Moscow Institute of Physics and Technology

Yolki-palki

Vsevolod Nagibin, Denis Mustafin, Tikhon Evteev

Adapted from KACTL

ICPC World Finals 2023

April 18, 2024

Mathematics (1)

1.1 Sums

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

1.2 Approximations

$$\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} = \ln(n) + \gamma + \frac{1}{2x} - \frac{1}{12x^2} + \frac{1}{120x^4} + \ldots$$

$$\gamma \approx 0.577215664901533$$

Data structures (2)

OrderStatisticTree.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change null_type.
Time: $\mathcal{O}(\log N)$

782797, 16 lines

```
#include <bits/extc++.h>
using namespace __gnu_pbds;

template<class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

void example() {
  Tree<int> t, t2; t.insert(8);
  auto it = t.insert(10).first;
  assert(it == t.lower_bound(9));
  assert(t.order_of_key(10) == 1);
  assert(t.order_of_key(11) == 2);
  assert(*t.find_by_order(0) == 8);
  t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

PersistentTreap.h

Description: Persistent treap with merge, split and finding element by index
Time: $\mathcal{O}(\log N)$

f18f45, 60 lines

```
struct node {
  int val;
  int l = 0, r = 0;
  int sz = 1;
  node() {}
  node(int val) : val(val) {}
};

const int MAXN = 6e7;
node tree[MAXN];
int c = 1;

int merge(int a, int b) {
  if (a == 0) return b;
  if (b == 0) return a;
  ll r = rand();  // int
  r %= (tree[a].sz + tree[b].sz);
  int c1 = c++;
  if (r < tree[a].sz) {
    tree[c1] = tree[a];
    tree[c1].sz += tree[b].sz;
    tree[c1].r = merge(tree[c1].r, b);
  } else {
    tree[c1] = tree[b];
    tree[c1].sz += tree[a].sz;
    tree[c1].l = merge(a, tree[c1].l);
  }
  return c1;
}

pii split(int a, ll sz) {  // first.sz = sz
  if (a == 0) return {0, 0};
  int c1 = c++;
  tree[c1] = tree[a];
  a = c1;
  pii p;
  if (tree[tree[a].l].sz >= sz) {
    p = split(tree[a].l, sz);
    tree[a].l = p.second;
    tree[a].sz -= tree[p.first].sz;
    p.second = a;
  } else {
    p = split(tree[a].r, sz - tree[tree[a].l].sz - 1);
    tree[a].r = p.first;
    tree[a].sz -= tree[p.second].sz;
    p.first = a;
  }
  return p;
}

int get(int a, ll pos) {
  if (a == 0) return 0;
  if (tree[tree[a].l].sz == pos) {
    return tree[a].val;
  } else if (tree[tree[a].l].sz > pos) {
    return get(tree[a].l, pos);
  } else {
    return get(tree[a].r, pos - tree[tree[a].l].sz - 1);
  }
}
```

KineticSegmentTree.h

Description: Initially there is an array of linear functions with pointers at x=0 update changes a line at ind to l with pointer at x=0 heaten advences pointers of lines from [l; r) by d get finds the minimum value at current pointer for lines in [l; r) requireres define int long long!
Time: $\mathcal{O}\left(N \log^? N\right)$ 186ms for n, q = 1e5

25691a, 127 lines

```
#pragma once

const int INFX = 2e12, INFY = 2e18;

struct Line {
  int k, b;

  Line(int k = 0, int b = 0): k(k), b(b) {}

  int operator()(int x) {
    return k * x + b;
  }
};

struct LazyKST {

  int n;
  vector<Line> tree;
  vector<int> melt;
  vector<int> add;

  void pull(int v) {
    int v1 = v * 2, v2 = v * 2 + 1;
    if (tree[v1].b < tree[v2].b || (tree[v1].b == tree[v2].b &&
        tree[v1].k < tree[v2].k)) {
      swap(v1, v2);
    }
    tree[v] = tree[v2];
    melt[v] = min(melt[v1], melt[v2]);
    if (tree[v1].k < tree[v2].k) {
      int x = (tree[v1].b - tree[v2].b + tree[v2].k - tree[v1].
          k - 1) / (tree[v2].k - tree[v1].k);
      melt[v] = min(melt[v], x);
    }
  }

  void push(int v) {
    int d = add[v];
    add[v] = 0;
    if (v * 2 < 4 * n) {
      add[v * 2] += d;
      add[v * 2 + 1] += d;
      melt[v * 2] -= d;
      tree[v * 2].b += tree[v * 2].k * d;
      melt[v * 2 + 1] -= d;
      tree[v * 2 + 1].b += tree[v * 2 + 1].k * d;
    }
  }

  void build(int v, int vl, int vr, vector<Line>& arr) {
    add[v] = 0;
    if (vr - vl == 1) {
      tree[v] = arr[vl];
      melt[v] = INFX;
      return;
    }
    int vm = (vl + vr) / 2;
    build(v * 2, vl, vm, arr);
    build(v * 2 + 1, vm, vr, arr);
    pull(v);
  }
  LazyKST(vector<Line> arr = {}): n(arr.size()), tree(n * 4),
      melt(n * 4), add(n * 4) {
    build(1, 0, n, arr);
  }

  void rec_upd(int v, int vl, int vr, int ind, Line val) {
    push(v);
    if (ind >= vr || ind < vl) return;
    if (vr - vl == 1) {
      tree[v] = val;
      return;
    }
    int vm = (vl + vr) / 2;
    rec_upd(v * 2, vl, vm, ind, val);
    rec_upd(v * 2 + 1, vm, vr, ind, val);
    pull(v);
  }

  void upd(int ind, Line l) {
    rec_upd(1, 0, n, ind, l);
  }

  void propagate(int v, int vl, int vr) {
    if (melt[v] > 0) return;
    int vm = (vl + vr) / 2;
    push(v);
    propagate(v * 2, vl, vm);
    propagate(v * 2 + 1, vm, vr);
    pull(v);
  }

  void rec_heaten(int v, int vl, int vr, int l, int r, int d) {
    if (l >= vr || r <= vl) return;
    if (l <= vl && r >= vr) {
      tree[v].b += tree[v].k * d;
      if (vr - vl > 1) {
        melt[v] -= d;
        add[v] += d;
      }
      propagate(v, vl, vr);
      return;
    }
    int vm = (vl + vr) / 2;
    push(v);
    rec_heaten(v * 2, vl, vm, l, r, d);
    rec_heaten(v * 2 + 1, vm, vr, l, r, d);
    pull(v);
  }

  void heaten(int l, int r, int d) {
    rec_heaten(1, 0, n, l, r, d);
  }

  int rec_get(int v, int vl, int vr, int l, int r) {
    if (l >= vr || r <= vl) return INFY;
    if (l <= vl && r >= vr) return tree[v].b;
    int vm = (vl + vr) / 2;
    push(v);
    int res1 = rec_get(v * 2, vl, vm, l, r);
    int res2 = rec_get(v * 2 + 1, vm, vr, l, r);
    return min(res1, res2);
  }

  int get(int l, int r) {
    return rec_get(1, 0, n, l, r);
  }
};
```

Numerical (3)

3.1 Polynomials and recurrences

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size $\leq n$.
Usage: `berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}`
Time: $\mathcal{O}(N^2)$

"../number-theory/ModPow.h" 96548b, 20 lines

```
vector<ll> berlekampMassey(vector<ll> s) {
  int n = sz(s), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;

  ll b = 1;
  rep(i,0,n) { ++m;
    ll d = s[i] % mod;
    rep(j,1,L+1) d = (d + C[j] * s[i - j]) % mod;
    if (!d) continue;
    T = C; ll coef = d * modpow(b, mod-2) % mod;
    rep(j,m,n) C[j] = (C[j] - coef * B[j - m]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }

  C.resize(L + 1); C.erase(C.begin());
  for (ll& x : C) x = (mod - x) % mod;
  return C;
}
```

3.2 Optimization

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
Usage:
```
vvd A = {{1,-1}, {-1,1}, {-1,-2}};
vd b = {1,1,-4}, c = {-1,-1}, x;
T val = LPSolver(A, b, c).solve(x);
```
Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case.

aa8530, 68 lines

