Universitat Politècnica de Catalunya

UPC2

Michael Sammler, Eric Valls, Dean Zhu

SWERC 2017

November 26, 2017

Contest (1)

template.cpp

18 lines

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

const ll oo = 0x3f3f3f3f3f3f3f3fLL;

#define FOR(i, a, b) for(ll i = (a); i < int(b); i++)
#define FORD(i, a, b) for(ll i = (b)-1; i >= int(a); i--)
#define has(c, e) ((c).find(e) != (c).end())

int main() {
  cin.sync_with_stdio(0); cin.tie(0);
  cin.exceptions(cin.failbit);
  return 0;
}
```

Makefile

1 lines

```
CXXFLAGS += -g -Wall -Wextra -Wshadow -std=c++11
```

.bashrc

3 lines

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++14 \
  -fsanitize=undefined,address'
xmodmap -e 'clear lock' -e 'keycode 66=less greater' #caps = <>
```

troubleshoot.txt

52 lines

```
Pre-submit:
Write a few simple test cases, if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all datastructures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a team mate.
Ask the team mate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a team mate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your team mates think about your algorithm?

Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all datastructures between test cases?
```

Data structures (2)

LineContainer.h

Description: Container where you can add lines of the form kx+m, and query maximum values at points x. Useful for dynamic programming.
Time: $\mathcal{O}(\log N)$

32 lines

```
bool Q;
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const {
    return Q ? p < o.p : k < o.k;
  }
};

struct LineContainer : multiset<Line> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) { x->p = inf; return false; }
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 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));
  }
  ll query(ll x) {
    assert(!empty());
    Q = 1; auto l = *lower_bound({0,0,x}); Q = 0;
    return l.k * x + l.m;
  }
};
```

FenwickTree2d.h

Description: Computes sums a[i,j] for all i<I, j<J, and increases single elements a[i,j]. Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).
Time: $\mathcal{O}(\log^2 N)$. (Use persistent segment trees for $\mathcal{O}(\log N)$.)

"FenwickTree.h" 22 lines

```
struct FT2 {
  vector<vi> ys; vector<FT> ft;
  FT2(int limx) : ys(limx) {}
  void fakeUpdate(int x, int y) {
    for (; x < sz(ys); x |= x + 1) ys[x].push_back(y);
  }
  void init() {
    trav(v, ys) sort(all(v)), ft.emplace_back(sz(v));
  }
  int ind(int x, int y) {
    return (int)(lower_bound(all(ys[x]), y) - ys[x].begin()); }
  void update(int x, int y, ll dif) {
    for (; x < sz(ys); x |= x + 1)
      ft[x].update(ind(x, y), dif);
  }
  ll query(int x, int y) {
    ll sum = 0;
    for (; x; x &= x - 1)
      sum += ft[x-1].query(ind(x-1, y));
    return sum;
  }
};
```

Numerical (3)

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \le b$, $x \ge 0$. 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.

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;
  }
};
```

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}\left(n^2 m\right)$

39 lines

```
typedef vector<double> vd;
const double eps = 1e-12;

int solveLinear(vector<vd>& A, vd& b, vd& x) {
  int n = A.size(), m = x.size(), rank = 0, br, bc;
  if (n) assert(A[0].size() == m);
  // FOR(i, 0, n) FOR(j, 0, m) A[i][j] %= MOD; also b[i]...
  vi col(m); iota(col.begin(), col.end(), 0);

  FOR(i,0,n) {
    double v, bv = 0;
    FOR(r,i,n) FOR(c,i,m)
      if ((v = fabs(A[r][c])) > bv)
        br = r, bc = c, bv = v;
    if (bv <= eps) {
      FOR(j,i,n) if (fabs(b[j]) > eps) return -1;
      break;
    }
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    FOR(j,0,n) swap(A[j][i], A[j][bc]);
    bv = 1/A[i][i];
    FOR(j,i+1,n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      FOR(k,i,m) A[j][k] -= fac*A[i][k];
    }
    rank++;
  }

  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    FOR(j,0,i) b[j] -= A[j][i] * b[i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

SolveLinear2.h

Description: To get all uniquely determined values of x back from SolveLinear, make the following changes:

"SolveLinear.h" 7 lines

```
rep(j,0,n) if (j != i) // instead of rep(j,i+1,n)
// ... then at the end:
x.assign(m, undefined);
rep(i,0,rank) {
  rep(j,rank,m) if (fabs(A[i][j]) > eps) goto fail;
  x[col[i]] = b[i] / A[i][i];
fail:; }
```

FFT.h

Description: Fast Fourier transform. Also includes a function for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. a and b should be of roughly equal size. For convolutions of integers, rounding the results of conv works if $(|a| + |b|)\max(a, b) < \sim 10^9$ (in theory maybe 10^6); you may want to use an NTT from the Number Theory chapter instead.
Time: $\mathcal{O}(N \log N)$

<valarray> 29 lines

```
typedef valarray<complex<double> > carray;
void fft(carray& x, carray& roots) {
  int N = sz(x);
  if (N <= 1) return;
  carray even = x[slice(0, N/2, 2)];
  carray odd = x[slice(1, N/2, 2)];
  carray rs = roots[slice(0, N/2, 2)];
  fft(even, rs);
  fft(odd, rs);
  rep(k,0,N/2) {
    auto t = roots[k] * odd[k];
    x[k    ] = even[k] + t;
    x[k+N/2] = even[k] - t;
  }
}

typedef vector<double> vd;
vd conv(const vd& a, const vd& b) {
  int s = sz(a) + sz(b) - 1, L = 32-__builtin_clz(s), n = 1<<L;
  if (s <= 0) return {};
  carray av(n), bv(n), roots(n);
  rep(i,0,n) roots[i] = polar(1.0, -2 * M_PI * i / n);
  copy(all(a), begin(av)); fft(av, roots);
  copy(all(b), begin(bv)); fft(bv, roots);
  roots = roots.apply(conj);
  carray cv = av * bv; fft(cv, roots);
  vd c(s); rep(i,0,s) c[i] = cv[i].real() / n;
  return c;
}
```

FFTIntegers.h

Description: NTT
Time: $\mathcal{O}(N \log N)$

39 lines

```
LL fpw(LL a, LL b, LL p){
    LL r = 1; while(b){if(b&1) r=r*a%p; a=a*a%p; b/=2;} return
        r;
}

const LL MOD = 2013265921; const LL ROOT = 440564289; // MOD ==
     15*(1<<27)+1 (prime)
vector<LL> e, er;                                    // ROOT
     has order 2^27
void FFT(vector<int> &x, LL d = 1){
  int n = x.size();
  if(n != e.size()){
    e.resize(n); er.resize(n);
    e[0] = 1; e[1] = fpw(ROOT,(1<<27)/n,MOD);
    er[0] = 1; er[1] = fpw(e[1],MOD-2,MOD);
    rep(i,2,n) e[i] = e[i-1] * e[1] % MOD;
    rep(i,2,n) er[i] = er[i-1] * er[1] % MOD;
  }
  if(d == -1) swap(e, er);
  rep(i,0,n){
    int j=0; for(int k=1; k<n; k<<=1, j<<=1) if(k&i) j++;   //
         haxy i cheetosy
    j>>=1; if(i<j) swap(x[i], x[j]);                        //
         haxy i cheetosy
  }
  int k=0;
  while((1<<k)<n) k++;
  for(int s=1; s<n; s<<=1){
    --k;
    for(int i=0; i<n; i+=2*s) rep(j,0,s){
      LL u = x[i+j], v = x[i+j+s]*e[j<<k]%MOD;
      x[i+j] = u+v-(u+v>=MOD?MOD:0);
      x[i+j+s] = u-v+(u-v<0?MOD:0);
    }
  }
  if(d == -1) swap(e, er);
}

vector<int> convolution(vector<int> a, vector<int> b){
  int n = 1; while(n < (int)max(a.size(), b.size())) n *= 2;
  n *= 2; a.resize(n); b.resize(n);
  FFT(a); FFT(b); rep(i,0,n) a[i] = (LL)a[i]*b[i]%MOD*fpw(n,MOD
      -2,MOD)%MOD; FFT(a, -1);
  return a;
}
```

Number theory (4)

4.1 Modular arithmetic

ModularArithmetic.h

Description: Operators for modular arithmetic. You need to set `mod` to some number first and then you can use the structure.

