  while (l > q.l)
    add(--l);
  ans[q.i] = solve();
}
```

To make it faster, change the order to $hilbert(l, r)$

```
lli hilbert(int x, int y, int pw = 21, int rot = 0) {
  if (pw == 0)
    return 0;
  int hpw = 1 << (pw - 1);
  int k = ((x < hpw ? y < hpw ? 0 : 3 : y < hpw ? 1 :
      2) + rot) & 3;
  const int d[4] = {3, 0, 0, 1};
  lli a = 1LL << ((pw << 1) - 2);
  lli b = hilbert(x & (x ^ hpw), y & (y ^ hpw), pw - 1
      , (rot + d[k]) & 3);
  return k * a + (d[k] ? a - b - 1 : b);
}
```

Mo's algorithm with updates in $\mathcal{O}(n^{\frac{5}{3}})$

- Choose a *block* of size $n^{\frac{2}{3}}$
- Do a normal Mo's algorithm, in the *Query* definition add an extra variable for the *updatesSoFar*
- Sort the queries by the order ($l/block$, $r/block$, $updatesSoFar$)
- If the update lies inside the current query, update the data structure properly

```
struct Update {
  int pos, prv, nxt;
};
```

```
void undo(Update &u) {
  if (l <= u.pos && u.pos <= r) {
    rem(u.pos);
    a[u.pos] = u.prv;
    add(u.pos);
  } else {
    a[u.pos] = u.prv;
  }
}
```

- Solve the problem :D

```
l = queries[0].l, r = l - 1, upd = sz(updates) - 1;
for (Query &q : queries) {
  while (upd < q.upd)
    dodo(updates[++upd]);
  while (upd > q.upd)
    undo(updates[upd--]);
  // write down the normal Mo's algorithm
}
```

1.8 Static to dynamic

```
template <class Black, class T>
struct StaticDynamic {
  Black box[LogN];
  vector<T> st[LogN];

  void insert(T &x) {
    int p = 0;
    fore (i, 0, LogN)
      if (st[i].empty()) {
        p = i;
        break;
      }
    st[p].pb(x);
    fore (i, 0, p) {
      st[p].insert(st[p].end(), all(st[i]));
      box[i].clear(), st[i].clear();
    }
    for (auto y : st[p])
      box[p].insert(y);
    box[p].init();
  }
};
```

1.9 Disjoint intervals

```
struct Range {
  int l, r;
  bool operator < (const Range& rge) const {
    return l < rge.l;
  }
};

struct DisjointIntervals : set<Range> {
  void add(Range rge) {
    iterator p = lower_bound(rge), q = p;
    if (p != begin() && rge.l <= (--p)->r)
      rge.l = p->l, --q;
    for (; q != end() && q->l <= rge.r; erase(q++))
      rge.r = max(rge.r, q->r);
    insert(rge);
  }

  void add(int l, int r) {
    add(Range{l, r});
  }
};
```

1.10 Ordered tree

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
```

```cpp
template <class K, class V = null_type>
using ordered_tree = tree<K, V, less<K>, rb_tree_tag,
    tree_order_statistics_node_update>;
// less_equal<K> for multiset, multimap (?
#define rank order_of_key
#define kth find_by_order
```

1.11 Unordered tree

```cpp
struct chash {
  const uint64_t C = uint64_t(2e18 * 3) + 71;
  const int R = rng();
  uint64_t operator ()(uint64_t x) const {
    return __builtin_bswap64((x ^ R) * C); }
};

template <class K, class V = null_type>
using unordered_tree = gp_hash_table<K, V, chash>;
```

1.12 D-dimensional Fenwick tree

```cpp
template <class T, int ...N>
struct Fenwick {
  T v = 0;
  void update(T v) { this->v += v; }
  T query() { return v; }
};

template <class T, int N, int ...M>
struct Fenwick<T, N, M...> {
  #define lsb(x) (x & -x)
  Fenwick<T, M...> fenw[N + 1];

  template <typename... Args>
  void update(int i, Args... args) {
    for (; i <= N; i += lsb(i))
      fenw[i].update(args...);
  }

  template <typename... Args>
  T query(int l, int r, Args... args) {
    T v = 0;
    for (; r > 0; r -= lsb(r))
      v += fenw[r].query(args...);
    for (--l; l > 0; l -= lsb(l))
      v -= fenw[l].query(args...);
    return v;
  }
};
```

1.13 Dynamic segment tree

```cpp
struct Dyn {
  int l, r;
  lli sum = 0;
  Dyn *ls, *rs;

  Dyn(int l, int r) : l(l), r(r), ls(0), rs(0) {}

  void pull() {
    sum = (ls ? ls->sum : 0);
    sum += (rs ? rs->sum : 0);
  }

  void update(int p, lli v) {
    if (l == r) {
      sum += v;
      return;
    }
    int m = (l + r) >> 1;
    if (p <= m) {
      if (!ls) ls = new Dyn(l, m);
      ls->update(p, v);
    } else {
      if (!rs) rs = new Dyn(m + 1, r);
      rs->update(p, v);
    }
    pull();
  }

  lli qsum(int ll, int rr) {
    if (rr < l || r < ll || r < l)
      return 0;
    if (ll <= l && r <= rr)
      return sum;
    int m = (l + r) >> 1;
    return (ls ? ls->qsum(ll, rr) : 0) +
           (rs ? rs->qsum(ll, rr) : 0);
  }
};
```

1.14 Persistent segment tree

```cpp
struct Per {
  int l, r;
  lli sum = 0;
  Per *ls, *rs;

  Per(int l, int r) : l(l), r(r), ls(0), rs(0) {}

  Per* pull() {
    sum = ls->sum + rs->sum;
    return this;
  }

  void build() {
    if (l == r)
      return;
    int m = (l + r) >> 1;
    (ls = new Per(l, m))->build();
    (rs = new Per(m + 1, r))->build();
    pull();
  }

  Per* update(int p, lli v) {
    if (p < l || r < p)
      return this;
    Per* t = new Per(l, r);
    if (l == r) {
      t->sum = v;
      return t;
    }
    t->ls = ls->update(p, v);
    t->rs = rs->update(p, v);
    return t->pull();
  }

  lli qsum(int ll, int rr) {
    if (r < ll || rr < l)
      return 0;
    if (ll <= l && r <= rr)
      return sum;
    return ls->qsum(ll, rr) + rs->qsum(ll, rr);
  }
};
```

1.15 Wavelet tree

```cpp
struct Wav {
  #define iter int* // vector<int>::iterator
  int lo, hi;
  Wav *ls, *rs;
  vi amt;

  Wav(int lo, int hi) : lo(lo), hi(hi), ls(0), rs(0)
```

```
    {}

  void build(iter b, iter e) { // array 1-indexed
    if (lo == hi || b == e)
      return;
    amt.reserve(e - b + 1);
    amt.pb(0);
    int m = (lo + hi) >> 1;
    for (auto it = b; it != e; it++)
      amt.pb(amt.back() + (*it <= m));
    auto p = stable_partition(b, e, [&](int x) {
      return x <= m;
    });
    (ls = new Wav(lo, m))->build(b, p);
    (rs = new Wav(m + 1, hi))->build(p, e);
  }

  int kth(int l, int r, int k) {
    if (r < l)
      return 0;
    if (lo == hi)
      return lo;
    if (k <= amt[r] - amt[l - 1])
      return ls->kth(amt[l - 1] + 1, amt[r], k);
    return rs->kth(l - amt[l - 1], r - amt[r], k - amt
        [r] + amt[l - 1]);
  }

  int leq(int l, int r, int mx) {
    if (r < l || mx < lo)
      return 0;
    if (hi <= mx)
      return r - l + 1;
    return ls->leq(amt[l - 1] + 1, amt[r], mx) +
           rs->leq(l - amt[l - 1], r - amt[r], mx);
  }
};
```