```
typedef double T; // long double, Rational, double + mod<P>...
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s])) s=j

struct LPSolver {
  int m, n;
  vi N, B;
  vvd D;

  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
      rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
      rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
      rep(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
      N[n] = -1; D[m+1][n] = 1;
    }

  void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      rep(j,0,n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    rep(j,0,n+2) if (j != s) D[r][j] *= inv;
    rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool simplex(int phase) {
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      rep(i,0,m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                     < MP(D[r][n+1] / D[r][s], B[r])) r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }

  T solve(vd &x) {
    int r = 0;
    rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
      rep(i,0,m) if (B[i] == -1) {
        int s = 0;
        rep(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  }
};
```

3.3 Fourier transforms

FastFourierTransform.h

Description: Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs).
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim ?s$ for $N = 2^{22}$)

d2bfef, 38 lines

```
using cd = complex<double>; // better implement by hand

void fft(vector<cd>& a, bool inv = false) {
  int n = a.size();
  int k = 0;
  while ((1 << k) < n) ++k;
  static vector<int> rev;
  static vector<cd> power = {0, 1};
  rev.resize(n);
  rev[0] = 0;
  for (int i = 1; i < n; ++i) {
    rev[i] = rev[i / 2] / 2 + ((i & 1) << (k - 1));
    if (i < rev[i]) swap(a[i], a[rev[i]]);
  }
  for (int l = 1; l < n; l *= 2) {
    if ((int)power.size() == l) {
      power.resize(2 * l);
      complex<long double> w = polar(1.0L, acos(-1.0L) / l);
      cd wcd = {(double)w.real(), (double)w.imag()};
      for (int i = l; i < 2 * l; ++i) {
        power[i] = power[i / 2];
        if (i & 1) power[i] *= w;
      }
    }
    for (int i = 0; i < n; i += 2 * l) {
      for (int j = 0; j < l; ++j) {
        cd x = a[i + j], y = a[i + j + l] * power[j + l];
        a[i + j] = x + y;
        a[i + j + l] = x - y;
      }
    }
  }
  if (inv) {
    reverse(a.begin() + 1, a.end());
    double anti = 1.0L / n;
    for (cd& x : a) x *= anti;
  }
}
```

FastFourierTransformMod.h

Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT)

"FastFourierTransform.h" b82773, 22 lines

```
typedef vector<ll> vl;
template<int M> vl convMod(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  vl res(sz(a) + sz(b) - 1);
  int B=32-__builtin_clz(sz(res)), n=1<<B, cut=int(sqrt(M));
  vector<C> L(n), R(n), outs(n), outl(n);
  rep(i,0,sz(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
  rep(i,0,sz(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
  fft(L), fft(R);
  rep(i,0,n) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
  }
  fft(outl), fft(outs);
  rep(i,0,sz(res)) {
    ll av = ll(real(outl[i])+.5), cv = ll(imag(outs[i])+.5);
    ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);
    res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
  }
  return res;
}
```

NumberTheoreticTransform.h

Description: fft(a, 0) computes direct dft, fft(a, 1) - inverted. Inputs must be in [0, mod), returns in [0, mod). Implemented with some foreign optimizations
Time: $\mathcal{O}(N \log N)$, multiplies polynomials of size 1 << 19 in 117 ms (-blazingio for i/o)

b8ef89, 62 lines

```
const int G1 = 3;

void fft(vector<int>& A, bool inv = false) {
  int k = 0;
  while ((1 << k) < (int)A.size())
    ++k;
  int N = 1 << k;
  static vector<int> rev;
  static vector<uint32_t> power = {0, 1};
  rev.resize(N);
  for (int i = 0; i < N; ++i) {
    rev[i] = rev[i / 2] / 2 + ((i & 1) << (k - 1));
    if (i < rev[i]) swap(A[i], A[rev[i]]);
  }

  static auto mul = [](uint32_t a, uint32_t b) {
    return (uint64_t(a) * b) % MOD;
  };

  static vector<uint32_t> A1;
  A1.resize(N);
  for (int i = 0; i < N; ++i)
    A1[i] = A[i];

  for (int l = 1, t = 0; l < N; l *= 2, ++t) {
    if ((int)power.size() == l) {
      power.resize(2 * l);
      uint32_t w = pw(G1, (MOD - 1) / 2 / l);
      for (int i = l; i < 2 * l; ++i) {
        power[i] = power[i / 2];
        if (i & 1) power[i] = mul(power[i], w);
      }
    }
    if ((k - t - 1) & 3) {
      for (int i = 0; i < N; i += 2 * l) {
        for (int j = 0; j < l; ++j) {
          uint32_t x = A1[i + j], y = mul(power[j + l], A1[i +
               j + l]);
          A1[i + j] = x + y;
          A1[i + j + l] = x + MOD - y;
        }
      }
    } else {
      for (int i = 0; i < N; i += 2 * l) {
        for (int j = 0; j < l; ++j) {
          uint32_t x = A1[i + j], y = mul(power[j + l], A1[i +
               j + l]);
          A1[i + j] = (uint64_t(x) + y) % MOD;
          A1[i + j + l] = (uint64_t(x) + MOD - y) % MOD;
        }
      }
    }
  }

  for (int i = 0; i < N; ++i)
    A[i] = A1[i];

  if (inv) {
    reverse(A.begin() + 1, A.end());
    int anti = pw(N, MOD - 2);
    for (int i = 0; i < N; ++i)
      A[i] = mul(A[i], anti);
  }
}
```

FastExponent.h

Description: Works faster than naive implemtntation with ln
Time: $\mathcal{O}(N \log N)$

e76419, 26 lines

```
void exp_step(vector<int>& f, vector<int>& g, const vector<int
    >& h) {
  int m = f.size();
  g = g + g - f * (g * g);
  g.resize(m);
  vector<int> q(m, 0);
  for (int i = 0; i < m - 1 && i + 1 < (int)h.size(); ++i) {
    q[i] = mul(h[i + 1], i + 1);
  }
  vector<int> w = q + g * (der(f) - f * q);
  w.resize(2 * m - 1);
  vector<int> h1(2 * m);
  for (int i = 0; i < 2 * m && i < (int)h.size(); ++i) {
    h1[i] = h[i];
  }
  f = f + f * (h1 - integ(w));
  f.resize(2 * m);
}

vector<int> exp(vector<int> h, int n) {
  vector<int> f = {1}, g = {1};
  for (int m = 1; m < n; m *= 2) {
    exp_step(f, g, h);
  }
  f.resize(n);
  return f;
}
```

Number theory (4)

4.1 Modular arithmetic

ModSum.h

Description: Sums of mod'ed arithmetic progressions.
modsum(to, c, k, m) = $\sum_{i=0}^{\text{to}-1} (ki + c)\%m$. divsum is similar but for floored division.
Time: $\log(m)$, with a large constant.

5c5bc5, 16 lines

```
typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }

ull divsum(ull to, ull c, ull k, ull m) {
  ull res = k / m * sumsq(to) + c / m * to;
  k %= m; c %= m;
  if (!k) return res;
  ull to2 = (to * k + c) / m;
  return res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k);
}

ll modsum(ull to, ll c, ll k, ll m) {
  c = ((c % m) + m) % m;
  k = ((k % m) + m) % m;
  return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
}
```

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \le a, b \le c \le 7.2 \cdot 10^{18}$.
Time: $\mathcal{O}(1)$ for modmul, $\mathcal{O}(\log b)$ for modpow

bbbd8f, 11 lines

```
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(1.L / M * a * b);
  return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = modmul(b, b, mod), e /= 2)
    if (e & 1) ans = modmul(ans, b, mod);
  return ans;
}
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution).
Time: $\mathcal{O}(\log^2 p)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h" 19a793, 24 lines

```
ll sqrt(ll a, ll p) {
  a %= p; if (a < 0) a += p;
  if (a == 0) return 0;
  assert(modpow(a, (p-1)/2, p) == 1); // else no solution
  if (p % 4 == 3) return modpow(a, (p+1)/4, p);
  // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
  ll s = p - 1, n = 2;
  int r = 0, m;
  while (s % 2 == 0)
    ++r, s /= 2;
  while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
  ll x = modpow(a, (s + 1) / 2, p);
  ll b = modpow(a, s, p), g = modpow(n, s, p);
  for (;; r = m) {
    ll t = b;
    for (m = 0; m < r && t != 1; ++m)
      t = t * t % p;
    if (m == 0) return x;
    ll gs = modpow(g, 1LL << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
  }
}
```

4.2 Primality

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.