"euclid.h" 18 lines

```
const ll mod = 17; // change to something else
struct Mod {
  ll x;
  Mod(ll xx) : x(xx) {}
  Mod operator+(Mod b) { return Mod((x + b.x) % mod); }
  Mod operator-(Mod b) { return Mod((x - b.x + mod) % mod); }
  Mod operator*(Mod b) { return Mod((x * b.x) % mod); }
  Mod operator/(Mod b) { return *this * invert(b); }
  Mod invert(Mod a) {
    ll x, y, g = euclid(a.x, mod, x, y);
    assert(g == 1); return Mod((x + mod) % mod);
  }
```

```
  Mod operator^(ll e) {
    if (!e) return Mod(1);
    Mod r = *this ^ (e / 2); r = r * r;
    return e&1 ? *this * r : r;
  }
};
```

ModInverse.h

Description: Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime.

3 lines

```
const ll mod = 1000000007, LIM = 200000;
ll* inv = new ll[LIM] - 1; inv[1] = 1;
rep(i,2,LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod;
```

ModPow.h

6 lines

```
const ll mod = 1000000007; // faster if const
ll modpow(ll a, ll e) {
  if (e == 0) return 1;
  ll x = modpow(a * a % mod, e >> 1);
  return e & 1 ? x * a % mod : x;
}
```

ModSum.h

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

21 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) {
    ull to2 = (to * k + c) / m;
    res += to * to2;
    res -= divsum(to2, m-1 - c, m, k) + to2;
  }
  return res;
}

ll modsum(ull to, ll c, ll k, ll m) {
  c %= m;
  k %= m;
  if (c < 0) c += m;
  if (k < 0) k += 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 large c.
Time: $\mathcal{O}(64/bits \cdot \log b)$, where $bits = 64 - k$, if we want to deal with k-bit numbers.

19 lines

```
typedef unsigned long long ull;
const int bits = 10;
// if all numbers are less than 2^k, set bits = 64-k
const ull po = 1 << bits;
ull mod_mul(ull a, ull b, ull &c) {
  ull x = a * (b & (po - 1)) % c;
  while ((b >>= bits) > 0) {
    a = (a << bits) % c;
    x += (a * (b & (po - 1))) % c;
  }
  return x % c;
}
ull mod_pow(ull a, ull b, ull mod) {
  if (b == 0) return 1;
  ull res = mod_pow(a, b / 2, mod);
  res = mod_mul(res, res, mod);
  if (b & 1) return mod_mul(res, a, mod);
  return res;
}
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots.
Time: $\mathcal{O}(\log^2 p)$ worst case, often $\mathcal{O}(\log p)$

"ModPow.h" 30 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);
  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;
  int r = 0;
  while (s % 2 == 0)
    ++r, s /= 2;
  ll n = 2; // find a non-square mod p
  while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
  ll x = modpow(a, (s + 1) / 2, p);
  ll b = modpow(a, s, p);
  ll g = modpow(n, s, p);
  for (;;) {
    ll t = b;
    int m = 0;
    for (; m < r; ++m) {
      if (t == 1) break;
      t = t * t % p;
    }
    if (m == 0) return x;
    ll gs = modpow(g, 1 << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
    r = m;
  }
}
```

4.2 Number theoretic transform

NTT.h

Description: Number theoretic transform. Can be used for convolutions modulo specific nice primes of the form $2^a b+1$, where the convolution result has size at most 2^a. For other primes/integers, use two different primes and combine with CRT. May return negative values.
Time: $\mathcal{O}(N \log N)$

"ModPow.h" 38 lines

```
const ll mod = (119 << 23) + 1, root = 3; // = 998244353
// For p < 2^30 there is also e.g. (5 << 25, 3), (7 << 26, 3),
// (479 << 21, 3) and (483 << 21, 5). The last two are > 10^9.

typedef vector<ll> vl;
void ntt(ll* x, ll* temp, ll* roots, int N, int skip) {
  if (N == 1) return;
  int n2 = N/2;
  ntt(x     , temp, roots, n2, skip*2);
  ntt(x+skip, temp, roots, n2, skip*2);
  rep(i,0,N) temp[i] = x[i*skip];
  rep(i,0,n2) {
    ll s = temp[2*i], t = temp[2*i+1] * roots[skip*i];
    x[skip*i] = (s + t) % mod; x[skip*(i+n2)] = (s - t) % mod;
  }
}
void ntt(vl& x, bool inv = false) {
  ll e = modpow(root, (mod-1) / sz(x));
  if (inv) e = modpow(e, mod-2);
  vl roots(sz(x), 1), temp = roots;
  rep(i,1,sz(x)) roots[i] = roots[i-1] * e % mod;
  ntt(&x[0], &temp[0], &roots[0], sz(x), 1);
}
vl conv(vl a, vl b) {
  int s = sz(a) + sz(b) - 1; if (s <= 0) return {};
  int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
  if (s <= 200) { // (factor 10 optimization for |a|,|b| = 10)
    vl c(s);
    rep(i,0,sz(a)) rep(j,0,sz(b))
      c[i + j] = (c[i + j] + a[i] * b[j]) % mod;
    return c;
  }
  a.resize(n); ntt(a);
  b.resize(n); ntt(b);
  vl c(n); ll d = modpow(n, mod-2);
  rep(i,0,n) c[i] = a[i] * b[i] % mod * d % mod;
  ntt(c, true); c.resize(s); return c;
}
```

4.3 Primality

eratosthenes.h

Description: Prime sieve for generating all primes up to a certain limit. isprime[i] is true iff i is a prime.
Time: lim=100'000'000 $\approx$ 0.8 s. Runs 30% faster if only odd indices are stored.

11 lines

```
const int MAX_PR = 5000000;
bitset<MAX_PR> isprime;
vi eratosthenes_sieve(int lim) {
  isprime.set(); isprime[0] = isprime[1] = 0;
  for (int i = 4; i < lim; i += 2) isprime[i] = 0;
  for (int i = 3; i*i < lim; i += 2) if (isprime[i])
    for (int j = i*i; j < lim; j += i*2) isprime[j] = 0;
  vi pr;
  rep(i,2,lim) if (isprime[i]) pr.push_back(i);
  return pr;
}
```

MillerRabin.h

Description: Miller-Rabin primality probabilistic test. Probability of failing one iteration is at most 1/4. 15 iterations should be enough for 50-bit numbers.
Time: 15 times the complexity of $a^b \mod c$.