1.16 Li Chao tree

```
struct Fun {
  lli m = 0, c = inf;
  lli operator ()(lli x) const { return m * x + c; }
};

struct LiChao {
  Fun f;
  lli l, r;
  LiChao *ls, *rs;

  LiChao(lli l, lli r) : l(l), r(r), ls(0), rs(0) {}

  void add(Fun &g) {
    if (f(l) <= g(l) && f(r) <= g(r))
      return;
    if (g(l) < f(l) && g(r) < f(r)) {
      f = g;
      return;
    }
    lli m = (l + r) >> 1;
    if (g(m) < f(m))
      swap(f, g);
    if (g(l) <= f(l))
     ls = ls ? (ls->add(g), ls) : new LiChao(l, m, g);
    else
     rs = rs ? (rs->add(g), rs) : new LiChao(m + 1, r,
          g);
  }

  lli query(lli x) {
    if (l == r)
      return f(x);
    lli m = (l + r) >> 1;
    if (x <= m)
      return min(f(x), ls ? ls->query(x) : inf);
    return min(f(x), rs ? rs->query(x) : inf);
  }
};
```

1.17 Explicit Treap

```
typedef struct Node* Treap;
struct Node {
  Treap ch[2] = {0, 0}, p = 0;
  uint32_t pri = rng();
  int sz = 1, rev = 0;
  int val, sum = 0;

  void push() {
    if (rev) {
      swap(ch[0], ch[1]);
      for (auto ch : ch) if (ch != 0) {
        ch->rev ^= 1;
      }
      rev = 0;
    }
  }

  Treap pull() {
    #define gsz(t) (t ? t->sz : 0)
    #define gsum(t) (t ? t->sum : 0)
    sz = 1, sum = val;
    for (auto ch : ch) if (ch != 0) {
      ch->push();
      sz += ch->sz;
      sum += ch->sum;
      ch->p = this;
    }
    p = 0;
    return this;
  }

  Node(int val) : val(val) {}
};

pair<Treap, Treap> split(Treap t, int val) {
  // <= val goes to the left, > val to the right
  if (!t)
    return {t, t};
  t->push();
  if (val < t->val) {
    auto p = split(t->ch[0], val);
    t->ch[0] = p.s;
    return {p.f, t->pull()};
  } else {
    auto p = split(t->ch[1], val);
    t->ch[1] = p.f;
    return {t->pull(), p.s};
  }
}

Treap merge(Treap l, Treap r) {
  if (!l || !r)
    return l ? l : r;
  l->push(), r->push();
  if (l->pri > r->pri)
    return l->ch[1] = merge(l->ch[1], r), l->pull();
  else
    return r->ch[0] = merge(l, r->ch[0]), r->pull();
}

Treap qkth(Treap t, int k) { // 0-indexed
```

```
  if (!t)
    return t;
  t->push();
  int sz = gsz(t->ch[0]);
  if (sz == k)
    return t;
  return k < sz ? qkth(t->ch[0], k) : qkth(t->ch[1], k
       - sz - 1);
}

int qrank(Treap t, int val) { // 0-indexed
  if (!t)
    return -1;
  t->push();
  if (val < t->val)
    return qrank(t->ch[0], val);
  if (t->val == val)
    return gsz(t->ch[0]);
  return gsz(t->ch[0]) + qrank(t->ch[1], val) + 1;
}

Treap insert(Treap t, int val) {
  auto p1 = split(t, val);
  auto p2 = split(p1.f, val - 1);
  return merge(p2.f, merge(new Node(val), p1.s));
}

Treap erase(Treap t, int val) {
  auto p1 = split(t, val);
  auto p2 = split(p1.f, val - 1);
  return merge(p2.f, p1.s);
}
```

1.18 Implicit Treap

```
pair<Treap, Treap> splitsz(Treap t, int sz) {
  // <= sz goes to the left, > sz to the right
  if (!t)
    return {t, t};
  t->push();
  if (sz <= gsz(t->ch[0])) {
    auto p = splitsz(t->ch[0], sz);
    t->ch[0] = p.s;
    return {p.f, t->pull()};
  } else {
    auto p = splitsz(t->ch[1], sz - gsz(t->ch[0]) - 1)
        ;
    t->ch[1] = p.f;
    return {t->pull(), p.s};
  }
}

int pos(Treap t) {
  int sz = gsz(t->ch[0]);
  for (; t->p; t = t->p) {
    Treap p = t->p;
    if (p->ch[1] == t)
      sz += gsz(p->ch[0]) + 1;
  }
  return sz + 1;
}
```

1.19 Splay tree

```
typedef struct Node* Splay;
struct Node {
  Splay ch[2] = {0, 0}, p = 0;
  bool rev = 0;
  int sz = 1;

  int dir() {
    if (!p) return -2; // root of LCT component
    if (p->ch[0] == this) return 0; // left child
    if (p->ch[1] == this) return 1; // right child
    return -1; // root of current splay tree
  }

  bool isRoot() { return dir() < 0; }

  friend void add(Splay u, Splay v, int d) {
    if (v) v->p = u;
    if (d >= 0) u->ch[d] = v;
  }

  void rotate() {
    // assume p and p->p propagated
    assert(!isRoot());
    int x = dir();
    Splay g = p;
    add(g->p, this, g->dir());
    add(g, ch[x ^ 1], x);
    add(this, g, x ^ 1);
    g->pull(), pull();
  }

  void splay() {
    // bring this to top of splay tree
    while (!isRoot() && !p->isRoot()) {
      p->p->push(), p->push(), push();
      dir() == p->dir() ? p->rotate() : rotate();
      rotate();
    }
    if (!isRoot()) p->push(), push(), rotate();
    push(), pull();
  }

  void pull() {
    #define gsz(t) (t ? t->sz : 0)
    sz = 1 + gsz(ch[0]) + gsz(ch[1]);
  }

  void push() {
    if (rev) {
      swap(ch[0], ch[1]);
      for (auto ch : ch) if (ch) {
        ch->rev ^= 1;
      }
      rev = 0;
    }
  }

  void vsub(Splay t, bool add) {}
};
```

2 Graphs

2.1 Tarjan algorithm (SCC)

```
int tin[N], fup[N];
bitset<N> still;
stack<int> stk;
int timer = 0;

void tarjan(int u) {
  tin[u] = fup[u] = ++timer;
  still[u] = true;
  stk.push(u);
  for (int v : graph[u]) {
    if (!tin[v])
      tarjan(v);
    if (still[v])
      fup[u] = min(fup[u], fup[v]);
  }
  if (fup[u] == tin[u]) {
```

```
    int v;
    do {
      v = stk.top();
      stk.pop();
      still[v] = false;
      // u and v are in the same scc
    } while (v != u);
  }
}
```

2.2 Kosaraju algorithm (SCC)

```
int scc[N], k = 0;
char vis[N];
vi order;

void dfs1(int u) {
  vis[u] = 1;
  for (int v : graph[u])
    if (vis[v] != 1)
      dfs1(v);
  order.pb(u);
}

void dfs2(int u, int k) {
  vis[u] = 2, scc[u] = k;
  for (int v : rgraph[u]) // reverse graph
    if (vis[v] != 2)
      dfs2(v, k);
}

void kosaraju() {
  fore (u, 1, n + 1)
    if (vis[u] != 1)
      dfs1(u);
  reverse(all(order));
  for (int u : order)
    if (vis[u] != 2)
      dfs2(u, ++k);
}
```

2.3 Two Sat

```
void add(int u, int v) {
  graph[u].pb(v);
  rgraph[v].pb(u);
}

void implication(int u, int v) {
  #define neg(u) ((n) + (u))
  add(u, v);
  add(neg(v), neg(u));
}

pair<bool, vi> satisfy(int n) {
  kosaraju(2 * n); // size of the two-sat is 2 * n
  vi ans(n + 1, 0);
  fore (u, 1, n + 1) {
    if (scc[u] == scc[neg(u)])
      return {0, ans};
    ans[u] = scc[u] > scc[neg(u)];
  }
  return {1, ans};
}
```