Time: 7 times the complexity of $a^b \mod c$.

"ModMulLL.h" 60dcd1, 12 lines

```
bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
      s = __builtin_ctzll(n-1), d = n >> s;
  for (ull a : A) {   // ^ count trailing zeroes
    ull p = modpow(a%n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i--)
      p = modmul(p, p, n);
    if (p != n-1 && i != s) return 0;
  }
  return 1;
}
```

Factor.h

Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).

Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.

"ModMulLL.h", "MillerRabin.h" a33cf6, 18 lines

```
ull pollard(ull n) {
  auto f = [n](ull x) { return modmul(x, x, n) + 1; };
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    if ((q = modmul(prd, max(x,y) - min(x,y), n))) prd = q;
    x = f(x), y = f(f(y));
  }
  return __gcd(prd, n);
}
vector<ull> factor(ull n) {
  if (n == 1) return {};
  if (isPrime(n)) return {n};
  ull x = pollard(n);
  auto l = factor(x), r = factor(n / x);
  l.insert(l.end(), all(r));
  return l;
}
```

4.3 Fractions

ContinuedFractions.h

Description: Given N and a real number $x \geq 0$, finds the closest rational approximation p/q with $p, q \leq N$. It will obey $|p/q - x| \leq 1/qN$.

For consecutive convergents, $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$. ($p_k/q_k$ alternates between $> x$ and $< x$.) If x is rational, y eventually becomes ∞; if x is the root of a degree 2 polynomial the a's eventually become cyclic.

4.4 Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

4.5 Estimates

$\sum_{d|n} d = O(n \log \log n)$.

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

4.6 Mobius Function

$g(n) = \sum_{1 \leq m \leq n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \leq m \leq n} \mu(m) g(\lfloor \frac{n}{m} \rfloor)$

Combinatorial (5)

5.1 Permutations

5.1.1 Burnside's lemma

Given a group G of symmetries and a set X, the number of elements of X *up to symmetry* equals

$$\frac{1}{|G|} \sum_{g \in G} |X^g|,$$

where X^g are the elements fixed by g ($g.x = x$).

If $f(n)$ counts "configurations" (of some sort) of length n, we can ignore rotational symmetry using $G = \mathbb{Z}_n$ to get

$$g(n) = \frac{1}{n} \sum_{k=0}^{n-1} f(\gcd(n, k)) = \frac{1}{n} \sum_{k|n} f(k)\phi(n/k).$$

5.2 Partitions and subsets

5.2.1 Partition function

Number of ways of writing n as a sum of positive integers, disregarding the order of the summands.

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

n	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	~2e5	~2e8

5.3 General purpose numbers

5.3.1 Bernoulli numbers

EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t - 1}$ (FFT-able). $B[0, \ldots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \ldots]$

Sums of powers:

$$\sum_{i=1}^{n} n^m = \frac{1}{m+1} \sum_{k=0}^{m} \binom{m+1}{k} B_k \cdot (n+1)^{m+1-k}$$

5.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

$$s(n, k) = s(n-1, k-1) + (n-1)s(n-1, k), \ s(0, 0) = 1$$

$$\textstyle\sum_{k=0}^{n} s(n, k)x^k = x(x+1)\ldots(x+n-1)$$

$$\sum_{n \geqslant 0} \sum_{k=0}^{n} s(n, k) u^k \frac{z^n}{n!} = (1+z)^u$$

5.3.3 Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j:s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j:s s.t. $\pi(j) \geq j$, k j:s s.t. $\pi(j) > j$.

$$E(n, k) = (n-k)E(n-1, k-1) + (k+1)E(n-1, k)$$

$$E(n, 0) = E(n, n-1) = 1$$

$$E(n, k) = \sum_{j=0}^{k} (-1)^j \binom{n+1}{j} (k+1-j)^n$$

$$\sum_{n, m \geqslant 0} E(n, k) w^m \frac{z^n}{n!} = \frac{1-w}{e^{(w-1)z} - w}$$

$$\frac{E_n}{(1-x)^{n+1}} = \frac{d}{dx}\left(\frac{E_{n-1}}{(1-x)^n}\right)$$

$$\sum_{k=n-m}^{n} E(n, k) \binom{k}{n-m} = m! S(n, m)$$

5.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

$$S(n, k) = S(n-1, k-1) + kS(n-1, k)$$

$$S(n, 1) = S(n, n) = 1$$

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n$$

$$\sum_{k \geqslant 0} \sum_{n \geqslant k} S(n, k) \frac{x^n}{n!} y^k = e^{y(e^x - 1)}$$

5.3.5 Bell numbers

Total number of partitions of n distinct elements. $B(x) = e^{e^x - 1}$

5.4 Euler's Pentagonal Theorem

$$\prod_{n=1}^{\infty}(1-x^n) = \sum_{n=-\infty}^{\infty}(-1)^n x^{\frac{n(3n-1)}{2}}$$

Graph (6)

6.1 Flows

Dinic.h

Description: Flow algorithm with complexity $O(VE\log U)$ where $U = \max|\text{cap}|$. $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite matching.

0eb3ef, 83 lines

```
// Warning: int flow!
const int INF = 1e9;

struct Edge {
  int from, to;
  int flow;
  int cap;
  Edge(int from, int to, int flow, int cap): from(from), to(to)
      , flow(flow), cap(cap) {}
};

struct DinicFlow {
  vector<Edge> edges;
  vector<vector<int>> G;
  vector<int> last_edge;
  vector<int> dist;
  int N;
  int s, t;

  DinicFlow(int N, int s, int t): edges(), G(N), last_edge(N,
      0), dist(N, 0), N(N), s(s), t(t) {}

  void add_undir_edge(int from, int to, int cap) {
    G[from].emplace_back(edges.size());
    edges.emplace_back(from, to, 0, cap);
    G[to].emplace_back(edges.size());
    edges.emplace_back(to, from, 0, cap);
  }

  void add_dir_edge(int from, int to, int cap) {
    G[from].emplace_back(edges.size());
    edges.emplace_back(from, to, 0, cap);
    G[to].emplace_back(edges.size());
    edges.emplace_back(to, from, 0, 0);
  }

  int dfs(int v, int mf) {
    if (v == t)
      return mf;
    int res = 0;
    for (; last_edge[v] < (int)G[v].size(); ++last_edge[v]) {
      int i = G[v][last_edge[v]];
      if (dist[edges[i].to] != dist[v] + 1)
        continue;
      if (edges[i].cap <= edges[i].flow)
        continue;
      int cur = dfs(edges[i].to, min(mf - res, edges[i].cap -
          edges[i].flow));
      if (cur == 0)
        continue;
      res += cur;
      edges[i].flow += cur;
      edges[i ^ 1].flow -= cur;
      if (edges[i].cap > edges[i].flow || res == mf)
        return res;
    }
    return res;
  }

  int flow() {
    while (true) {
      dist.assign(N, N);
      deque<int> Q;
      dist[s] = 0;
      Q.emplace_back(s);
      while (!Q.empty()) {
        int v = Q.front();
        Q.pop_front();
        for (int i : G[v]) {
          if (edges[i].cap > edges[i].flow && dist[edges[i].to]
              > dist[v] + 1) {
            dist[edges[i].to] = dist[v] + 1;
            Q.emplace_back(edges[i].to);
          }
        }
      }
      if (dist[t] == N)
        break;
      fill(last_edge.begin(), last_edge.end(), 0);
      while (dfs(s, INF));
    }
    int res = 0;
    for (int i : G[s])
      res += edges[i].flow;
    return res;
  }
};
```

MinCostMaxFlow.h

Description: Min-cost max-flow.