"ModMulLL.h" 16 lines

```
bool prime(ull p) {
  if (p == 2) return true;
  if (p == 1 || p % 2 == 0) return false;
  ull s = p - 1;
  while (s % 2 == 0) s /= 2;
  rep(i,0,15) {
    ull a = rand() % (p - 1) + 1, tmp = s;
    ull mod = mod_pow(a, tmp, p);
    while (tmp != p - 1 && mod != 1 && mod != p - 1) {
      mod = mod_mul(mod, mod, p);
      tmp *= 2;
    }
    if (mod != p - 1 && tmp % 2 == 0) return false;
  }
  return true;
}
```

factor.h
Description: Pollard's rho algorithm. It is a probabilistic factorisation algorithm, whose expected time complexity is good. Before you start using it, run `init(bits)`, where bits is the length of the numbers you use.
Time: Expected running time should be good enough for 50-bit numbers.
"MillerRabin.h", "eratosthenes.h", "euclid.h" 37 lines

```
vector<ull> pr;
ull f(ull a, ull n, ull &has) {
  return (mod_mul(a, a, n) + has) % n;
}
vector<ull> factor(ull d) {
  vector<ull> res;
  for (size_t i = 0; i < pr.size() && pr[i]*pr[i] <= d; i++)
    if (d % pr[i] == 0) {
      while (d % pr[i] == 0) d /= pr[i];
      res.push_back(pr[i]);
    }
  //d is now a product of at most 2 primes.
  if (d > 1) {
    if (prime(d))
      res.push_back(d);
    else while (true) {
      ull has = rand() % 2321 + 47;
      ull x = 2, y = 2, c = 1;
      for (; c==1; c = gcd((y > x ? y - x : x - y), d)) {
        x = f(x, d, has);
        y = f(f(y, d, has), d, has);
      }
      if (c != d) {
        res.push_back(c); d /= c;
        if (d != c) res.push_back(d);
        break;
      }
    }
  }
  return res;
}
void init(int bits) {//how many bits do we use?
  vi p = eratosthenes_sieve(1 << ((bits + 2) / 3));
  vector<ull> pr(p.size());
  for (size_t i=0; i<pr.size(); i++)
    pr[i] = p[i];
}
```

4.4 Divisibility

euclid.h
Description: Finds the Greatest Common Divisor to the integers a and b. Euclid also finds two integers x and y, such that $ax + by = \gcd(a, b)$. If a and b are coprime, then x is the inverse of $a \pmod{b}$.
7 lines

```
ll gcd(ll a, ll b) { return __gcd(a, b); }

ll euclid(ll a, ll b, ll &x, ll &y) {
  if (b) { ll d = euclid(b, a % b, y, x);
    return y -= a/b * x, d; }
  return x = 1, y = 0, a;
}
```

4.4.1 Bézout's identity

For $a \neq$, $b \neq 0$, then $d = gcd(a, b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x, y) is one solution, then all solutions are given by

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)}\right), \quad k \in \mathbb{Z}$$

phiFunction.h
Description: *Euler's totient* or *Euler's phi* function is defined as $\phi(n) :=$ # of positive integers $\leq n$ that are coprime with n. The *cototient* is $n - \phi(n)$. $\phi(1) = 1$, p prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, m, n coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) = (p_1 - 1)p_1^{k_1 - 1} ... (p_r - 1)p_r^{k_r - 1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$.
$\sum_{d|n} \phi(d) = n$, $\sum_{1 \leq k \leq n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$
Euler's thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod{n}$.
Fermat's little thm: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod{p}\ \forall a$.
10 lines

```
const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
  rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
  for(int i = 3; i < LIM; i += 2)
    if(phi[i] == i)
      for(int j = i; j < LIM; j += i)
        (phi[j] /= i) *= i-1;
}
```

4.5 Chinese remainder theorem

chinese.h
Description: Chinese Remainder Theorem.
`chinese(a, m, b, n)` returns a number x, such that $x \equiv a \pmod{m}$ and $x \equiv b \pmod{n}$. For not coprime n, m, use `chinese_common`. Note that all numbers must be less than 2^{31} if you have Z = unsigned long long.
Time: $\log(m + n)$
"euclid.h" 13 lines

```
template <class Z> Z chinese(Z a, Z m, Z b, Z n) {
  Z x, y; euclid(m, n, x, y);
  Z ret = a * (y + m) % m * n + b * (x + n) % n * m;
  if (ret >= m * n) ret -= m * n;
  return ret;
}

template <class Z> Z chinese_common(Z a, Z m, Z b, Z n) {
  Z d = gcd(m, n);
  if (((b -= a) %= n) < 0) b += n;
  if (b % d) return -1; // No solution
  return d * chinese(Z(0), m/d, b/d, n/d) + a;
}
```

4.6 Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \;\; b = k \cdot (2mn), \;\; c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

4.7 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.

Primitive roots exist modulo any prime power p^a, except for $p = 2, a > 2$, and there are $\phi(\phi(p^a))$ many. For $p = 2, a > 2$, the group $\mathbb{Z}_{2^a}^{\times}$ is instead isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

4.8 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 $n < 1e19$.

Combinatorial (5)

Graph (6)

globalmincut.h
Description: Given an adjacency matrix returns the global mincut and the vertices of one of the cuts.
Time: $\mathcal{O}(V^3)$
84 lines

```
/*
 * If you dont need the cut you can eliminate every thing with
     this coment "// *********"
 *Explanation of algorithm:
 *  -getting the mincut value: it does n-1 iterations. In each
      iteration it starts by a vertex (random) as set A.
 *  then it iterates until only two vertices are left by adding
      to set A the most tightly connected vertex to A (vertex
      not in A).
 *  it insert this vertex to A. When only two vertices are left
      , the mincut between those two is the weight W of the
      edges between
 *  the last vertex and A. mincut = min(mincut, W)
 *  We then merge the two last vertices and start again.
 *
 *  -getting the cut: basically when we merge two nodes we
      merge them with mfset. When we obtain a new best mincut
      value, a cut
 *  is represented buy the nodes in the same component as the
      last node;
 */

// Maximum number of vertices in the graph
#define NN 256
// Maximum edge weight (MAXW * NN * NN must fit into an int)
#define MAXW 1000
// Adjacency matrix and some internal arrays
int v[NN], w[NN];
bool a[NN];

int pare[NN]; // *********
int par (int b){ // *********
  if (pare[b] == b) return b;
```

```
  pare[b] = par(pare[b]);
  return pare[b];
}

inline void merge (int b, int c){ // **********
  pare[par(b)] = par(c);
}

pair < int, vi > minCut(vvi& g, int n) {
  int n1 = n;
  // init the remaining vertex set
  for (int i = 0; i < n; i++){
    v[i] = i;
    pare[i] = i; // **********
  }
  // run Stoer-Wagner
  int best = MAXW * n * n;
  vi cut; // **********
  while (n > 1) {
    // initialize the set A and vertex weights
    a[v[0]] = true;
    for (int i = 1; i < n; i++) {
      a[v[i]] = false;
      w[i] = g[v[0]][v[i]];
    }
    // add the other vertices
    int prev = v[0];
    for (int i = 1; i < n; i++) {
      // find the most tightly connected non-A vertex
      int zj = -1;
      for (int j = 1; j < n; j++)
        if (!a[v[j]] && (zj < 0 || w[j] > w[zj])) zj = j;
      // add it to A
      a[v[zj]] = true;
      // last vertex?
      if (i == n - 1) {
        // remember the cut weight
        if (best > w[zj]){
          best = w[zj];
          cut.clear(); // ********
          for (int ko = 0; ko < n1; ko++) if (par(ko) == par(v[
               zj])) cut.push_back(ko); // ********
        }

        // merge prev and v[zj]
        merge(prev, v[zj]); // *************
        for (int j = 0; j < n; j++)
          g[v[j]][prev] = g[prev][v[j]] += g[v[zj]][v[j]];
        v[zj] = v[--n];
        break;
      }
      prev = v[zj];
      // update the weights of its neighbours
      for (int j = 1; j < n; j++)
        if (!a[v[j]]) w[j] += g[v[zj]][v[j]];
    }
  }
  return {best, cut};
}
```

EulerianCycle.h

Description: returns de eulerian cycle/tour starting at u, cycle is in reverse order. If its a tour it must start at a vertex with odd degree. It is common to add edges between odd vertex to find a pseudo euler tour.
Usage: Call find cycle with a vertex where a eulerian tour/cycle is possible, when adding edges make sure that two vertex have the same edge iff it is undirected.
Time: $\mathcal{O}(E)$ 25 lines

```
typedef vector<int> vi;
struct edge{
  int u, v;
  bool used;
};

void Eulerdfs(int u, vi &nxt, vi &Euler, vector<edge> &E, const
      vector<vi> &adj) {
  while(nxt[u] < adj[u].size()){
    int go = adj[u][nxt[u]++];
    if(!E[go].used){
      E[go].used = 1;
      int to = (E[go].u ^ E[go].v ^ u);
      Eulerdfs(to, nxt, Euler, E, adj);
    }
  }
  Euler.push_back(u);
}

vi Eulerian(int u, vector<edge> &E, const vector<vi> &adj) {
  vi nxt (adj.size(),0);
  vi Euler;
  Eulerdfs(u, nxt, Euler, E, adj);
  reverse(Euler.begin(), Euler.end());
  return Euler;
}
```