2.4 Topological sort

```
vi order;
int indeg[N];

void topsort() { // first fill the indeg[]
  queue<int> qu;
  fore (u, 1, n + 1)
    if (indeg[u] == 0)
      qu.push(u);
  while (!qu.empty()) {
    int u = qu.front();
    qu.pop();
    order.pb(u);
    for (int v : graph[u])
      if (--indeg[v] == 0)
        qu.push(v);
  }
}
```

2.5 Cutpoints and Bridges

```
int tin[N], fup[N], timer = 0;

void findWeakness(int u, int p = 0) {
  tin[u] = fup[u] = ++timer;
  int children = 0;
  for (int v : graph[u]) if (v != p) {
    if (!tin[v]) {
      ++children;
      findWeakness(v, u);
      fup[u] = min(fup[u], fup[v]);
      if (fup[v] >= tin[u] && p) // u is a cutpoint
      if (fup[v] > tin[u]) // bridge u -> v
    }
    fup[u] = min(fup[u], tin[v]);
  }
  if (!p && children > 1) // u is a cutpoint
}
```

2.6 Detect a cycle

```
bool cycle(int u) {
  vis[u] = 1;
  for (int v : graph[u]) {
    if (vis[v] == 1)
      return true;
    if (!vis[v] && cycle(v))
      return true;
  }
  vis[u] = 2;
  return false;
}
```

2.7 Euler tour for Mo's in a tree

Mo's in a tree, extended euler tour tin[u] = ++timer, tout[u] = ++timer

- u = lca(u, v), query(tin[u], tin[v])
- u $\neq$ lca(u, v), query(tout[u], tin[v]) + query(tin[lca], tin[lca])

2.8 Lowest common ancestor (LCA)

```
const int LogN = 1 + __lg(N);
int par[LogN][N], dep[N];

void dfs(int u, int par[]) {
  for (int v : graph[u])
    if (v != par[u]) {
      par[v] = u;
      dep[v] = dep[u] + 1;
      dfs(v, par);
    }
}

int lca(int u, int v){
  if (dep[u] > dep[v])
    swap(u, v);
  fore (k, LogN, 0)
    if (dep[v] - dep[u] >= (1 << k))
      v = par[k][v];
  if (u == v)
    return u;
  fore (k, LogN, 0)
```

```
    if (par[k][v] != par[k][u])
      u = par[k][u], v = par[k][v];
  return par[0][u];
}

int dist(int u, int v) {
  return dep[u] + dep[v] - 2 * dep[lca(u, v)];
}

void init(int r) {
  dfs(r, par[0]);
  fore (k, 1, LogN)
    fore (u, 1, n + 1)
      par[k][u] = par[k - 1][par[k - 1][u]];
}
```

2.9 Isomorphism

```
lli f(lli x) {
  // K * n <= 9e18
  static uniform_int_distribution<lli> uid(1, K);
  if (!mp.count(x))
    mp[x] = uid(rng);
  return mp[x];
}

lli hsh(int u, int p = 0) {
  dp[u] = h[u] = 0;
  for (int v : graph[u]) {
    if (v == p)
      continue;
    dp[u] += hsh(v, u);
  }
  return h[u] = f(dp[u]);
}
```

2.10 Guni

```
int cnt[C], color[N];
int sz[N];

int guni(int u, int p = 0) {
  sz[u] = 1;
  for (int &v : graph[u]) if (v != p) {
    sz[u] += guni(v, u);
    if (sz[v] > sz[graph[u][0]] || p == graph[u][0])
      swap(v, graph[u][0]);
  }
  return sz[u];
}

void add(int u, int p, int x, bool skip) {
  cnt[color[u]] += x;
  for (int i = skip; i < sz(graph[u]); i++) // don't
     change it with a fore!!!
    if (graph[u][i] != p)
      add(graph[u][i], u, x, 0);
}

void solve(int u, int p, bool keep = 0) {
  fore (i, sz(graph[u]), 0)
    if (graph[u][i] != p)
      solve(graph[u][i], u, !i);
  add(u, p, +1, 1); // add
  // now cnt[i] has how many times the color i appears
      in the subtree of u
  if (!keep) add(u, p, -1, 0); // remove
}
```

2.11 Centroid decomposition

```
int cdp[N], sz[N];
bitset<N> rem;

int dfsz(int u, int p = 0) {
  sz[u] = 1;
  for (int v : graph[u])
    if (v != p && !rem[v])
      sz[u] += dfsz(v, u);
  return sz[u];
}

int centroid(int u, int n, int p = 0) {
  for (int v : graph[u])
    if (v != p && !rem[v] && 2 * sz[v] > n)
      return centroid(v, n, u);
  return u;
}

void solve(int u, int p = 0) {
  cdp[u = centroid(u, dfsz(u))] = p;
  rem[u] = true;
  for (int v : graph[u])
    if (!rem[v])
      solve(v, u);
}
```

2.12 Heavy-light decomposition

```
int par[N], dep[N], sz[N], head[N], pos[N], who[N],
    timer = 0;
Lazy* tree;

int dfs(int u) {
  sz[u] = 1, head[u] = 0;
  for (int &v : graph[u]) if (v != par[u]) {
    par[v] = u;
    dep[v] = dep[u] + 1;
    sz[u] += dfs(v);
    if (sz[v] > sz[graph[u][0]])
      swap(v, graph[u][0]);
  }
  return sz[u];
}

void hld(int u, int h) {
  head[u] = h, pos[u] = ++timer, who[timer] = u;
  for (int &v : graph[u])
    if (v != par[u])
      hld(v, v == graph[u][0] ? h : v);
}

template <class F>
void processPath(int u, int v, F f) {
  for (; head[u] != head[v]; v = par[head[v]]) {
    if (dep[head[u]] > dep[head[v]]) swap(u, v);
    f(pos[head[v]], pos[v]);
  }
  if (dep[u] > dep[v]) swap(u, v);
  if (u != v) f(pos[graph[u][0]], pos[v]);
  f(pos[u], pos[u]); // only if hld over vertices
}

void updatePath(int u, int v, lli z) {
  processPath(u, v, [&](int l, int r) {
    tree->update(l, r, z);
  });
}

lli queryPath(int u, int v) {
  lli sum = 0;
  processPath(u, v, [&](int l, int r) {
    sum += tree->qsum(l, r);
  });
  return sum;
```

```
}
```

2.13 Link-Cut tree

```
void access(Splay u) {
  // puts u on the preferred path, splay (right
      subtree is empty)
  for (Splay v = u, pre = NULL; v; v = v->p) {
    v->splay(); // now pull virtual children
    if (pre) v->vsub(pre, false);
    if (v->ch[1]) v->vsub(v->ch[1], true);
    v->ch[1] = pre, v->pull(), pre = v;
  }
  u->splay();
}

void rootify(Splay u) {
  // make u root of LCT component
  access(u), u->rev ^= 1, access(u);
  assert(!u->ch[0] && !u->ch[1]);
}

Splay lca(Splay u, Splay v) {
  if (u == v) return u;
  access(u), access(v);
  if (!u->p) return NULL;
  return u->splay(), u->p ?: u;
}

bool connected(Splay u, Splay v) {
  return lca(u, v) != NULL;
}

void link(Splay u, Splay v) { // make u parent of v
  if (!connected(u, v)) {
    rootify(v), access(u);
    add(v, u, 0), v->pull();
  }
}

void cut(Splay u) {
  // cut u from its parent
  access(u);
  u->ch[0]->p = u->ch[0] = NULL;
  u->pull();
}

void cut(Splay u, Splay v) { // if u, v are adjacent
     in the tree
  cut(depth(u) > depth(v) ? u : v);
}

int depth(Splay u) {
  access(u);
  return gsz(u->ch[0]);
}

Splay getRoot(Splay u) { // get root of LCT component
  access(u);
  while (u->ch[0]) u = u->ch[0], u->push();
  return access(u), u;
}

Splay ancestor(Splay u, int k) {
  // get k-th parent on path to root
  k = depth(u) - k;
  assert(k >= 0);
  for (;; u->push()) {
    int sz = gsz(u->ch[0]);
    if (sz == k) return access(u), u;
    if (sz < k) k -= sz + 1, u = u->ch[1];
    else u = u->ch[0];
  }
  assert(0);
}

Splay query(Splay u, Splay v) {
  return rootify(u), access(v), v;
}
```