7952c9, 93 lines

```
// Warning: inf flow and cost!
// Each step works in O(E log V), sometimes O(V^2) may be
    better
// Dijkstra can be replaced
const int INF = 1e9;

struct CostEdge {
  int from, to;
  int flow;
  int cap;
  int cost;
  CostEdge(int from, int to, int flow, int cap, int cost): from
      (from), to(to), flow(flow), cap(cap), cost(cost) {}
};

struct MinCost {
  vector<CostEdge> edges;
  vector<vector<int>> G;
  vector<int> dist;
  vector<int> potential;
  vector<int> par;
  int N;
  int s, t;

  MinCost(int N, int s, int t): edges(), G(N), dist(N),
      potential(N), par(N), N(N), s(s), t(t) {}

  void add_dir_edge(int from, int to, int cap, int cost) {
    G[from].emplace_back(edges.size());
    edges.emplace_back(from, to, 0, cap, cost);
    G[to].emplace_back(edges.size());
    edges.emplace_back(to, from, 0, 0, -cost);
  }

  void calc_potential() {
    for (int i = 0; i < N; ++i) {
      for (auto e : edges) {
        if (e.flow < e.cap)
          potential[e.to] = min(potential[e.to], potential[e.
              from] + e.cost);
      }
    }
  }

  int flow;
  int cost;

  bool step() {
    fill(dist.begin(), dist.end(), INF);
    fill(par.begin(), par.end(), -1);
    set<pair<int, int>> Q;
    dist[s] = 0;
    Q.emplace(0, s);
    while (!Q.empty()) {
      int v = Q.begin()->second;
      Q.erase(Q.begin());
      for (int i : G[v]) {
        if (edges[i].cap > edges[i].flow) {
          int u = edges[i].to;
          int opt = dist[v] + edges[i].cost + potential[v] -
              potential[u];
          if (dist[u] > opt) {
            Q.erase(make_pair(dist[u], u));
            par[u] = i;
            dist[u] = opt;
            Q.emplace(dist[u], u);
          }
        }
      }
    }
    if (dist[t] == INF)
      return false;
    int mn = INF;
    int cur = t;
    while (cur != s) {
      mn = min(mn, edges[par[cur]].cap - edges[par[cur]].flow);
      cur = edges[par[cur]].from;
    }
    cur = t;
    while (cur != s) {
      edges[par[cur]].flow += mn;
      edges[par[cur] ^ 1].flow -= mn;
      cur = edges[par[cur]].from;
    }
    cost += mn * (dist[t] + potential[t] - potential[s]);
    for (int i = 0; i < N; ++i)
      potential[i] += dist[i];
    return true;
  }

  pair<int, int> min_cost_max_flow() {
    flow = 0;
    cost = 0;
    calc_potential();
    while (step());
    return make_pair(cost, flow);
  }
};
```

GlobalMinCut.h

Description: Find a global minimum cut in an undirected weighted graph, as represented by an adjacency matrix.
Time: $\mathcal{O}(V^3)$ e6817e, 46 lines

```
// G[i][j] = weight of the edge from i to j
// symmetric matrix
pair<int, vector<int>> global_min_cut(vector<vector<int>> G) {
  int n = (int) G.size();
  vector<vector<int>> comps(n);
  for (int i = 0; i < n; ++i)
    comps[i].resize(1, i);
  pair<int, vector<int>> ans(1e9, {});
  for (int i = 0; i < n - 1; ++i) {
    vector<int> w = G[0];
    vector<bool> taken(n, false);
    taken[0] = true;
    int s = 0, t = 0, last_cost = 0;
    for (int j = 0; j < n - i - 1; ++j) {
      int opt = -1;
      for (int u = 0; u < n; ++u) {
        if (!taken[u] && !comps[u].empty()) {
          if (opt == -1 || w[u] > w[opt])
            opt = u;
        }
      }
      s = t;
      t = opt;
      last_cost = w[opt];
      taken[opt] = true;
      for (int u = 0; u < n; ++u) {
        if (!taken[u])
          w[u] += G[opt][u];
      }
    }
    if (last_cost < ans.first) {
      ans.first = last_cost;
      ans.second = comps[t];
    }
    // merge s and t
    for (int u : comps[t]) {
      comps[s].emplace_back(u);
    }
    comps[t].clear();
    for (int i = 0; i < n; ++i) {
      G[s][i] += G[t][i];
      G[i][s] = G[s][i];
    }
  }
  return ans;
}
```

6.2 Matching

WeightedMatching.h

Description: Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires $N \leq M$.
Time: $\mathcal{O}(N^2M)$ 1e0fe9, 31 lines

```
pair<int, vi> hungarian(const vector<vi> &a) {
  if (a.empty()) return {0, {}};
  int n = sz(a) + 1, m = sz(a[0]) + 1;
  vi u(n), v(m), p(m), ans(n - 1);
  rep(i,1,n) {
    p[0] = i;
    int j0 = 0; // add "dummy" worker 0
    vi dist(m, INT_MAX), pre(m, -1);
    vector<bool> done(m + 1);
    do { // dijkstra
      done[j0] = true;
      int i0 = p[j0], j1, delta = INT_MAX;
      rep(j,1,m) if (!done[j]) {
        auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
        if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
        if (dist[j] < delta) delta = dist[j], j1 = j;
      }
      rep(j,0,m) {
        if (done[j]) u[p[j]] += delta, v[j] -= delta;
        else dist[j] -= delta;
      }
      j0 = j1;
    } while (p[j0]);
    while (j0) { // update alternating path
      int j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  rep(j,1,m) if (p[j]) ans[p[j] - 1] = j - 1;
  return {-v[0], ans}; // min cost
}
```

6.3 Dominator Tree

DominatorTree.h

Description: Dominator tree 3a84ae, 72 lines

```
struct Dom {
    static const int MAXN = MAX;
    vector<int> g[MAXN], e[MAXN], ch[MAXN];
    int tin[MAXN], ind[MAXN];
    int t = 0;
    int n;
    int s;

    Dom(int n, int s): n(n), s(s) {}

    void add(int u, int v) {
        g[u].push_back(v);
    }

    void calc_tin(int v) {
        tin[v] = t++;
        ind[tin[v]] = v;
        for (int u : g[v]) {
            if (tin[u] == -1) {
                calc_tin(u);
                ch[tin[v]].push_back(tin[u]);
            }
            e[tin[u]].push_back(tin[v]);
        }
    }

    vector<int> inv_sdom[MAXN];
    int dom[MAXN], sdom[MAXN], p[MAXN], val[MAXN];

    int get_min(int u, int v) {
        return sdom[u] < sdom[v] ? u : v;
    }

    int get(int v) {
        if (v == p[v]) return val[v];
        int res = get(p[v]);
        p[v] = p[p[v]];
        return val[v] = get_min(val[v], res);
    }

    void solve() {
        fill(tin, tin + n, -1);
        fill(ind, ind + n, -1);
        iota(p, p + n, 0);
        iota(val, val + n, 0);
        iota(sdom, sdom + n, 0);
        calc_tin(s);
        for (int v = n - 1; v >= 0; --v) {
            for (int u : e[v])
                sdom[v] = min(sdom[v], sdom[get(u)]);
            inv_sdom[sdom[v]].push_back(v);
            for (int u : inv_sdom[v]) {
                int res = get(u);
                if (sdom[res] == v) {
                    dom[u] = v;
                } else {
                    dom[u] = res;
                }
            }
            for (int u : ch[v]) p[u] = v;
        }
        vector<int> rdom(n);
        for (int v = 0; v < n; ++v) {
            if (dom[v] != sdom[v])
                dom[v] = dom[dom[v]];
            if (ind[v] != -1)
                rdom[ind[v]] = ind[dom[v]];
        }
        for (int v = 0; v < n; ++v)
            dom[v] = rdom[v];
    }
};
```