MaxFlow.h

Description: Returns maximum flow.
Usage: To obtain a cut in the mincut problem one must bfs from the source. All the vertices reached from it using only edges with capacity > 0 are in the same cut
Time: $\mathcal{O}(VE^2)$ for general graphs and $\mathcal{O}(E * sqrt(V))$ for the maximum matching problem (bipartite unit weighted graf). It is generally very fast. 67 lines

```
typedef long long ll;
typedef vector<int> VI;
typedef vector<VI> VVI;
const ll INF = 1000000000000000000LL;

#define VEI(w,e) ((E[e].u == w) ? E[e].v : E[e].u)
#define CAP(w,e) ((E[e].u == w) ? E[e].cap[0] - E[e].flow : E[e
     ].cap[1] + E[e].flow)
#define ADD(w,e,f) E[e].flow += ((E[e].u == w) ? (f) : (-(f)))

struct Edge { int u, v; ll cap[2], flow; };

VI d, act;

bool bfs(int s, int t, VVI& adj, vector<Edge>& E) {
  queue<int> Q;
  d = VI(adj.size(), -1);
  d[t] = 0;
  Q.push(t);
  while (not Q.empty()) {
    int u = Q.front(); Q.pop();
    for (int i = 0; i < int(adj[u].size()); ++i) {
      int e = adj[u][i], v = VEI(u, e);
      if (CAP(v, e) > 0 and d[v] == -1) {
        d[v] = d[u] + 1;
        Q.push(v);
      }
    }
  }
  return d[s] >= 0;
}

ll dfs(int u,int t,ll bot,VVI& adj,vector<Edge>& E) {
  if (u == t) return bot;
  for (; act[u] < int(adj[u].size()); ++act[u]) {
    int e = adj[u][act[u]];
    if (CAP(u, e) > 0 and d[u] == d[VEI(u, e)] + 1) {
      ll inc=dfs(VEI(u,e),t,min(bot,CAP(u,e)),adj,E);
      if (inc) {
        ADD(u, e, inc);
        return inc;
      }
    }
  }
  return 0;
}

ll maxflow(int s, int t, VVI& adj, vector<Edge>& E) {
  for (int i=0; i<int(E.size()); ++i) E[i].flow = 0;
  ll flow = 0, bot;
  while (bfs(s, t, adj, E)) {
    act = VI(adj.size(), 0);
    while ((bot = dfs(s,t,INF, adj, E))) flow += bot;
  }
  return flow;
}

void addEdge(int u, int v, VVI& adj, vector<Edge>& E, ll cap){
  Edge e;
  e.u = u;
  e.v = v;
  e.cap[0] = cap;
  e.cap[1] = 0;
  e.flow = 0;
  adj[u].push_back(E.size());
  adj[v].push_back(E.size());
  E.push_back(e);
}
```

SCC.h

Description: Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa.
Usage: Use addedge to addedges in a directed graph(will also add reverse edges), after calling Kosaraju comp will save the component number of each vertex ordered by topological order.
Time: $\mathcal{O}(E+V)$ 27 lines

```
const int MAXN = 100010;
stack<int> st;
int m[MAXN], comp[MAXN];
vector<int> adj[2][MAXN];
int c = 0;

void addedge(vector<vector<int>> &adj, int u, int v) {
    adj[0][u].push_back(v);
    adj[1][v].push_back(u);
}

void dfs(int u, int t, vector<int>& m) {
    m[u] = 1;
    for(int v : adj[t][u]) if(!m[v]) dfs(v,t);
    if(t) comp[u] = c;
    else st.push(u);
}

void kosaraju(int n) {
    vector<int> m(n,0);
    for(int i = 0; i < n; ++i) if(!m[i]) dfs(i,0,m);
    m = vector<int>(n,0);
    for(;st.size();st.pop()) {
        int u = st.top();
        if(!m[u]) dfs(u,1), c++;
```

```
  }
}
```

Geometry (7)

7.1 Geometric primitives

Point.h

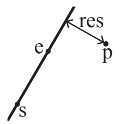
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

25 lines

```
template <class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y) {}
  bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
  bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
  P operator+(P p) const { return P(x+p.x, y+p.y); }
  P operator-(P p) const { return P(x-p.x, y-p.y); }
  P operator*(T d) const { return P(x*d, y*d); }
  P operator/(T d) const { return P(x/d, y/d); }
  T dot(P p) const { return x*p.x + y*p.y; }
  T cross(P p) const { return x*p.y - y*p.x; }
  T cross(P a, P b) const { return (a-*this).cross(b-*this); }
  T dist2() const { return x*x + y*y; }
  double dist() const { return sqrt((double)dist2()); }
  // angle to x-axis in interval [-pi, pi]
  double angle() const { return atan2(y, x); }
  P unit() const { return *this/dist(); } // makes dist()=1
  P perp() const { return P(-y, x); } // rotates +90 degrees
  P normal() const { return perp().unit(); }
  // returns point rotated 'a' radians ccw around the origin
  P rotate(double a) const {
    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
};
```

lineDistance.h

Description:

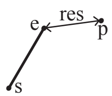


Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance.

"Point.h" 4 lines

```
template <class P>
double lineDist(const P& a, const P& b, const P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist();
}
```

SegmentDistance.h



Description:
Returns the shortest distance between point p and the line segment from point s to e.

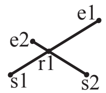
Usage: `Point<double> a, b(2,2), p(1,1);`
`bool onSegment = segDist(a,b,p) < 1e-10;`

"Point.h" 6 lines

```
typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
  auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)));
  return ((p-s)*d-(e-s)*t).dist()/d;
}
```

SegmentIntersection.h

Description:



If a unique intersetion point between the line segments going from s1 to e1 and from s2 to e2 exists r1 is set to this point and 1 is returned. If no intersection point exists 0 is returned and if infinitely many exists 2 is returned and r1 and r2 are set to the two ends of the common line. The wrong position will be returned if P is Point<int> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long. Use segmentIntersectionQ to get just a true/false answer.

Usage: `Point<double> intersection, dummy;`
`if (segmentIntersection(s1,e1,s2,e2,intersection,dummy)==1)`
`cout << "segments intersect at " << intersection << endl;`

"Point.h" 27 lines

```
template <class P>
int segmentIntersection(const P& s1, const P& e1,
    const P& s2, const P& e2, P& r1, P& r2) {
  if (e1==s1) {
    if (e2==s2) {
      if (e1==e2) { r1 = e1; return 1; } //all equal
      else return 0; //different point segments
    } else return segmentIntersection(s2,e2,s1,e1,r1,r2);//swap
  }
  //segment directions and separation
  P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
  auto a = v1.cross(v2), a1 = v1.cross(d), a2 = v2.cross(d);
  if (a == 0) { //if parallel
    auto b1=s1.dot(v1), c1=e1.dot(v1),
         b2=s2.dot(v1), c2=e2.dot(v1);
    if (a1 || a2 || max(b1,min(b2,c2))>min(c1,max(b2,c2)))
      return 0;
    r1 = min(b2,c2)<b1 ? s1 : (b2<c2 ? s2 : e2);
    r2 = max(b2,c2)>c1 ? e1 : (b2>c2 ? s2 : e2);
    return 2-(r1==r2);
  }
  if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
  if (0<a1 || a<-a1 || 0<a2 || a<-a2)
    return 0;
  r1 = s1-v1*a2/a;
  return 1;
}
```

SegmentIntersectionQ.h

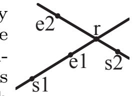
Description: Like segmentIntersection, but only returns true/false. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