3 Flows

3.1 Dinic $\mathcal{O}(min(E \cdot flow, V^2E))$

If the network is massive, try to compress it by looking for patterns.

```
template <class F>
struct Dinic {
  struct Edge {
    int v, inv;
    F cap, flow;
    Edge(int v, F cap, int inv) : v(v), cap(cap), flow
        (0), inv(inv) {}
  };

  F eps = (F) 1e-9;
  int s, t, n, m = 0;
  vector< vector<Edge> > g;
  vi dist, ptr;

  Dinic(int n) : n(n), g(n), dist(n), ptr(n), s(n - 2)
      , t(n - 1) {}

  void add(int u, int v, F cap) {
    g[u].pb(Edge(v, cap, sz(g[v])));
    g[v].pb(Edge(u, 0, sz(g[u]) - 1));
    m += 2;
  }

  bool bfs() {
    fill(all(dist), -1);
    queue<int> qu({s});
    dist[s] = 0;
    while (sz(qu) && dist[t] == -1) {
      int u = qu.front();
      qu.pop();
      for (Edge &e : g[u]) if (dist[e.v] == -1)
        if (e.cap - e.flow > eps) {
          dist[e.v] = dist[u] + 1;
          qu.push(e.v);
        }
    }
    return dist[t] != -1;
  }

  F dfs(int u, F flow = numeric_limits<F>::max()) {
    if (flow <= eps || u == t)
      return max<F>(0, flow);
    for (int &i = ptr[u]; i < sz(g[u]); i++) {
      Edge &e = g[u][i];
      if (e.cap - e.flow > eps && dist[u] + 1 == dist[
          e.v]) {
        F pushed = dfs(e.v, min<F>(flow, e.cap - e.flow
            ));
        if (pushed > eps) {
          e.flow += pushed;
          g[e.v][e.inv].flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }
```

```
  F maxFlow() {
    F flow = 0;
    while (bfs()) {
      fill(all(ptr), 0);
      while (F pushed = dfs(s))
        flow += pushed;
    }
    return flow;
  }
};
```

3.2 Min cost flow $\mathcal{O}(min(E \cdot flow, V^2E))$

If the network is massive, try to compress it by looking for patterns.

```
template <class C, class F>
struct Mcmf {
  struct Edge {
    int u, v, inv;
    F cap, flow;
    C cost;
    Edge(int u, int v, C cost, F cap, int inv) : u(u),
        v(v), cost(cost), cap(cap), flow(0), inv(inv
        ) {}
  };

  F eps = (F) 1e-9;
  int s, t, n, m = 0;
  vector< vector<Edge> > g;
  vector<Edge*> prev;
  vector<C> cost;
  vi state;

  Mcmf(int n) : n(n), g(n), cost(n), state(n), prev(n)
      , s(n - 2), t(n - 1) {}

  void add(int u, int v, C cost, F cap) {
    g[u].pb(Edge(u, v, cost, cap, sz(g[v])));
    g[v].pb(Edge(v, u, -cost, 0, sz(g[u]) - 1));
    m += 2;
  }

  bool bfs() {
    fill(all(state), 0);
    fill(all(cost), numeric_limits<C>::max());
    deque<int> qu;
    qu.push_back(s);
    state[s] = 1, cost[s] = 0;
    while (sz(qu)) {
      int u = qu.front(); qu.pop_front();
      state[u] = 2;
      for (Edge &e : g[u]) if (e.cap - e.flow > eps)
        if (cost[u] + e.cost < cost[e.v]) {
          cost[e.v] = cost[u] + e.cost;
          prev[e.v] = &e;
          if (state[e.v] == 2 || (sz(qu) && cost[qu.
              front()] > cost[e.v]))
            qu.push_front(e.v);
          else if (state[e.v] == 0)
            qu.push_back(e.v);
          state[e.v] = 1;
        }
    }
    return cost[t] != numeric_limits<C>::max();
  }

  pair<C, F> minCostFlow() {
    C cost = 0; F flow = 0;
    while (bfs()) {
      F pushed = numeric_limits<F>::max();
      for (Edge* e = prev[t]; e != nullptr; e = prev[e
          ->u])
        pushed = min(pushed, e->cap - e->flow);
      for (Edge* e = prev[t]; e != nullptr; e = prev[e
          ->u]) {
        e->flow += pushed;
        g[e->v][e->inv].flow -= pushed;
        cost += e->cost * pushed;
      }
      flow += pushed;
    }
    return make_pair(cost, flow);
  }
};
```

3.3 Hopcroft-Karp $\mathcal{O}(E\sqrt{V})$

```
struct HopcroftKarp {
  int n, m = 0;
  vector<vi> g;
  vi dist, match;

  HopcroftKarp(int _n) : n(_n + 5), g(n), dist(n),
      match(n, 0) {} // 1-indexed!!

  void add(int u, int v) {
    g[u].pb(v), g[v].pb(u);
    m += 2;
  }

  bool bfs() {
    queue<int> qu;
    fill(all(dist), -1);
    fore (u, 1, n)
      if (!match[u])
        dist[u] = 0, qu.push(u);
    while (!qu.empty()) {
      int u = qu.front(); qu.pop();
      for (int v : g[u])
        if (dist[match[v]] == -1) {
          dist[match[v]] = dist[u] + 1;
          if (match[v])
            qu.push(match[v]);
        }
    }
    return dist[0] != -1;
  }

  bool dfs(int u) {
    for (int v : g[u])
      if (!match[v] || (dist[u] + 1 == dist[match[v]]
          && dfs(match[v]))) {
        match[u] = v, match[v] = u;
        return 1;
      }
    dist[u] = 1 << 30;
    return 0;
  }

  int maxMatching() {
    int tot = 0;
    while (bfs())
      fore (u, 1, n)
        tot += match[u] ? 0 : dfs(u);
    return tot;
  }
};
```

3.4 Hungarian $\mathcal{O}(N^3)$

n jobs, m people

```
template <class C>
pair<C, vi> Hungarian(vector< vector<C> > &a) {
  int n = sz(a), m = sz(a[0]), p, q, j, k; // n <= m
```

```
  vector<C> fx(n, numeric_limits<C>::min()), fy(m, 0);
  vi x(n, -1), y(m, -1);
  fore (i, 0, n)
    fore (j, 0, m)
      fx[i] = max(fx[i], a[i][j]);
  fore (i, 0, n) {
    vi t(m, -1), s(n + 1, i);
    for (p = q = 0; p <= q && x[i] < 0; p++)
      for (k = s[p], j = 0; j < m && x[i] < 0; j++)
        if (abs(fx[k] + fy[j] - a[k][j]) < eps && t[j]
            < 0) {
          s[++q] = y[j], t[j] = k;
          if (s[q] < 0) for (p = j; p >= 0; j = p)
            y[j] = k = t[j], p = x[k], x[k] = j;
        }
    if (x[i] < 0) {
      C d = numeric_limits<C>::max();
      fore (k, 0, q + 1)
        fore (j, 0, m) if (t[j] < 0)
          d = min(d, fx[s[k]] + fy[j] - a[s[k]][j]);
      fore (j, 0, m)
        fy[j] += (t[j] < 0 ? 0 : d);
      fore (k, 0, q + 1)
        fx[s[k]] -= d;
      i--;
    }
  }
  C cost = 0;
  fore (i, 0, n) cost += a[i][x[i]];
  return make_pair(cost, x);
}
```