6.4 Trees

LinkCutTree.h

Description: Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree.
Time: All operations take amortized $\mathcal{O}(\log N)$. 0fb462, 90 lines

```
struct Node { // Splay tree. Root's pp contains tree's parent.
  Node *p = 0, *pp = 0, *c[2];
  bool flip = 0;
  Node() { c[0] = c[1] = 0; fix(); }
  void fix() {
    if (c[0]) c[0]->p = this;
    if (c[1]) c[1]->p = this;
    // (+ update sum of subtree elements etc. if wanted)
  }
  void pushFlip() {
    if (!flip) return;
    flip = 0; swap(c[0], c[1]);
    if (c[0]) c[0]->flip ^= 1;
    if (c[1]) c[1]->flip ^= 1;
  }
  int up() { return p ? p->c[1] == this : -1; }
  void rot(int i, int b) {
    int h = i ^ b;
    Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y : x;
    if ((y->p = p)) p->c[up()] = y;
    c[i] = z->c[i ^ 1];
    if (b < 2) {
      x->c[h] = y->c[h ^ 1];
      y->c[h ^ 1] = x;
    }
    z->c[i ^ 1] = this;
    fix(); x->fix(); y->fix();
    if (p) p->fix();
    swap(pp, y->pp);
  }
  void splay() {
    for (pushFlip(); p; ) {
      if (p->p) p->p->pushFlip();
      p->pushFlip(); pushFlip();
      int c1 = up(), c2 = p->up();
      if (c2 == -1) p->rot(c1, 2);
      else p->p->rot(c2, c1 != c2);
    }
  }
  Node* first() {
    pushFlip();
    return c[0] ? c[0]->first() : (splay(), this);
  }
};

struct LinkCut {
  vector<Node> node;
  LinkCut(int N) : node(N) {}

  void link(int u, int v) { // add an edge (u, v)
    assert(!connected(u, v));
    makeRoot(&node[u]);
    node[u].pp = &node[v];
  }
  void cut(int u, int v) { // remove an edge (u, v)
    Node *x = &node[u], *top = &node[v];
    makeRoot(top); x->splay();
    assert(top == (x->pp ?: x->c[0]));
    if (x->pp) x->pp = 0;
    else {
      x->c[0] = top->p = 0;
      x->fix();
    }
  }
  bool connected(int u, int v) { // are u, v in the same tree?
    Node* nu = access(&node[u])->first();
    return nu == access(&node[v])->first();
  }
  void makeRoot(Node* u) {
    access(u);
    u->splay();
    if(u->c[0]) {
      u->c[0]->p = 0;
      u->c[0]->flip ^= 1;
      u->c[0]->pp = u;
      u->c[0] = 0;
      u->fix();
    }
  }
  Node* access(Node* u) {
    u->splay();
    while (Node* pp = u->pp) {
      pp->splay(); u->pp = 0;
      if (pp->c[1]) {
        pp->c[1]->p = 0; pp->c[1]->pp = pp; }
      pp->c[1] = u; pp->fix(); u = pp;
    }
    return u;
  }
};
```

DirectedMST.h

Description: Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.
Time: $\mathcal{O}(E \log V)$

"../data-structures/UnionFindRollback.h" 39e620, 60 lines

```
struct Edge { int a, b; ll w; };
struct Node {
  Edge key;
  Node *l, *r;
  ll delta;
  void prop() {
    key.w += delta;
    if (l) l->delta += delta;
    if (r) r->delta += delta;
    delta = 0;
  }
  Edge top() { prop(); return key; }
};
Node *merge(Node *a, Node *b) {
  if (!a || !b) return a ?: b;
  a->prop(), b->prop();
  if (a->key.w > b->key.w) swap(a, b);
  swap(a->l, (a->r = merge(b, a->r)));
  return a;
}
void pop(Node*& a) { a->prop(); a = merge(a->l, a->r); }

pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {
  RollbackUF uf(n);
  vector<Node*> heap(n);
  for (Edge e : g) heap[e.b] = merge(heap[e.b], new Node{e});
  ll res = 0;
  vi seen(n, -1), path(n), par(n);
  seen[r] = r;
  vector<Edge> Q(n), in(n, {-1,-1}), comp;
  deque<tuple<int, int, vector<Edge>>> cycs;
  rep(s,0,n) {
    int u = s, qi = 0, w;
    while (seen[u] < 0) {
      if (!heap[u]) return {-1,{}};
      Edge e = heap[u]->top();
      heap[u]->delta -= e.w, pop(heap[u]);
      Q[qi] = e, path[qi++] = u, seen[u] = s;
      res += e.w, u = uf.find(e.a);
      if (seen[u] == s) {
        Node* cyc = 0;
        int end = qi, time = uf.time();
        do cyc = merge(cyc, heap[w = path[--qi]]);
        while (uf.join(u, w));
        u = uf.find(u), heap[u] = cyc, seen[u] = -1;
        cycs.push_front({u, time, {&Q[qi], &Q[end]}});
      }
    }
    rep(i,0,qi) in[uf.find(Q[i].b)] = Q[i];
  }

  for (auto& [u,t,comp] : cycs) { // restore sol (optional)
    uf.rollback(t);
    Edge inEdge = in[u];
    for (auto& e : comp) in[uf.find(e.b)] = e;
    in[uf.find(inEdge.b)] = inEdge;
  }
  rep(i,0,n) par[i] = in[i].a;
  return {res, par};
}
```

6.5 Shortest Paths

KthShortestPath.h

Description: Finding K shortest paths in a directed weighted graph c6de09, 140 lines

```
//Leftist Heap

struct Node{
  long long val;
  int val_v;
  int left;
  int right;
  int s;
  Node(long long _val = 0, int _val_v = 0) {
    val = _val;
    val_v = _val_v;
    left = right = -1;
    s = 1;
  }
};

// don't know why
Node nodes[5000000];
int nodes_cnt = 0;

int node(long long val, int val_v) {
  nodes[nodes_cnt] = Node(val, val_v);
  ++nodes_cnt;
  return nodes_cnt - 1;
}

int get_s(int v) {
  if (v == -1)
    return 0;
  return nodes[v].s;
}

int merge(int a, int b) {
  if (a == -1)
    return b;
  if (b == -1)
    return a;
  if (nodes[a].val > nodes[b].val)
    swap(a, b);
  int ans = node(nodes[a].val, nodes[a].val_v);
  nodes[ans].left = nodes[a].left;
  nodes[ans].right = merge(nodes[a].right, b);
  if (get_s(nodes[ans].right) > get_s(nodes[ans].left))
    swap(nodes[ans].left, nodes[ans].right);
  nodes[ans].s = get_s(nodes[ans].right) + 1;
  return ans;
}

int extract_min(int v) {
  return merge(nodes[v].left, nodes[v].right);
}

// max number of vertices
const int MAXN = 500000;
const long long INF = 1e18;
vector<pair<pair<int, int>, int>> G[MAXN];
vector<pair<pair<int, int>, int>> GT[MAXN];
long long d[MAXN];
int par[MAXN];
int par_edge[MAXN];

int main() {
#ifdef DEBUG
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);
#else
  ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
#endif

  int n, m, k;
  cin >> n >> m >> k;
  for (int i = 0; i < m; ++i) {
    int u, v, w;
    cin >> u >> v >> w;
    G[u].emplace_back(make_pair(v, i), w);
    GT[v].emplace_back(make_pair(u, i), w);
  }
  fill(d, d + n, INF);
  d[n - 1] = 0;
  set<pair<long long, int>> Q;
  vector<int> order;
  par_edge[n - 1] = -1;
  Q.emplace(0, n - 1);
  while (!Q.empty()) {
    int v = Q.begin()->second;
    order.emplace_back(v);
    Q.erase(Q.begin());
    for (auto t : GT[v]) {
      if (d[t.first.first] > d[v] + t.second) {
        Q.erase(make_pair(d[t.first.first], t.first.first));
        d[t.first.first] = d[v] + t.second;
        par_edge[t.first.first] = t.first.second;
        par[t.first.first] = v;
        Q.emplace(d[t.first.first], t.first.first);
      }
    }
  }
  vector<int> heaps(n, -1);
  for (int v : order) {
    if (v != n - 1)
      heaps[v] = heaps[par[v]];
    for (auto t : G[v]) {
      if (t.first.second == par_edge[v])
        continue;
      if (d[t.first.first] != INF)
        heaps[v] = merge(heaps[v], node(d[t.first.first] + t.
             second - d[v], t.first.first));
    }
  }
  if (d[0] == INF) {
    for (int i = 0; i < k; ++i)
      cout << -1 << ' ';
    cout << '\n';
    return 0;
  }
  vector<long long> ans(k, -1);
  vector<int> rest_heaps(k, -1);
  ans[0] = d[0];
  rest_heaps[0] = heaps[0];
  if (rest_heaps[0] != -1)
    Q.emplace(d[0] + nodes[rest_heaps[0]].val, 0);
  for (int i = 1; i < k; ++i) {
    if (Q.empty())
      break;
    int j = Q.begin()->second;
    Q.erase(Q.begin());
    ans[i] = ans[j] + nodes[rest_heaps[j]].val;
    rest_heaps[i] = heaps[nodes[rest_heaps[j]].val_v];
    rest_heaps[j] = extract_min(rest_heaps[j]);
    if (rest_heaps[i] != -1)
      Q.emplace(ans[i] + nodes[rest_heaps[i]].val, i);
    if (rest_heaps[j] != -1)
      Q.emplace(ans[j] + nodes[rest_heaps[j]].val, j);
  }
  for (int i = 0; i < k; ++i)
    cout << ans[i] << ' ';
  cout << '\n';
  return 0;
}
```