"Point.h" 16 lines

```
template <class P>
bool segmentIntersectionQ(P s1, P e1, P s2, P e2) {
  if (e1 == s1) {
    if (e2 == s2) return e1 == e2;
    swap(s1,s2); swap(e1,e2);
  }
  P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
  auto a = v1.cross(v2), a1 = d.cross(v1), a2 = d.cross(v2);
  if (a == 0) { // parallel
    auto b1 = s1.dot(v1), c1 = e1.dot(v1),
         b2 = s2.dot(v1), c2 = e2.dot(v1);
    return !a1 && max(b1,min(b2,c2)) <= min(c1,max(b2,c2));
  }
  if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
  return (0 <= a1 && a1 <= a && 0 <= a2 && a2 <= a);
}
```

lineIntersection.h

Description:



If a unique intersetion point of the lines going through s1,e1 and s2,e2 exists r is set to this point and 1 is returned. If no intersection point exists 0 is returned and if infinitely many exists -1 is returned. If s1==e1 or s2==e2 -1 is returned. The wrong position will be returned if P is Point<int> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

Usage: `point<double> intersection;`
`if (1 == LineIntersection(s1,e1,s2,e2,intersection))`
`cout << "intersection point at " << intersection << endl;`

"Point.h" 9 lines

```
template <class P>
int lineIntersection(const P& s1, const P& e1, const P& s2,
    const P& e2, P& r) {
  if ((e1-s1).cross(e2-s2)) { //if not parallell
    r = s2-(e2-s2)*(e1-s1).cross(s2-s1)/(e1-s1).cross(e2-s2);
    return 1;
  } else
    return -((e1-s1).cross(s2-s1)==0 || s2==e2);
}
```

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 $\Leftrightarrow$ left/on line/right. If the optional argument *eps* is given 0 is returned if p is within distance *eps* from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.

Usage: `bool left = sideOf(p1,p2,q)==1;`

"Point.h" 11 lines

```
template <class P>
int sideOf(const P& s, const P& e, const P& p) {
  auto a = (e-s).cross(p-s);
  return (a > 0) - (a < 0);
}
template <class P>
int sideOf(const P& s, const P& e, const P& p, double eps) {
  auto a = (e-s).cross(p-s);
  double l = (e-s).dist()*eps;
  return (a > l) - (a < -l);
}
```

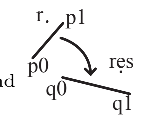
onSegment.h

Description: Returns true iff p lies on the line segment from s to e. Intended for use with e.g. Point<long long> where overflow is an issue. Use (segDist(s,e,p)<=epsilon) instead when using Point<double>.

"Point.h" 5 lines

```
template <class P>
bool onSegment(const P& s, const P& e, const P& p) {
  P ds = p-s, de = p-e;
  return ds.cross(de) == 0 && ds.dot(de) <= 0;
}
```

linearTransformation.h



Description:
Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.

"Point.h" 6 lines

```
typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
  return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
}
```

Angle.h

Description: A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping.

Usage:
```
vector<Angle> v = {w[0], w[0].t360() ...}; // sorted
int j = 0; rep(i,0,n) {
while (v[j] < v[i].t180()) ++j;
} // sweeps j such that (j-i) represents the number of
positively oriented triangles with vertices at 0 and i
```
37 lines

```
struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
  Angle operator-(Angle a) const { return {x-a.x, y-a.y, t}; }
  int quad() const {
    assert(x || y);
    if (y < 0) return (x >= 0) + 2;
    if (y > 0) return (x <= 0);
    return (x <= 0) * 2;
  }
  Angle t90() const { return {-y, x, t + (quad() == 3)}; }
  Angle t180() const { return {-x, -y, t + (quad() >= 2)}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.quad(), a.y * (ll)b.x) <
         make_tuple(b.t, b.quad(), a.x * (ll)b.y);
}
bool operator>=(Angle a, Angle b) { return !(a < b); }
bool operator>(Angle a, Angle b) { return b < a; }
bool operator<=(Angle a, Angle b) { return !(b < a); }

// Given two points, this calculates the smallest angle between
// them, i.e., the angle that covers the defined line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.t360()));
}

Angle operator+(Angle a, Angle b) { // where b is a vector
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (r > a.t180()) r.t--;
  return r.t180() < a ? r.t360() : r;
}
```

7.2 Circles

CircleIntersection.h

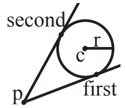
Description: Computes a pair of points at which two circles intersect. Returns false in case of no intersection.

"Point.h" 14 lines

```
typedef Point<double> P;
bool circleIntersection(P a, P b, double r1, double r2,
    pair<P, P>* out) {
  P delta = b - a;
  assert(delta.x || delta.y || r1 != r2);
  if (!delta.x && !delta.y) return false;
  double r = r1 + r2, d2 = delta.dist2();
  double p = (d2 + r1*r1 - r2*r2) / (2.0 * d2);
  double h2 = r1*r1 - p*p*d2;
  if (d2 > r*r || h2 < 0) return false;
  P mid = a + delta*p, per = delta.perp() * sqrt(h2 / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

circleTangents.h

Description:



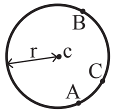
Returns a pair of the two points on the circle with radius r centered around c whos tangent lines intersect p. If p lies within the circle NaN-points are returned. P is intended to be Point<double>. The first point is the one to the right as seen from the p towards c.

Usage:
```
typedef Point<double> P;
pair<P,P> p = circleTangents(P(100,2),P(0,0),2);
```
"Point.h" 6 lines

```
template <class P>
pair<P,P> circleTangents(const P &p, const P &c, double r) {
  P a = p-c;
  double x = r*r/a.dist2(), y = sqrt(x-x*x);
  return make_pair(c+a*x+a.perp()*y, c+a*x-a.perp()*y);
}
```

circumcircle.h

Description:



The circumcirle of a triangle is the circle intersecting all three vertices. ccRadius returns the radius of the circle going through points A, B and C and ccCenter returns the center of the same circle.

"Point.h" 9 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/
      abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```

MinimumEnclosingCircle.h

Description: Computes the minimum circle that encloses a set of points.

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

"circumcircle.h" 28 lines

```
pair<double, P> mec2(vector<P>& S, P a, P b, int n) {
  double hi = INFINITY, lo = -hi;
  rep(i,0,n) {
    auto si = (b-a).cross(S[i]-a);
    if (si == 0) continue;
    P m = ccCenter(a, b, S[i]);
    auto cr = (b-a).cross(m-a);
    if (si < 0) hi = min(hi, cr);
    else lo = max(lo, cr);
  }
  double v = (0 < lo ? lo : hi < 0 ? hi : 0);
  P c = (a + b) / 2 + (b - a).perp() * v / (b - a).dist2();
  return {(a - c).dist2(), c};
}
pair<double, P> mec(vector<P>& S, P a, int n) {
  random_shuffle(S.begin(), S.begin() + n);
  P b = S[0], c = (a + b) / 2;
  double r = (a - c).dist2();
  rep(i,1,n) if ((S[i] - c).dist2() > r * (1 + 1e-8)) {
    tie(r,c) = (n == sz(S) ?
      mec(S, S[i], i) : mec2(S, a, S[i], i));
  }
  return {r, c};
}
pair<double, P> enclosingCircle(vector<P> S) {
  assert(!S.empty()); auto r = mec(S, S[0], sz(S));
  return {sqrt(r.first), r.second};
}
```

7.3 Polygons

insidePolygon.h

Description: Returns true if p lies within the polygon described by the points between iterators begin and end. If strict false is returned when p is on the edge of the polygon. Answer is calculated by counting the number of intersections between the polygon and a line going from p to infinity in the positive x-direction. The algorithm uses products in intermediate steps so watch out for overflow. If points within epsilon from an edge should be considered as on the edge replace the line "if (onSegment..." with the comment bellow it (this will cause overflow for int and long long).

Usage:
```
typedef Point<int> pi;
vector<pi> v; v.push_back(pi(4,4));
v.push_back(pi(1,2)); v.push_back(pi(2,1));
bool in = insidePolygon(v.begin(),v.end(), pi(3,4), false);
```
Time: $\mathcal{O}(n)$

"Point.h", "onSegment.h", "SegmentDistance.h" 14 lines