4 Strings

4.1 Hash

```
vi mod = {999727999, 999992867, 1000000123, 1000002193
    , 1000003211, 1000008223, 1000009999, 1000027163,
    1070777777};

struct H : array<lli, 2> {
  #define oper(op) friend H operator op (H a, H b) { \
  fore (i, 0, sz(a)) a[i] = (a[i] op b[i] + mod[i]) %
      mod[i]; \
  return a; }
  oper(+) oper(-) oper(*)
} pw[N], ipw[N];

struct Hash {
  vector<H> h;

  Hash(string &s) : h(sz(s) + 1) {
    fore (i, 0, sz(s)) {
      int x = s[i] - 'a' + 1;
      h[i + 1] = h[i] + pw[i] * H{x, x};
    }
  }

  H cut(int l, int r) {
    return (h[r + 1] - h[l]) * ipw[l];
  }
};

const int P = uniform_int_distribution<int>(27, min(
    mod[0], mod[1]) - 1)(rng);
pw[0] = ipw[0] = {1, 1};
H Q = {inv(P, mod[0]), inv(P, mod[1])};
fore (i, 1, N) {
  pw[i] = pw[i - 1] * H{P, P};
  ipw[i] = ipw[i - 1] * Q;
}
```

```
// Save {l, r} in the struct and when you do a cut
H merge(vector<H> &cuts) {
  F f = {0, 0};
  fore (i, sz(cuts), 0) {
    F g = cuts[i];
    f = g + f * pw[g.r - g.l + 1];
  }
  return f;
}
```

4.2 KMP

period $= n - p[n-1]$, period$(abcabc) = 3$, $n \mod period \equiv 0$

```
vi lps(string &s) {
  vi p(sz(s), 0);
  int j = 0;
  fore (i, 1, sz(s)) {
    while (j && s[i] != s[j])
      j = p[j - 1];
    if (s[i] == s[j])
      j++;
    p[i] = j;
  }
  return p;
}
// how many times t occurs in s
int kmp(string &s, string &t) {
  vi p = lps(t);
  int j = 0, tot = 0;
  fore (i, 0, sz(s)) {
    while (j && s[i] != t[j])
      j = p[j - 1];
    if (s[i] == t[j])
      j++;
    if (j == sz(t))
      tot++; // pos: i - sz(t) + 1;
  }
  return tot;
}
```

4.3 KMP automaton

```
int go[N][A];

void kmpAutomaton(string &s) {
  s += "$";
  vi p = lps(s);
  fore (i, 0, sz(s))
    fore (c, 0, A) {
      if (i && s[i] != 'a' + c)
        go[i][c] = go[p[i - 1]][c];
      else
        go[i][c] = i + ('a' + c == s[i]);
    }
  s.pop_back();
}
```

4.4 Z algorithm

```
vi zf(string &s) {
  vi z(sz(s), 0);
  for (int i = 1, l = 0, r = 0; i < sz(s); i++) {
    if (i <= r)
      z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < sz(s) && s[i + z[i]] == s[z[i]])
      ++z[i];
    if (i + z[i] - 1 > r)
      l = i, r = i + z[i] - 1;
  }
  return z;
}
```

4.5 Manacher algorithm

```
vector<vi> manacher(string &s) {
```

```
  vector<vi> pal(2, vi(sz(s), 0));
  fore (k, 0, 2) {
    int l = 0, r = 0;
    fore (i, 0, sz(s)) {
      int t = r - i + !k;
      if (i < r)
        pal[k][i] = min(t, pal[k][l + t]);
      int p = i - pal[k][i], q = i + pal[k][i] - !k;
      while (p >= 1 && q + 1 < sz(s) && s[p - 1] == s[
          q + 1])
        ++pal[k][i], --p, ++q;
      if (q > r)
        l = p, r = q;
    }
  }
  return pal;
}
```

4.6 Suffix array

- Duplicates $\sum_{i=1}^{n} lcp[i]$
- Longest Common Substring of various strings
  Add $notUsed$ characters between strings, i.e. $a+\$+b+\#+c$
  Use two-pointers to find a range $[l, r]$ such that all $notUsed$ characters are present, then $query(lcp[l+1], .., lcp[r])$ for that window is the common length.

```
struct SuffixArray {
  int n;
  string s;
  vi sa, lcp;

  SuffixArray(string &s) : n(sz(s) + 1), s(s), sa(n),
      lcp(n) {
    vi top(max(256, n)), rk(n);
    fore (i, 0, n)
      top[rk[i] = s[i] & 255]++;
    partial_sum(all(top), top.begin());
    fore (i, 0, n)
      sa[--top[rk[i]]] = i;
    vi sb(n);
    for (int len = 1, j; len < n; len <<= 1) {
      fore (i, 0, n) {
        j = (sa[i] - len + n) % n;
        sb[top[rk[j]]++] = j;
      }
      sa[sb[top[0] = 0]] = j = 0;
      fore (i, 1, n) {
        if (rk[sb[i]] != rk[sb[i - 1]] || rk[sb[i] +
            len] != rk[sb[i - 1] + len])
          top[++j] = i;
        sa[sb[i]] = j;
      }
      copy(all(sa), rk.begin());
      copy(all(sb), sa.begin());
      if (j >= n - 1)
        break;
    }
    for (int i = 0, j = rk[lcp[0] = 0], k = 0; i < n -
        1; i++, k++)
      while (k >= 0 && s[i] != s[sa[j - 1] + k])
        lcp[j] = k--, j = rk[sa[j] + 1];
  }

  char at(int i, int j) {
    int k = sa[i] + j;
    return k < n ? s[k] : 'z' + 1;
  }

  int count(string &t) {
    ii lo(0, n - 1), hi(0, n - 1);
    fore (i, 0, sz(t)) {
      while (lo.f + 1 < lo.s) {
        int mid = (lo.f + lo.s) / 2;
        (at(mid, i) < t[i] ? lo.f : lo.s) = mid;
      }
      while (hi.f + 1 < hi.s) {
        int mid = (hi.f + hi.s) / 2;
        (t[i] < at(mid, i) ? hi.s : hi.f) = mid;
      }
      int p1 = (at(lo.f, i) == t[i] ? lo.f : lo.s);
      int p2 = (at(hi.s, i) == t[i] ? hi.s : hi.f);
      if (at(p1, i) != t[i] || at(p2, i) != t[i] || p1
          > p2)
        return 0;
      lo = hi = ii(p1, p2);
    }
    return lo.s - lo.f + 1;
  }
};
```