6.6 Tarjan's SCC

SCC.h

Description: Finds strongly connected components in a directed graph.
Time: $\mathcal{O}(E + V)$ 9262d3, 54 lines

```
struct SCC {
  int n;
  vector<int>* g;
  vector<int> tin, used, up;
  int tim = 0;

  SCC(int n): n(n) {
    g = new vector<int>[n];
    tin.resize(n, -1);
    used.resize(n);
    up.resize(n);
  }

  ~SCC() {
    delete[] g;
  }

  void add_edge(int u, int v) {
    g[u].push_back(v);
  }

  vector<int> stk;

  void dfs(int v) {
    tin[v] = tim++;
    up[v] = tin[v];
    stk.push_back(v);
    used[v] = true;
    for (int u : g[v]) {
      if (tin[u] == -1) {
        dfs(u);
        up[v] = min(up[v], up[u]);
      } else if (used[u]) {
        up[v] = min(up[v], tin[u]);
      }
    }
    if (tin[v] == up[v]) {
      while (stk.back() != v) {
        used[stk.back()] = false;
        stk.pop_back();
      }
      used[v] = false;
      stk.pop_back();
    }
  }

  void get() {
    for (int i = 0; i < n; ++i) {
      if (tin[i] == -1) {
        dfs(i);
      }
    }
  }
};
```

6.7 Math

6.7.1 Number of Spanning Trees

Create an $N \times N$ matrix `mat`, and for each edge $a \to b \in G$, do `mat[a][b]--`, `mat[b][b]++` (and `mat[b][a]--`, `mat[a][a]++` if G is undirected). Remove the ith row and column and take the determinant; this yields the number of directed spanning trees rooted at i (if G is undirected, remove any row/column).

Geometry (7)

7.1 Half planes Intersection

HalfplaneIntersection.h

Description: Calculates the intersection of half-planes in a bounding box Works in O(N log N) 62b734, 195 lines

```
struct Point {
    long double x, y;
    Point(long double x, long double y): x(x), y(y) {}
    Point(): x(0), y(0) {}
};

Point operator+(const Point& first, const Point& second) {
    return Point(first.x + second.x, first.y + second.y);
}

Point operator-(const Point& first, const Point& second) {
    return Point(first.x - second.x, first.y - second.y);
}

long double cross(const Point& first, const Point& second) {
    return first.x * second.y - first.y * second.x;
}

long double sqrlen(const Point& P) {
    return P.x * P.x + P.y * P.y;
}

long double len(const Point& P) {
    return sqrtl(sqrlen(P));
}

const long double EPS = 1e-9;

bool is_zero(long double x) {
    return abs(x) < EPS;
}

bool is_less(long double a, long double b) {
    return (a - b) < -EPS;
}

bool is_more(long double a, long double b) {
    return (a - b) > EPS;
}

bool is_leq(long double a, long double b) {
    return (a - b) < EPS;
}

struct Line {
    long double a;
    long double b;
    long double c;
    Line(long double a, long double b, long double c): a(a), b(
         b), c(c) {}
    Point get_perp() const {
        return Point(a, b);
    }
    long double get_y_by_x(long double x) const {
        return (-c - a * x) / b;
    }
};

// for non collinear !
long double inter_x(const Line& l1, const Line& l2) {
    return (-l1.c * l2.b + l2.c * l1.b) / (l1.a * l2.b - l2.a *
         l1.b);
}

vector<Point> convex_hull(vector<Point> points) {
    if (points.empty())
        return {};
    for (size_t i = 1; i < points.size(); ++i) {
        if (is_less(points[i].y, points[0].y) || (is_zero(
             points[i].y - points[0].y) && points[i].x < points
             [0].x))
            swap(points[0], points[i]);
    }
    sort(points.begin() + 1, points.end(), [&](const Point& a,
         const Point& b) {
        if (is_zero(cross(a - points[0], b - points[0])))
            return sqrlen(a - points[0]) < sqrlen(b - points
                 [0]);
        return cross(a - points[0], b - points[0]) > 0;
    });
    vector<Point> stack;
    for (Point P : points) {
        while (stack.size() >= 2 && is_leq(cross(stack.back() -
             stack[stack.size() - 2], P - stack.back()), 0))
            stack.pop_back();
        stack.emplace_back(P);
    }
    return stack;
}

const long double INF = 1e9;

vector<Point> halfplane_inter_points(vector<Line> lines) {
    // ax + by + c >= 0
    long double min_x = -INF;
    long double max_x = INF;
    vector<Line> up; // facing up
    vector<Line> down; // facing down
    for (Line l : lines) {
        if (is_zero(l.b)) {
            if (l.a > 0)
                min_x = max(min_x, -l.c / l.a);
            else
                max_x = min(max_x, -l.c / l.a);
        } else if (l.b > 0) {
            up.emplace_back(l);
        } else {
            down.emplace_back(l);
        }
    }
    if (is_leq(max_x, min_x)) {
        return {};
    }
    auto make_hull = [](vector<Line>& lines, vector<Line>& hull
         , vector<long double>& xs, bool up) {
        sort(lines.begin(), lines.end(), [up](const Line& l1,
             const Line& l2) {
            if (is_zero(cross(l1.get_perp(), l2.get_perp()))) {
                // parallel
                // with sqrtl:
                //   return l1.c / len(l1.get_perp()) < l2.c /
                     len(l2.get_perp());
                // without, faster:
                if (is_leq(l1.c, 0)) {
                    if (is_more(l2.c, 0))
                        return true;
                    return is_more(l1.c * l1.c * sqrlen(l2.
                         get_perp()), l2.c * l2.c * sqrlen(l1.
                         get_perp()));
                } else {
                    if (is_leq(l2.c, 0))
                        return false;
                    return is_less(l1.c * l1.c * sqrlen(l2.
                         get_perp()), l2.c * l2.c * sqrlen(l1.
                         get_perp()));
                }
            }
            if (up) {
                return cross(l1.get_perp(), l2.get_perp()) > 0;
            }
            return cross(l1.get_perp(), l2.get_perp()) < 0;
        });
        for (Line l : lines) {
            // skip parallel
            if (!hull.empty() && is_zero(cross(hull.back().
                 get_perp(), l.get_perp())))
                continue;
            if (hull.empty()) {
                hull.emplace_back(l);
                continue;
            }
            while (!xs.empty() && is_leq(inter_x(hull.back(), l
                 ), xs.back())) {
                hull.pop_back();
                xs.pop_back();
            }
            xs.emplace_back(inter_x(hull.back(), l));
            hull.emplace_back(l);
        }
    };
    vector<Line> up_hull;
    vector<long double> up_xs;
    make_hull(up, up_hull, up_xs, true);
    vector<Line> down_hull;
    vector<long double> down_xs;
    make_hull(down, down_hull, down_xs, false);
    size_t i1 = 0;
    size_t i2 = 0;
    while (i1 < up_xs.size() && is_less(up_xs[i1], min_x))
        ++i1;
    while (i2 < down_xs.size() && is_less(down_xs[i2], min_x))
        ++i2;
    long double prev = min_x;
    vector<Point> points;
    auto try_pushing_points = [&](long double x, size_t i1,
         size_t i2) {
        long double y_up = up_hull[i1].get_y_by_x(x);
        long double y_down = down_hull[i2].get_y_by_x(x);
        if (is_less(y_up, y_down)) {
            points.emplace_back(x, y_up);
            points.emplace_back(x, y_down);
        }
        if (!is_zero(cross(up_hull[i1].get_perp(), down_hull[i2
             ].get_perp()))) {
            long double cur_x = inter_x(up_hull[i1], down_hull[
                 i2]);
            if (is_leq(prev, cur_x) && is_leq(cur_x, x)) {
                points.emplace_back(cur_x, up_hull[i1].
                     get_y_by_x(cur_x));
            }
        }
        prev = x;
    };
    try_pushing_points(min_x, i1, i2);
    while (i1 < up_xs.size() || i2 < down_xs.size()) {
        long double cur_x = 0;
        int old_i1 = i1;
        int old_i2 = i2;
        if (i1 < up_xs.size() && (i2 == down_xs.size() || up_xs
             [i1] < down_xs[i2])) {
            cur_x = up_xs[i1];
            ++i1;
        } else {
            cur_x = down_xs[i2];
            ++i2;
        }
        if (cur_x > max_x)
            cur_x = max_x;
        try_pushing_points(cur_x, old_i1, old_i2);
        if (cur_x == max_x)
            break;
    }
    if (prev < max_x)
        try_pushing_points(max_x, i1, i2);
    return convex_hull(points);
}
```