```
template <class It, class P>
bool insidePolygon(It begin, It end, const P& p,
    bool strict = true) {
  int n = 0; //number of isects with line from p to (inf,p.y)
  for (It i = begin, j = end-1; i != end; j = i++) {
    //if p is on edge of polygon
    if (onSegment(*i, *j, p)) return !strict;
    //or: if (segDist(*i, *j, p) <= epsilon) return !strict;
    //increment n if segment intersects line from p
    n += (max(i->y,j->y) > p.y && min(i->y,j->y) <= p.y &&
        ((*j-*i).cross(p-*i) > 0) == (i->y <= p.y));
  }
  return n&1; //inside if odd number of intersections
}
```

PolygonArea.h

Description: Returns twice the signed area of a polygon. Clockwise enumeration gives negative area. Watch out for overflow if using int as T!

"Point.h" 6 lines

```
template <class T>
T polygonArea2(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
  return a;
}
```

PolygonCenter.h

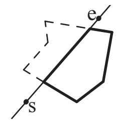
Description: Returns the center of mass for a polygon.

"Point.h" 10 lines

```
typedef Point<double> P;
Point<double> polygonCenter(vector<P>& v) {
  auto i = v.begin(), end = v.end(), j = end-1;
  Point<double> res{0,0}; double A = 0;
  for (; i != end; j=i++) {
    res = res + (*i + *j) * j->cross(*i);
    A += j->cross(*i);
  }
  return res / A / 3;
}
```

PolygonCut.h

Description:



Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.

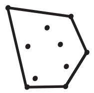
Usage:
```
vector<P> p = ...;
p = polygonCut(p, P(0,0), P(1,0));
```
"Point.h", "lineIntersection.h" 15 lines

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
```

```
  vector<P> res;
  rep(i,0,sz(poly)) {
    P cur = poly[i], prev = i ? poly[i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0)) {
      res.emplace_back();
      lineIntersection(s, e, cur, prev, res.back());
    }
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

ConvexHull.h

Description:
Returns a vector of indices of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.



Usage: `vector<P> ps, hull;`
`trav(i, convexHull(ps)) hull.push_back(ps[i]);`
Time: $\mathcal{O}(n \log n)$

"Point.h" 20 lines

```
typedef Point<ll> P;
pair<vi, vi> ulHull(const vector<P>& S) {
  vi Q(sz(S)), U, L;
  iota(all(Q), 0);
  sort(all(Q), [&S](int a, int b){ return S[a] < S[b]; });
  trav(it, Q) {
#define ADDP(C, cmp) while (sz(C) > 1 && S[C[sz(C)-2]].cross(\
  S[it], S[C.back()]) cmp 0) C.pop_back(); C.push_back(it);
    ADDP(U, <=); ADDP(L, >=);
  }
  return {U, L};
}

vi convexHull(const vector<P>& S) {
  vi u, l; tie(u, l) = ulHull(S);
  if (sz(S) <= 1) return u;
  if (S[u[0]] == S[u[1]]) return {0};
  l.insert(l.end(), u.rbegin()+1, u.rend()-1);
  return l;
}
```

PolygonDiameter.h

Description: Calculates the max squared distance of a set of points.

"ConvexHull.h" 19 lines

```
vector<pii> antipodal(const vector<P>& S, vi& U, vi& L) {
  vector<pii> ret;
  int i = 0, j = sz(L) - 1;
  while (i < sz(U) - 1 || j > 0) {
    ret.emplace_back(U[i], L[j]);
    if (j == 0 || (i != sz(U)-1 && (S[L[j]] - S[L[j-1]])
          .cross(S[U[i+1]] - S[U[i]]) > 0)) ++i;
    else --j;
  }
  return ret;
}

pii polygonDiameter(const vector<P>& S) {
  vi U, L; tie(U, L) = ulHull(S);
  pair<ll, pii> ans;
  trav(x, antipodal(S, U, L))
    ans = max(ans, {(S[x.first] - S[x.second]).dist2(), x});
  return ans.second;
}
```

PointInsideHull.h

Description: Determine whether a point t lies inside a given polygon (counter-clockwise order). The polygon must be such that every point on the circumference is visible from the first point in the vector. It returns 0 for points outside, 1 for points on the circumference, and 2 for points inside.
Time: $\mathcal{O}(\log N)$

"Point.h", "sideOf.h", "onSegment.h" 22 lines

```
typedef Point<ll> P;
int insideHull2(const vector<P>& H, int L, int R, const P& p) {
  int len = R - L;
  if (len == 2) {
    int sa = sideOf(H[0], H[L], p);
    int sb = sideOf(H[L], H[L+1], p);
    int sc = sideOf(H[L+1], H[0], p);
    if (sa < 0 || sb < 0 || sc < 0) return 0;
    if (sb==0 || (sa==0 && L == 1) || (sc == 0 && R == sz(H)))
      return 1;
    return 2;
  }
  int mid = L + len / 2;
  if (sideOf(H[0], H[mid], p) >= 0)
    return insideHull2(H, mid, R, p);
  return insideHull2(H, L, mid+1, p);
}

int insideHull(const vector<P>& hull, const P& p) {
  if (sz(hull) < 3) return onSegment(hull[0], hull.back(), p);
  else return insideHull2(hull, 1, sz(hull), p);
}
```

LineHullIntersection.h

Description: Line-convex polygon intersection. The polygon must be ccw and have no colinear points. isct(a, b) returns a pair describing the intersection of a line with the polygon: • $(-1,-1)$ if no collision, • $(i,-1)$ if touching the corner i, • (i,i) if along side $(i,i+1)$, • (i,j) if crossing sides $(i,i+1)$ and $(j,j+1)$. In the last case, if a corner i is crossed, this is treated as happening on side $(i,i+1)$. The points are returned in the same order as the line hits the polygon.
Time: $\mathcal{O}(N + Q \log n)$

"Point.h" 63 lines

```
ll sgn(ll a) { return (a > 0) - (a < 0); }
typedef Point<ll> P;
struct HullIntersection {
  int N;
  vector<P> p;
  vector<pair<P, int>> a;

  HullIntersection(const vector<P>& ps) : N(sz(ps)), p(ps) {
    p.insert(p.end(), all(ps));
    int b = 0;
    rep(i,1,N) if (P{p[i].y,p[i].x} < P{p[b].y, p[b].x}) b = i;
    rep(i,0,N) {
      int f = (i + b) % N;
      a.emplace_back(p[f+1] - p[f], f);
    }
  }

  int qd(P p) {
    return (p.y < 0) ? (p.x >= 0) + 2
         : (p.x <= 0) * (1 + (p.y <= 0));
  }

  int bs(P dir) {
    int lo = -1, hi = N;
    while (hi - lo > 1) {
      int mid = (lo + hi) / 2;
      if (make_pair(qd(dir), dir.y * a[mid].first.x) <
        make_pair(qd(a[mid].first), dir.x * a[mid].first.y))
        hi = mid;
      else lo = mid;
    }
    return a[hi%N].second;
  }

  bool isign(P a, P b, int x, int y, int s) {
    return sgn(a.cross(p[x], b)) * sgn(a.cross(p[y], b)) == s;
  }

  int bs2(int lo, int hi, P a, P b) {
    int L = lo;
    if (hi < lo) hi += N;
    while (hi - lo > 1) {
      int mid = (lo + hi) / 2;
      if (isign(a, b, mid, L, -1)) hi = mid;
      else lo = mid;
    }
    return lo;
  }

  pii isct(P a, P b) {
    int f = bs(a - b), j = bs(b - a);
    if (isign(a, b, f, j, 1)) return {-1, -1};
    int x = bs2(f, j, a, b)%N,
        y = bs2(j, f, a, b)%N;
    if (a.cross(p[x], b) == 0 &&
        a.cross(p[x+1], b) == 0) return {x, x};
    if (a.cross(p[y], b) == 0 &&
        a.cross(p[y+1], b) == 0) return {y, y};
    if (a.cross(p[f], b) == 0) return {f, -1};
    if (a.cross(p[j], b) == 0) return {j, -1};
    return {x, y};
  }
};
```

7.4 Misc. Point Set Problems

closestPair.h

Description: $i1$, $i2$ are the indices to the closest pair of points in the point vector p after the call. The distance is returned.
Time: $\mathcal{O}(n \log n)$

"Point.h" 58 lines