4.7 Suffix automaton

- $sam[u].len - sam[sam[u].link].len$ = distinct strings
- Number of different substrings (dp)

$$diff(u) = 1 + \sum_{v \in trie[u]} diff(v)$$

- Total length of all different substrings (2 x dp)

$$totLen(u) = \sum_{v \in trie[u]} diff(v) + totLen(v)$$

- Leftmost occurrence $trie[u].pos = trie[u].len - 1$
  if it is **clone** then $trie[clone].pos = trie[q].pos$
- All occurrence positions
- Smallest cyclic shift
  Construct sam of $s + s$, find the lexicographically smallest path of $sz(s)$
- Shortest non-appearing string

$$nonAppearing(u) = \min_{v \in trie[u]} nonAppearing(v) + 1$$

```
struct SuffixAutomaton {
  struct Node : map<char, int> {
    int link = -1, len = 0;
  };

  vector<Node> trie;
  int last;

  SuffixAutomaton() { last = newNode(); }

  int newNode() {
    trie.pb({});
    return sz(trie) - 1;
  }

  void extend(char c) {
    int u = newNode();
    trie[u].len = trie[last].len + 1;
    int p = last;
    while (p != -1 && !trie[p].count(c)) {
      trie[p][c] = u;
      p = trie[p].link;
    }
    if (p == -1)
      trie[u].link = 0;
    else {
      int q = trie[p][c];
      if (trie[p].len + 1 == trie[q].len)
        trie[u].link = q;
      else {
        int clone = newNode();
```

```
      trie[clone] = trie[q];
      trie[clone].len = trie[p].len + 1;
      while (p != -1 && trie[p][c] == q) {
        trie[p][c] = clone;
        p = trie[p].link;
      }
      trie[q].link = trie[u].link = clone;
    }
  }
  last = u;
}

string kthSubstring(lli kth, int u = 0) {
  // number of different substrings (dp)
  string s = "";
  while (kth > 0)
    for (auto &[c, v] : trie[u]) {
      if (kth <= diff(v)) {
        s.pb(c), kth--, u = v;
        break;
      }
      kth -= diff(v);
    }
  return s;
}

void occurs() {
  // trie[u].occ = 1, trie[clone].occ = 0
  vi who;
  fore (u, 1, sz(trie))
    who.pb(u);
  sort(all(who), [&](int u, int v) {
    return trie[u].len > trie[v].len;
  });
  for (int u : who) {
    int l = trie[u].link;
    trie[l].occ += trie[u].occ;
  }
}

lli queryOccurences(string &s, int u = 0) {
  for (char c : s) {
    if (!trie[u].count(c))
      return 0;
    u = trie[u][c];
  }
  return trie[u].occ;
}

int longestCommonSubstring(string &s, int u = 0) {
  int mx = 0, clen = 0;
  for (char c : s) {
    while (u && !trie[u].count(c)) {
      u = trie[u].link;
      clen = trie[u].len;
    }
    if (trie[u].count(c))
      u = trie[u][c], clen++;
    mx = max(mx, clen);
  }
  return mx;
}

string smallestCyclicShift(int n, int u = 0) {
  string s = "";
  fore (i, 0, n) {
    char c = trie[u].begin()->f;
    s += c;
    u = trie[u][c];
  }
    return s;
  }

  int leftmost(string &s, int u = 0) {
    for (char c : s) {
      if (!trie[u].count(c))
        return -1;
      u = trie[u][c];
    }
    return trie[u].pos - sz(s) + 1;
  }

  Node& operator [](int u) {
    return trie[u];
  }
};
```

4.8 Aho corasick

```
struct AhoCorasick {
  struct Node : map<char, int> {
    int link = 0, out = 0;
    int cnt = 0, isw = 0;
  };

  vector<Node> trie;

  AhoCorasick() { newNode(); }

  int newNode() {
    trie.pb({});
    return sz(trie) - 1;
  }

  void insert(string &s, int u = 0) {
    for (char c : s) {
      if (!trie[u][c])
        trie[u][c] = newNode();
      u = trie[u][c];
    }
    trie[u].cnt++, trie[u].isw = 1;
  }

  int go(int u, char c) {
    while (u && !trie[u].count(c))
      u = trie[u].link;
    return trie[u][c];
  }

  void pushLinks() {
    queue<int> qu;
    qu.push(0);
    while (!qu.empty()) {
      int u = qu.front();
      qu.pop();
      for (auto &[c, v] : trie[u]) {
        int l = (trie[v].link = u ? go(trie[u].link, c
            ) : 0);
        trie[v].cnt += trie[l].cnt;
        trie[v].out = trie[l].isw ? l : trie[l].out;
        qu.push(v);
      }
    }
  }

  int match(string &s, int u = 0) {
    int ans = 0;
    for (char c : s) {
      u = go(u, c);
      ans += trie[u].cnt;
      for (int x = u; x != 0; x = trie[x].out)
        // pass over all elements of the implicit
```

```
        vector
    }
    return ans;
  }

  Node& operator [](int u) {
    return trie[u];
  }
};
```

4.9 Eertree

```
struct Eertree {
  struct Node : map<char, int> {
    int link = 0, len = 0;
  };

  vector<Node> trie;
  string s = "$";
  int last;

  Eertree() {
    last = newNode(), newNode();
    trie[0].link = 1, trie[1].len = -1;
  }

  int newNode() {
    trie.pb({});
    return sz(trie) - 1;
  }

  int go(int u) {
    while (s[sz(s) - trie[u].len - 2] != s.back())
      u = trie[u].link;
    return u;
  }

  void extend(char c) {
    s += c;
    int u = go(last);
    if (!trie[u][c]) {
      int v = newNode();
      trie[v].len = trie[u].len + 2;
      trie[v].link = trie[go(trie[u].link)][c];
      trie[u][c] = v;
    }
    last = trie[u][c];
  }

  Node& operator [](int u) {
    return trie[u];
  }
};
```

5 Dynamic Programming

5.1 All submasks of a mask

```
for (int B = A; B > 0; B = (B - 1) & A)
```

5.2 Matrix Chain Multiplication

```
int dp(int l, int r) {
  if (l > r)
    return 0LL;
  int &ans = mem[l][r];
  if (!done[l][r]) {
    done[l][r] = true, ans = inf;
    fore (k, l, r + 1) // split in [l, k] [k + 1, r]
      ans = min(ans, dp(l, k) + dp(k + 1, r) + add);
  }
  return ans;
}
```

5.3 Digit DP

Counts the amount of numbers in $[l, r]$ such are divisible by k. (flag *nonzero* is for different lengths)
It can be reduced to $dp(i, x, small)$, and has to be solve like $f(r) - f(l - 1)$

```
#define state [i][x][small][big][nonzero]
int dp(int i, int x, bool small, bool big, bool
    nonzero) {
  if (i == sz(r))
    return x % k == 0 && nonzero;
  int &ans = mem state;
  if (done state != timer) {
    done state = timer;
    ans = 0;
    int lo = small ? 0 : l[i] - '0';
    int hi = big ? 9 : r[i] - '0';
    fore (y, lo, max(lo, hi) + 1) {
      bool small2 = small | (y > lo);
      bool big2 = big | (y < hi);
      bool nonzero2 = nonzero | (x > 0);
      ans += dp(i + 1, (x * 10 + y) % k, small2, big2,
           nonzero2);
    }
  }
  return ans;
}
```

5.4 Knapsack 0/1

```
for (auto &cur : items)
  fore (w, W + 1, cur.w) // [cur.w, W]
    umax(dp[w], dp[w - cur.w] + cur.cost);
```

5.5 Convex Hull Trick $\mathcal{O}(n^2) \Rightarrow \mathcal{O}(n)$

$dp[i] = \min_{j<i}(dp[j] + b[j] * a[i])$
$dp[i][j] = \min_{k<j}(dp[i-1][k] + b[k] * a[j])$
$b[j] \geq b[j+1]$ optionally $a[i] \leq a[i+1]$

```
// for doubles, use inf = 1/.0, div(a,b) = a / b
struct Line {
  mutable lli m, c, p;
  bool operator < (const Line &l) const { return m < l
      .m; }
  bool operator < (lli x) const { return p < x; }
  lli operator ()(lli x) const { return m * x + c; }
};

struct DynamicHull : multiset<Line, less<>> {
  lli div(lli a, lli b) {
    return a / b - ((a ^ b) < 0 && a % b);
  }

  bool isect(iterator x, iterator y) {
    if (y == end())
      return x->p = inf, 0;
    if (x->m == y->m)
      x->p = (x->c > y->c ? inf : -inf);
    else
      x->p = div(x->c - y->c, y->m - x->m);
    return x->p >= y->p;
  }

  void add(lli m, lli c) {
    auto z = insert({m, c, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y))
      isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }

  lli query(lli x) {
```

```
    if (empty()) return 0LL;
    auto f = *lower_bound(x);
    return f(x);
  }
};
```