7.2 Misc. Point Set Problems

kdTree.h

Description: KD-tree (2d, can be extended to 3d)

"Point.h" bac5b0, 63 lines

```
typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();

bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }

struct Node {
  P pt; // if this is a leaf, the single point in it
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
  Node *first = 0, *second = 0;

  T distance(const P& p) { // min squared distance to a point
    T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
    T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
    return (P(x,y) - p).dist2();
  }

  Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
      x0 = min(x0, p.x); x1 = max(x1, p.x);
      y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
      // split on x if width >= height (not ideal...)
      sort(all(vp), x1 - x0 >= y1 - y0 ? on_x : on_y);
      // divide by taking half the array for each child (not
      // best performance with many duplicates in the middle)
      int half = sz(vp)/2;
      first = new Node({vp.begin(), vp.begin() + half});
      second = new Node({vp.begin() + half, vp.end()});
    }
  }
};

struct KDTree {
  Node* root;
  KDTree(const vector<P>& vp) : root(new Node({all(vp)})) {}

  pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
      // uncomment if we should not find the point itself:
      // if (p == node->pt) return {INF, P()};
      return make_pair((p - node->pt).dist2(), node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->distance(p);
    if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);

    // search closest side first, other side if needed
    auto best = search(f, p);
    if (bsec < best.first)
      best = min(best, search(s, p));
    return best;
  }

  // find nearest point to a point, and its squared distance
  // (requires an arbitrary operator< for Point)
  pair<T, P> nearest(const P& p) {
    return search(root, p);
  }
};
```

FastDelaunay.h

Description: Fast Delaunay triangulation. Each circumcircle contains none of the input points. There must be no duplicate points. If all points are on a line, no triangles will be returned. Should work for doubles as well, though there may be precision issues in 'circ'. Returns triangles in order {t[0][0], t[0][1], t[0][2], t[1][0], . . . }, all counter-clockwise.
Time: $\mathcal{O}(n \log n)$

"Point.h" eefdf5, 88 lines

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point

struct Quad {
  Q rot, o; P p = arb; bool mark;
  P& F() { return r()->p; }
  Q& r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
} *H;

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}
Q makeEdge(P orig, P dest) {
  Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
  H = r->o; r->r()->r() = r;
  rep(i,0,4) r = r->rot, r->p = arb, r->o = i & 1 ? r : r->r();
  r->p = orig; r->F() = dest;
  return r;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

pair<Q,Q> rec(const vector<P>& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back());
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec({all(s) - half});
  tie(B, rb) = rec({sz(s) - half + all(s)});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e->o = H; H = e; e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}

vector<P> triangulate(vector<P> pts) {
  sort(all(pts));  assert(unique(all(pts)) == pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
  q.push_back(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

7.3 3D

PolyhedronVolume.h

Description: Magic formula for the volume of a polyhedron. Faces should point outwards.

3058c3, 6 lines

```
template<class V, class L>
double signedPolyVolume(const V& p, const L& trilist) {
  double v = 0;
  for (auto i : trilist) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}
```

Point3D.h

Description: Class to handle points in 3D space. T can be e.g. double or long long.

8058ae, 32 lines

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0, pi]
  double theta() const { return atan2(sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)dist(); } //makes dist()=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit();
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
  }
};
```

3dHull.h

Description: Computes all faces of the 3-dimension hull of a point set. *No four points must be coplanar*, or else random results will be returned. All faces will point outwards.
Time: $\mathcal{O}(n^2)$

"Point3D.h" 5b45fc, 49 lines

```
typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F { P3 q; int a, b, c; };

vector<F> hull3d(const vector<P3>& A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
    FS.push_back(f);
  };
  rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

  rep(i,4,sz(A)) {
    rep(j,0,sz(FS)) {
      F f = FS[j];
      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a,b).rem(f.c);
        E(a,c).rem(f.b);
        E(b,c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    rep(j,0,nw) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
  return FS;
};
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

611f07, 8 lines

```
double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

Strings (8)

PrefixFunction.h

Description: pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.
Time: $\mathcal{O}(n)$

067f0d, 11 lines

```
vector<int> get_pi(string s) {
    int n = (int)s.size();
    vector<int> pi(n);
    for (int i = 1; i < n; ++i) {
        int j = pi[i - 1];
        while (j != -1 && s[j] != s[i])
            j = (j == 0 ? -1 : pi[j - 1]);
        pi[i] = j + 1;
    }
    return pi;
}
```

ZFunction.h

Description: z[i] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)
Time: $\mathcal{O}(n)$

b51941, 14 lines

```
vector<int> get_z(string s) {
  int n = (int)s.size();
  int j = 0;
  vector<int> z(n);
  for (int i = 1; i < n; ++i) {
    if (j + z[j] > i)
      z[i] = min(z[i - j], j + z[j] - i);
    while (s[z[i]] == s[i + z[i]])
      ++z[i];
    if (i + z[i] > j + z[j])
      j = i;
  }
  return z;
}
```

Manacher.h

Description: For each position in a string, computes p[i] = half length of longest odd palindrome around pos i (half rounded down).
Time: $\mathcal{O}(N)$

1ea0a1, 17 lines

```
vector<int> get_odd_pali(string s) {
  int n = (int)s.size();
  vector<int> pali(n);
  int l = 0, r = 0;
  for (int i = 0; i < n; ++i) {
    if (i < r) {
      pali[i] = min(r - i - 1, pali[l + r - i - 1]);
    }
    while (i > pali[i] && i + pali[i] + 1 < n && s[i - pali[i]
        - 1] == s[i + pali[i] + 1])
      ++pali[i];
    if (i + pali[i] + 1 > r) {
      r = i + pali[i] + 1;
      l = i - pali[i];
    }
  }
  return pali;
}
```

SuffixArray.h

Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any zero bytes.
Time: $\mathcal{O}(n \log n)$

f9c581, 64 lines