```
template <class It>
bool it_less(const It& i, const It& j) { return *i < *j; }
template <class It>
bool y_it_less(const It& i,const It& j) {return i->y < j->y;}

template<class It, class IIt> /* IIt = vector<It>::iterator */
double cp_sub(IIt ya, IIt yaend, IIt xa, It &i1, It &i2) {
  typedef typename iterator_traits<It>::value_type P;
  int n = yaend-ya, split = n/2;
  if(n <= 3) { // base case
    double a = (*xa[1]-*xa[0]).dist(), b = 1e50, c = 1e50;
    if(n==3) b=(*xa[2]-*xa[0]).dist(), c=(*xa[2]-*xa[1]).dist()
      ;
    if(a <= b) { i1 = xa[1];
      if(a <= c) return i2 = xa[0], a;
      else return i2 = xa[2], c;
    } else { i1 = xa[2];
      if(b <= c) return i2 = xa[0], b;
      else return i2 = xa[1], c;
  } }
  vector<It> ly, ry, stripy;
  P splitp = *xa[split];
  double splitx = splitp.x;
  for(IIt i = ya; i != yaend; ++i) { // Divide
```

```
    if(*i != xa[split] && (**i-splitp).dist2() < 1e-12)
      return i1 = *i, i2 = xa[split], 0;// nasty special case!
    if (**i < splitp) ly.push_back(*i);
    else ry.push_back(*i);
  } // assert((signed)lefty.size() == split)
  It j1, j2; // Conquer
  double a = cp_sub(ly.begin(), ly.end(), xa, i1, i2);
  double b = cp_sub(ry.begin(), ry.end(), xa+split, j1, j2);
  if(b < a) a = b, i1 = j1, i2 = j2;
  double a2 = a*a;
  for(IIt i = ya; i != yaend; ++i) { // Create strip (y-sorted)
    double x = (*i)->x;
    if(x >= splitx-a && x <= splitx+a) stripy.push_back(*i);
  }
  for(IIt i = stripy.begin(); i != stripy.end(); ++i) {
    const P &p1 = **i;
    for(IIt j = i+1; j != stripy.end(); ++j) {
      const P &p2 = **j;
      if(p2.y-p1.y > a) break;
      double d2 = (p2-p1).dist2();
      if(d2 < a2) i1 = *i, i2 = *j, a2 = d2;
  } }
  return sqrt(a2);
}

template<class It> // It is random access iterators of point<T>
double closestpair(It begin, It end, It &i1, It &i2 ) {
  vector<It> xa, ya;
  assert(end-begin >= 2);
  for (It i = begin; i != end; ++i)
    xa.push_back(i), ya.push_back(i);
  sort(xa.begin(), xa.end(), it_less<It>);
  sort(ya.begin(), ya.end(), y_it_less<It>);
  return cp_sub(ya.begin(), ya.end(), xa.begin(), i1, i2);
}
```

kdTree.h

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

"Point.h" 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 the box is wider than high (not best
           heuristic...)
      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);
  }
};
```

DelaunayTriangulation.h

Description: Computes the Delaunay triangulation of a set of points. Each circumcircle contains none of the input points. If any three points are colinear or any four are on the same circle, behavior is undefined.

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

"Point.h", "3dHull.h" 10 lines

```
template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
  if (sz(ps) == 3) { int d = (ps[0].cross(ps[1], ps[2]) < 0);
    trifun(0,1+d,2-d); }
  vector<P3> p3;
  trav(p, ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (sz(ps) > 3) trav(t, hull3d(p3)) if ((p3[t.b]-p3[t.a]).
      cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
    trifun(t.a, t.c, t.b);
}
```

7.5 3D

PolyhedronVolume.h

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

6 lines

```
template <class V, class L>
double signed_poly_volume(const V& p, const L& trilist) {
  double v = 0;
  trav(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.

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" 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);
    }
  }
  trav(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. 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.

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)

AhoCorasick.h

Description: Builds an Ahocorasick Trie, with suffix links

Usage: This is an offline Algorithm, Pass the vector of patterns to Trie.init(), find function return the words matched in order of matching

Time: $\mathcal{O}(n+m+z)$

74 lines

```
const int MaxM = 200005;

struct Trie{
    static const int Alpha = 26;
    static const int first = 'a';
    int lst = 1;
    struct node{
        int nxt[Alpha] = {}, p = -1;
        char c;
        vector<int> end;
        //bitset<MaxN> bitmask;
        int SuffixLink;
    };
    vector<node> V;

    inline int getval(char c) {
        return c - first;
    }

    void CreateSuffixLink() {
        queue<int> q;
        for(q.push(0); q.size(); q.pop()) {
            int pos = q.front();      if(!pos) V[pos].
                SuffixLink = -1;
            else {
                int val = getval(V[pos].c);
                int j = V[V[pos].p].SuffixLink;
                while(j > -1 && !V[j].nxt[val]) j = V[j].
                    SuffixLink;
                if(j == -1) V[pos].SuffixLink = 0;
                else {
                    V[pos].SuffixLink = V[j].nxt[val];
                    for(auto &i : V[V[pos].SuffixLink].end) V[
                        pos].end.emplace_back(i);
                    //V[pos].bitmask |= V[V[pos].SuffixLink].
                        bitmask;
                }

            }
            for(int i = 0; i < Alpha; ++i) if(V[pos].nxt[i]) q.
                push(V[pos].nxt[i]);
        }
    }

    void init(vector<string> &v) {
        V.resize(MaxM);
        int id = 0;
        for(auto &s : v) {
            int pos = 0;
            for(char &c : s) {
                int val = getval(c);
                if(!V[pos].nxt[val]) {
                    V[lst].p = pos;
                    V[lst].c = c;
                    V[pos].nxt[val] = lst++;
                }
                pos = V[pos].nxt[val];
            }
            V[pos].end.emplace_back(id++);
            //V[pos].bitmask.set(id++);
        }
        CreateSuffixLink();
    }

    vector<int> find(string& word) {
        int pos = 0;
        vector<int> ans;
        for(auto &c : word) {
            int val = getval(c);
            while(pos > -1 && !V[pos].nxt[val]) pos = V[pos].
                SuffixLink;
            if(pos == -1) pos = 0;
            else pos = V[pos].nxt[val];
            if(V[pos].end.size()) {
                for(auto &i : V[pos].end) ans.emplace_back(i);
            }
        }
        return ans;
    }
};
```

PalindromeTree.h

Description: Palindrome Tree for string s

Time: $\mathcal{O}(sz(s))$ for building

64 lines

```
const int maxN = 1000010; // at least sz(s) + 3
struct Node {
  int suffix;
  int len;
  map<char, int> children;