5.6 Divide and conquer $\mathcal{O}(kn^2) \Rightarrow \mathcal{O}(k \cdot nlogn)$

Split the array of size n into k continuous groups. $k \leq n$
$cost(a, c) + cost(b, d) \leq cost(a, d) + cost(b, c)$ with $a \leq b \leq c \leq d$

```
void dc(int cut, int l, int r, int optl, int optr) {
  if (r < l)
    return;
  int mid = (l + r) / 2;
  pair<lli, int> best = {inf, -1};
  fore (p, optl, min(mid, optr) + 1) {
    lli nxt = dp[~cut & 1][p - 1] + cost(p, mid);
    if (nxt < best.f)
      best = {nxt, p};
  }
  dp[cut & 1][mid] = best.f;
  int opt = best.s;
  dc(cut, l, mid - 1, optl, opt);
  dc(cut, mid + 1, r, opt, optr);
}

fore (i, 1, n + 1)
  dp[1][i] = cost(1, i);
fore (cut, 2, k + 1)
  dc(cut, cut, n, cut, n);
```

5.7 Knuth optimization $\mathcal{O}(n^3) \Rightarrow \mathcal{O}(n^2)$

$dp[l][r] = \min_{l \leq k \leq r}\{dp[l][k] + dp[k][r]\} + cost(l, r)$

```
fore (len, 1, n + 1)
  fore (l, 0, n) {
    int r = l + len - 1;
    if (r > n - 1)
      break;
    if (len <= 2) {
      dp[l][r] = 0;
      opt[l][r] = l;
      continue;
    }
    dp[l][r] = inf;
    fore (k, opt[l][r - 1], opt[l + 1][r] + 1) {
      lli cur = dp[l][k] + dp[k][r] + cost(l, r);
      if (cur < dp[l][r]) {
        dp[l][r] = cur;
        opt[l][r] = k;
      }
    }
  }
```

6 Game Theory

6.1 Grundy Numbers

If the moves are consecutive $S = \{1, 2, 3, ..., x\}$ the game can be solved like $stackSize \pmod{x + 1} \neq 0$

```
int mem[N];

int mex(set<int> &st) {
  int x = 0;
  while (st.count(x))
    x++;
  return x;
}
```

```
int grundy(int n) {
  if (n < 0)
    return inf;
  if (n == 0)
    return 0;
  int &g = mem[n];
  if (g == -1) {
    set<int> st;
    for (int x : {a, b})
      st.insert(grundy(n - x));
    g = mex(st);
  }
  return g;
}
```

7 Combinatorics

Combinatorics table		
Number	Factorial	Catalan
0	1	1
1	1	1
2	2	2
3	6	5
4	24	14
5	120	42
6	720	132
7	5,040	429
8	40,320	1,430
9	362,880	4,862
10	3,628,800	16,796
11	39,916,800	58,786
12	479,001,600	208,012
13	6,227,020,800	742,900

7.1 Factorial

```
fac[0] = 1LL;
fore (i, 1, N)
  fac[i] = lli(i) * fac[i - 1] % mod;
ifac[N - 1] = fpow(fac[N - 1], mod - 2);
fore (i, N - 1, 0)
  ifac[i] = lli(i + 1) * ifac[i + 1] % mod;
```

7.2 Factorial mod $smallPrime$

```
lli facMod(lli n, int p) {
  lli r = 1LL;
  for (; n > 1; n /= p) {
    r = (r * ((n / p) % 2 ? p - 1: 1)) % p;
    fore (i, 2, n % p + 1)
      r = r * i % p;
  }
  return r % p;
}
```

7.3 Lucas theorem

Changes $\binom{n}{k}$ mod p, with $n \geq 2e6, k \geq 2e6$ and $p \leq 1e7$

$$\binom{n}{k} \equiv \prod_{i=0}^{n} \binom{n_i}{k_i} \mod p$$

```
lli lucas(lli n, lli k) {
  if (k == 0)
    return 1LL;
  return lucas(n / mod, k / mod) * choose(n % mod, k %
       mod) % mod;
}
```

7.4 Stars and bars

Enclosing n objects in k boxes

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

7.5 N choose K

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

$$\binom{n}{k_1, k_2, ..., k_m} = \frac{n!}{k_1! * k_2! * ... * k_m!}$$

```
lli choose(int n, int k) {
  if (n < 0 || k < 0 || n < k)
    return 0LL;
  return fac[n] * ifac[k] % mod * ifac[n - k] % mod;
}

lli choose(int n, int k) {
  double r = 1;
  fore (i, 1, k + 1)
    r = r * (n - k + i) / i;
  return lli(r + 0.01);
}
```

7.6 Catalan

```
catalan[0] = 1LL;
fore (i, 0, N) {
  catalan[i + 1] = catalan[i] * lli(4 * i + 2) % mod *
       fpow(i + 2, mod - 2) % mod;
}
```

7.7 Burnside's lemma

$$|classes| = \frac{1}{|G|} \cdot \sum_{x \in G} f(x)$$

7.8 Prime factors of N!

```
vector< pair<lli, int> > factorialFactors(lli n) {
  vector< pair<lli, int> > fac;
  for (lli p : primes) {
    if (n < p)
      break;
    lli mul = 1LL, k = 0;
    while (mul <= n / p) {
      mul *= p;
      k += n / mul;
    }
    fac.emplace_back(p, k);
  }
  return fac;
}
```

8 Number Theory

8.1 Goldbach conjecture

- All number ≥ 6 can be written as sum of 3 $primes$
- All even number > 2 can be written as sum of 2 $primes$

8.2 Prime numbers distribution

Amount of primes approximately $\frac{n}{\ln(n)}$

8.3 Sieve of Eratosthenes

Numbers up to $2e8$

```
int erat[N >> 6];
#define bit(i) ((i >> 1) & 31)
#define prime(i) !(erat[i >> 6] >> bit(i) & 1)

void bitSieve() {
  for (int i = 3; i * i < N; i += 2) if (prime(i))
    for (int j = i * i; j < N; j += (i << 1))
      erat[j >> 6] |= 1 << bit(j);
}
```

To factorize divide x by $factor[x]$ until is equal to 1

```
void factorizeSieve() {
  iota(factor, factor + N, 0);
  for (int i = 2; i * i < N; i++) if (factor[i] == i)
    for (int j = i * i; j < N; j += i)
      factor[j] = i;
}
```

Use it if you need a huge amount of $phi[x]$ up to some N

```
void phiSieve() {
  isp.set(); // bitset<N> is faster
  iota(phi, phi + N, 0);
  fore (i, 2, N) if (isp[i])
    for (int j = i; j < N; j += i) {
      isp[j] = (i == j);
```

```
      phi[j] /= i;
      phi[j] *= i - 1;
    }
}
```

8.4 Phi of euler

```
lli phi(lli n) {
  if (n == 1)
    return 0;
  lli r = n;
  for (lli i = 2; i * i <= n; i++)
    if (n % i == 0) {
      while (n % i == 0)
        n /= i;
      r -= r / i;
    }
  if (n > 1)
    r -= r / n;
  return r;
}
```

8.5 Miller-Rabin

```
bool compo(lli p, lli d, lli n, lli k) {
  lli x = fpow(p % n, d, n), i = k;
  while (x != 1 && x != n - 1 && p % n && i--)
    x = mul(x, x, n);
  return x != n - 1 && i != k;
}

bool miller(lli n) {
  if (n < 2 || n % 6 % 4 != 1)
    return (n | 1) == 3;
  int k = __builtin_ctzll(n - 1);
  lli d = n >> k;
  for (lli p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31
      , 37}) {
    if (compo(p, d, n, k))
      return 0;
    if (compo(2 + rng() % (n - 3), d, n, k))
      return 0;
  }
  return 1;
}
```