```
// all symbols > $
struct SuffixArray {
  vector<int> order;
  vector<int> lcp;
  vector<int> id;
  SuffixArray(string s) {
    s += '$';
    int n = (int) s.size();
    order.resize(n);
    vector<int> classes(n);
    vector<int> new_order(n);
    vector<int> new_classes(n);
    vector<int> cnt(n);
    iota(order.begin(), order.end(), 0);
    sort(order.begin(), order.end(), [&](int a, int b) {
      return s[a] < s[b];
    });
    classes[order[0]] = 0;
    for (int i = 1; i < n; ++i)
      classes[order[i]] = classes[order[i - 1]] + (s[order[i]]
          != s[order[i - 1]]);
    auto safe = [&](int x) {
      if (x >= n) return x - n;
      return x;
    };
    for (int l = 1; l < n; l *= 2) {
      fill(cnt.begin(), cnt.end(), 0);
      for (int i = 0; i < n; ++i)
        ++cnt[classes[i]];
      for (int i = 1; i < n; ++i)
        cnt[i] += cnt[i - 1];
      for (int i = n - 1; i >= 0; --i) {
        int j = order[i] - l;
        if (j < 0)
          j += n;
        --cnt[classes[j]];
        new_order[cnt[classes[j]]] = j;
      }
      new_classes[new_order[0]] = 0;
      for (int i = 1; i < n; ++i) {
        new_classes[new_order[i]] = new_classes[new_order[i -
            1]] +
        (make_pair(classes[new_order[i]], classes[safe(
            new_order[i] + l)]) !=
        make_pair(classes[new_order[i - 1]], classes[safe(
            new_order[i - 1] + l)]));
      }
      swap(classes, new_classes);
      swap(order, new_order);
    }
    lcp.resize(n);
    id.resize(n);
    for (int i = 0; i < n; ++i)
      id[order[i]] = i;
    int tmp = 0;
    for (int i = 0; i < n; ++i) {
      if (id[i] == n - 1) {
        tmp = 0;
        continue;
      }
      int j = order[id[i] + 1];
      while (s[i + tmp] == s[j + tmp])
        ++tmp;
      lcp[id[i]] = tmp;
      tmp = max(tmp - 1, 0);
    }
  }
};
```

SuffixAuto.h

Description: Minimum automaton accepting all the suffixes of the line (and only them). Has linear size (up to 2n vertices, 3n edges)
Time: $\mathcal{O}(n)$

e31179, 55 lines

```
// maxlen * 2
const int MAXN = 300300;
const int ALPHA = 26;

int go[MAXN][ALPHA];
int par[MAXN];
int len[MAXN];
int suff[MAXN];
int nodes_cnt = 0;

int get_char_id(char c) {
  return c - 'a';
}
```

```
int node() {
  fill(go[nodes_cnt], go[nodes_cnt] + ALPHA, -1);
  suff[nodes_cnt] = -1;
  len[nodes_cnt] = 0;
  ++nodes_cnt;
  return nodes_cnt - 1;
}

int push_back(int last, char c) {
  int new_v = node();
  go[last][get_char_id(c)] = new_v;
  par[new_v] = last;
  len[new_v] = len[last] + 1;
  last = suff[last];
  while (last != -1 && go[last][get_char_id(c)] == -1) {
    go[last][get_char_id(c)] = new_v;
    last = suff[last];
  }
  if (last == -1) {
    suff[new_v] = 0;
  } else {
    int pos_suff = go[last][get_char_id(c)];
    if (len[pos_suff] == len[last] + 1) {
      suff[new_v] = pos_suff;
    } else {
      int new_suff = node();
      len[new_suff] = len[last] + 1;
      suff[new_suff] = suff[pos_suff];
      par[new_suff] = last;
      suff[pos_suff] = new_suff;
      for (int i = 0; i < ALPHA; ++i)
        go[new_suff][i] = go[pos_suff][i];
      while (last != -1 && go[last][get_char_id(c)] == pos_suff
          ) {
        go[last][get_char_id(c)] = new_suff;
        last = suff[last];
      }
      suff[new_v] = new_suff;
    }
  }
  return new_v;
}
```

PalindromeTree.h

Description: Two trees for all the palindrome substring of a line. One tree for odd palindromes, other for even. Has linear size (up to n+2 nodes)

Time: $\mathcal{O}(n)$

8984ee, 52 lines

```
// maxlen + eps
const int MAXSZ = 400'100;
const int ALPHA = 26;

int go[MAXSZ][ALPHA];
int len[MAXSZ];
int suff[MAXSZ];
int nodes_cnt = 0;

int node() {
  fill(go[nodes_cnt], go[nodes_cnt] + ALPHA, -1);
  len[nodes_cnt] = 0;
  suff[nodes_cnt] = -1;
  ++nodes_cnt;
  return nodes_cnt - 1;
}

int get_char_id(char c) {
  return c - 'a';
}

string cur_s;

int push_back(int last, char c) {
  cur_s += c;
  while ((int)cur_s.size() < len[last] + 2 || cur_s[cur_s.size
      () - len[last] - 2] != c)
    last = suff[last];
  if (go[last][get_char_id(c)] == -1) {
    int new_v = node();
    go[last][get_char_id(c)] = new_v;
    len[new_v] = len[last] + 2;
    if (len[last] == -1) {
      suff[new_v] = 1; // even root
    } else {
      int new_suff = suff[last];
      while (new_suff != -1 && cur_s[cur_s.size() - len[
          new_suff] - 2] != c)
        new_suff = suff[new_suff];
      suff[new_v] = go[new_suff][get_char_id(c)];
    }
  }
  last = go[last][get_char_id(c)];
  return last;
}

/* in main
  node();  0 - odd root
  node();  1 - even root
  suff[0] = -1;
  len[0] = -1;
  suff[1] = 0;
  len[1] = 0;
*/
```

Various (9)

9.1 Misc. algorithms

FastKnapsack.h

Description: Given N non-negative integer weights w and a non-negative target t, computes the maximum S <= t such that S is the sum of some subset of the weights.

Time: $\mathcal{O}(N \max(w_i))$

b20ccc, 16 lines

```
int knapsack(vi w, int t) {
  int a = 0, b = 0, x;
  while (b < sz(w) && a + w[b] <= t) a += w[b++];
  if (b == sz(w)) return a;
  int m = *max_element(all(w));
  vi u, v(2*m, -1);
  v[a+m-t] = b;
  rep(i,b,sz(w)) {
    u = v;
    rep(x,0,m) v[x+w[i]] = max(v[x+w[i]], u[x]);
    for (x = 2*m; --x > m;) rep(j, max(0,u[x]), v[x])
      v[x-w[j]] = max(v[x-w[j]], j);
  }
  for (a = t; v[a+m-t] < 0; a--) ;
  return a;
}
```

9.2 Dynamic programming

KnuthDP.h

Description: When doing DP on intervals: $a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i,j)$, where the (minimal) optimal k increases with both i and j, one can solve intervals in increasing order of length, and search $k = p[i][j]$ for $a[i][j]$ only between $p[i][j-1]$ and $p[i+1][j]$. This is known as Knuth DP. Sufficient criteria for this are if $f(b,c) \le f(a,d)$ and $f(a,c) + f(b,d) \le f(a,d) + f(b,c)$ for all $a \le b \le c \le d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.

Time: $\mathcal{O}(N^2)$

9.3 Debugging tricks

9.4 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

9.4.1 Bit hacks

- `x & -x` is the least bit in `x`.
- `c = x&-x, r = x+c; (((r^x) >> 2)/c) | r` is the next number after `x` with the same number of bits set.

9.4.2 Pragmas

- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

FastMod.h

Description: Compute $a\%b$ about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to $a \pmod{b}$ in the range $[0, 2b)$.

751a02, 8 lines

```
typedef unsigned long long ull;
struct FastMod {
  ull b, m;
  FastMod(ull b) : b(b), m(-1ULL / b) {}
  ull reduce(ull a) { // a % b + (0 or b)
    return a - (ull)((__uint128_t(m) * a) >> 64) * b;
  }
};
```

FastInput.h

Description: Read an integer from stdin. Usage requires your program to pipe in input from file.

Usage: ./a.out < input.txt

Time: About 5x as fast as cin/scanf.

7b3c70, 17 lines

```
inline char gc() { // like getchar()
  static char buf[1 << 16];
  static size_t bc, be;
  if (bc >= be) {
    buf[0] = 0, bc = 0;
    be = fread(buf, 1, sizeof(buf), stdin);
  }
  return buf[bc++]; // returns 0 on EOF
}

int readInt() {
  int a, c;
  while ((a = gc()) < 40);
  if (a == '-') return -readInt();
  while ((c = gc()) >= 48) a = a * 10 + c - 480;
  return a - 48;
}
```

BumpAllocator.h

Description: When you need to dynamically allocate many objects and don't care about freeing them. "new X" otherwise has an overhead of something like 0.05us + 16 bytes per allocation.

745db2, 8 lines

```
// Either globally or in a single class:
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*) {}
```