8.6 Pollard-Rho

```
lli rho(lli n) {
  while (1) {
    lli x = 2 + rng() % (n - 3), c = 1 + rng() % 20;
    auto f = [&](lli x) { return (mul(x, x, n) + c) %
        n; };
    lli y = f(x), g;
    while ((g = __gcd(n + y - x, n)) == 1)
      x = f(x), y = f(f(y));
    if (g != n) return g;
  }
  return -1;
}

void pollard(lli n, map<lli, int> &fac) {
  if (n == 1) return;
  if (n % 2 == 0) {
    fac[2]++;
    pollard(n / 2, fac);
    return;
  }
  if (miller(n)) {
    fac[n]++;
    return;
  }
  lli x = rho(n);
  pollard(x, fac);
  pollard(n / x, fac);
}
```

8.7 Amount of divisors

```
lli divs(lli n) {
  lli cnt = 1LL;
  for (lli p : primes) {
    if (p * p * p > n)
      break;
    if (n % p == 0) {
      lli k = 0;
      while (n > 1 && n % p == 0)
        n /= p, ++k;
      cnt *= (k + 1);
    }
  }
  lli sq = mysqrt(n); // A binary search, the last x *
      x <= n
  if (miller(n))
    cnt *= 2;
  else if (sq * sq == n && miller(sq))
    cnt *= 3;
  else if (n > 1)
    cnt *= 4;
  return cnt;
}
```

8.8 Bézout's identity

$a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n = g$
$g = \gcd(a_1, a_2, ..., a_n)$

8.9 GCD

$a \le b; \gcd(a + k, b + k) = \gcd(b - a, a + k)$

8.10 LCM

$x = p * lcm(a_1, a_2, ..., a_k) + q, 0 \le q \le lcm(a_1, a_2, ..., a_k)$
$x \pmod{a_i} \equiv q \pmod{a_i}$ as $a_i \mid lcm(a_1, a_2, ..., a_k)$

8.11 Euclid

```
pair<lli, lli> euclid(lli a, lli b) {
  if (b == 0)
    return {1, 0};
  auto p = euclid(b, a % b);
  return {p.s, p.f - a / b * p.s};
}
```

8.12 Chinese remainder theorem

```
pair<lli, lli> crt(pair<lli, lli> a, pair<lli, lli> b)
    {
  if (a.s < b.s)
    swap(a, b);
  auto p = euclid(a.s, b.s);
  lli g = a.s * p.f + b.s * p.s, l = a.s / g * b.s;
  if ((b.f - a.f) % g != 0)
    return {-1, -1}; // no solution
 p.f = a.f + (b.f - a.f) % b.s * p.f % b.s / g * a.s;
  return {p.f + (p.f < 0) * l, l};
}
```

9 Math

9.1 Progressions

Arithmetic progressions

$$a_n = a_1 + (n - 1) * diff$$

$$\sum_{i=1}^{n} a_i = n * \frac{a_1 + a_n}{2}$$

Geometric progressions

$$a_n = a_1 * r^{n-1}$$

$$\sum_{k=1}^{n} a_1 * r^k = a_1 * \left(\frac{r^{n+1} - 1}{r - 1}\right) : r \ne 1$$

9.2 Mod multiplication

```
lli mul(lli x, lli y, lli mod) {
  lli r = 0LL;
  for (x %= mod; y > 0; y >>= 1) {
    if (y & 1) r = (r + x) % mod;
    x = (x + x) % mod;
  }
  return r;
}
```

9.3 Fpow

```
lli fpow(lli x, lli y, lli mod) {
  lli r = 1;
  for (; y > 0; y >>= 1) {
    if (y & 1) r = mul(r, x, mod);
    x = mul(x, x, mod);
  }
  return r;
}
```

9.4 Fibonacci

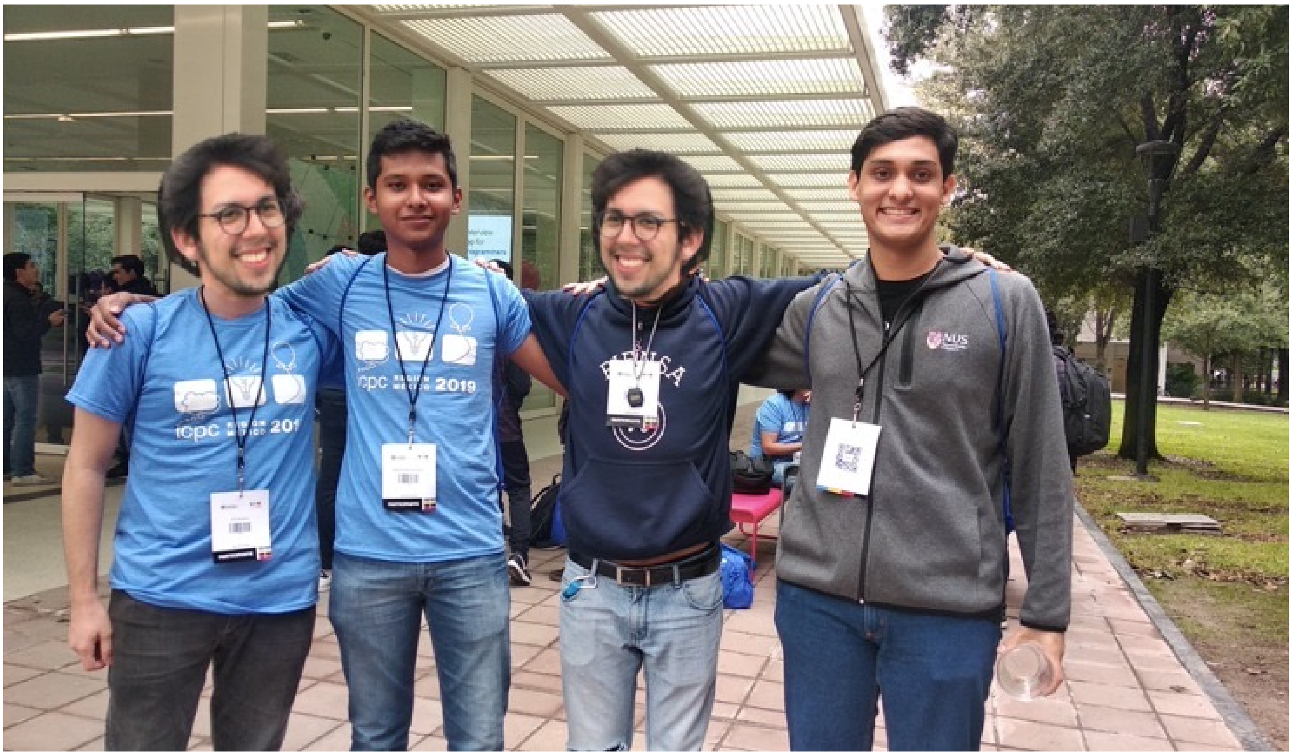
$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} fib_{n+1} & fib_n \\ fib_n & fib_{n-1} \end{bmatrix}$$

10 Bit tricks

Bits++	
Operations on *int*	Function
`x & -x`	Least significant bit in x
`__lg(x)`	Most significant bit in x
`c = x&-x, r = x+c; (((r^x) » 2)/c) | r`	Next number after x with same number of bits set
__builtin_	Function
popcount(x)	Amount of 1's in x
clz(x)	0's to the **left** of biggest bit
ctz(x)	0's to the **right** of smallest bit

10.1 Bitset

Bitset<Size>	
Operation	Function
_Find_first()	Least significant bit
_Find_next(idx)	First set bit after index *idx*
any(), none(), all()	Just what the expression says
set(), reset(), flip()	Just what the expression says x2
to_string('.', 'A')	Print 011010 like .AA.A.



The end...