  // not needed for construction, add if needed
  char c;
  int parent;
  vector<int> suffixof;
};

int nodeid;
Node tree[maxN]; // 0: -1 root, 1: empty string
int pos2node[maxN]; // not needed for construction

int add(int parent, char c) {
  if(has(tree[parent].children, c)) {
    return tree[parent].children[c];
  }
  int newid = nodeid++;
  tree[newid].suffix = -1;
  tree[newid].len = tree[parent].len + 2;
  tree[newid].parent = parent;
  tree[newid].c = c;
  tree[parent].children[c] = newid;
  return newid;
}

void build(string& s) {
  nodeid = 2;
  tree[0].parent = -1;
  tree[0].len = -1;
  tree[1].parent = -1;
  tree[0].suffixof.push_back(1);
  int cur = 0;
  FOR(i, 0, s.size()) {
    int newn = -1;
    while(1) {
      int curlen = tree[cur].len;
      if(i-1-curlen >= 0 && s[i-1-curlen] == s[i]) {
        newn = add(cur, s[i]);
        break;
      }
      cur = tree[cur].suffix;
    }
    pos2node[i] = newn;
    if(tree[newn].suffix != -1) {
      cur = newn;
      continue;
    }
    if(cur == 0) {
      tree[newn].suffix = 1;
    } else {
      do {
        cur = tree[cur].suffix;
      } while(i-1-tree[cur].len < 0
          || s[i-1-tree[cur].len] != s[i]);
      tree[newn].suffix = tree[cur].children[s[i]];
    }
    tree[tree[newn].suffix].suffixof.push_back(newn);
    cur = newn;
  }
}
```

SuffixArray.h

Description: lcp (x,y) = min(lcp(x,x+1),lcp(x+1,x+2)...lcp(y-1,y)) to answer queries with RMQ O(1)

Memory: Build $\mathcal{O}(N \log N)$ (Can be optimised as in B you only use the previous row to compute one row. But then you cannot do lcp
Time: Build: $\mathcal{O}(N \log N)$ where N is the length of the string for creation of the SA. LCP $\mathcal{O}(\log N)$ It is not necessary to use raddixsort if the $\mathcal{O}(nlog^2n)$ fits the time limit, one can just use stl sort.

65 lines

```
struct SF {
  pair<ll, ll> ord;
  ll id;
  bool operator<(const SF& s) const { return ord < s.ord; }
};

ll lcp(ll x, ll y, vector < vector < ll > > &B, ll N, ll step)
    {
  if (x == y) return N - x;
  ll res = 0;
  for (ll i = step - 1; i >= 0 and x < N and y < N; --i)
    if (B[i][x] == B[i][y]) { x += 1<<i; y += 1<<i; res += 1<<i
        ; }
  return res;
}

void raddixSort(vector < SF > & A, vector < vector <ll > > & B,
     vector < ll > &times, vector < ll > & pos, vector < SF >
    & L2, ll N) {
  ll k = max(N, 256LL);
  for (ll i = 0; i < k + 2; ++i) times[i] = 0;
  for (ll i = 0; i < N; ++i)
    times[A[i].ord.second + 1]++;
  pos[0] = 0;
  for (ll i = 1; i < k + 2; ++i)
    pos[i] = pos[i - 1] + times[i - 1];
  for (ll i = 0; i < N; ++i)
    L2[pos[A[i].ord.second + 1]++] = A[i];

  for (ll i = 0; i < k + 2; ++i)
    times[i] = 0;
  for (ll i = 0; i < N; ++i)
    times[L2[i].ord.first + 1]++;
  pos[0] = 0;
  for (ll i = 1; i < k + 2; ++i)
    pos[i] = pos[i - 1] + times[i - 1];
  for (ll i = 0; i < N; ++i)
    A[pos[L2[i].ord.first + 1]++] = L2[i];
}

void compute_suffix_array(vector < SF > & A, vector<vector<ll>
    > & B, ll N, string & S, ll &step) {
  ll MAXN = 3000005; //millor posar numero gran que algo en
      funcio de N pq peta
  vector < SF > L2(MAXN);
  vector < ll > pos(MAXN + 2,0), times(MAXN + 2,0);
  A.resize(N); B.resize(1); B[0].resize(N);
  for (ll i = 0; i < N; ++i) B[0][i] = S[i];
  step = 1;
  for (ll b = 0, pw = 1; b < N; ++step, pw<<=1) {
    for (ll i = 0; i < N; ++i) {
      A[i].ord.first = B[step - 1][i];
      A[i].ord.second = i + pw < N ? B[step - 1][i + pw] : -1;
      A[i].id = i;
    }
    raddixSort(A, B, times, pos, L2, N); //sort(A.begin(), A.
        end());
    B.resize(step + 1); B[step].resize(N);
    b = B[step][A[0].id] = 1;
    for (ll i = 1; i < N; ++i) {
      if (A[i - 1] < A[i]) ++b;
      B[step][A[i].id] = b;
    }
  }
}
```

Hashing.h

Description: Various self-explanatory methods for string hashing.

45 lines

```
typedef unsigned long long H;
static const H C = 123891739; // arbitrary

// Arithmetic mod 2^64-1. 5x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse).
// "typedef H K;" instead if you think test data is random.
struct K {
  typedef __uint128_t H2;
  H x; K(H x=0) : x(x) {}
  K operator+(K o){ return x + o.x + H(((H2)x + o.x)>>64); }
  K operator*(K o){ return K(x*o.x)+ H(((H2)x * o.x)>>64); }
  H operator-(K o) { K a = *this + ~o.x; return a.x + !~a.x; }
};

struct HashInterval {
  vector<K> ha, pw;
  HashInterval(string& str) : ha(str.size()+1), pw(ha) {
    pw[0] = 1;
    FOR(i,0,str.size())
      ha[i+1] = ha[i] * C + str[i],
      pw[i+1] = pw[i] * C;
  }
  H hashInterval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
};

vector<H> getHashes(string& str, int length) {
  if (str.size() < length) return {};
  K h = 0, pw = 1;
  FOR(i,0,length)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h - 0};
  FOR(i,length,str.size()) {
    ret.push_back(h * C + str[i] - pw * str[i-length]);
    h = ret.back();
  }
  return ret;
}

H hashString(string& s) {
  K h = 0;
  for(auto c:s) h = h * C + c;
  return h - 0;
}
```

Various (9)

Techniques (A)

techniques.txt

159 lines

```
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
Greedy algorithm
  Scheduling
  Max contigous subvector sum
  Invariants
  Huffman encoding
Graph teory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algoritm
  MST: Prim's algoritm
  Bellman-Ford
  Konig's theorem and vertex cover
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Eulercykler
  Flow networks
  * Augumenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cutvertices, cutedges och biconnected components
  Edge coloring
  * Trees
  Vertex coloring
  * Bipartite graphs (=> trees)
  * 3^n (special case of set cover)
  Diameter and centroid
  K'th shortest path
  Shortest cycle
Dynamic programmering
  Knapsack
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Dynprog over intervals
  Dynprog over subsets
  Dynprog over probabilities
  Dynprog over trees
  3^n set cover
  Divide and conquer
  Knuth optimization
  Convex hull optimizations
  RMQ (sparse table a.k.a 2^k-jumps)
  Bitonic cycle
  Log partitioning (loop over most restricted)
Combinatorics
  Computation of binomial coefficients
  Pigeon-hole principle
  Inclusion/exclusion
  Catalan number
  Pick's theorem
Number theory
  Integer parts
  Divisibility
  Euklidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's small theorem
  Euler's theorem
  Phi function
  Frobenius number
  Quadratic reciprocity
  Pollard-Rho
  Miller-Rabin
  Hensel lifting
  Vieta root jumping
Game theory
  Combinatorial games
  Game trees
  Mini-max
  Nim
  Games on graphs
  Games on graphs with loops
  Grundy numbers
  Bipartite games without repetition
  General games without repetition
  Alpha-beta pruning
Probability theory
Optimization
  Binary search
  Ternary search
  Unimodality and convex functions
  Binary search on derivative
Numerical methods
  Numeric integration
  Newton's method
  Root-finding with binary/ternary search
  Golden section search
Matrices
  Gaussian elimination
  Exponentiation by squaring
Sorting
  Radix sort
Geometry
  Coordinates and vectors
  * Cross product
  * Scalar product
  Convex hull
  Polygon cut
  Closest pair
  Coordinate-compression
  Quadtrees
  KD-trees
  All segment-segment intersection
Sweeping
  Discretization (convert to events and sweep)
  Angle sweeping
  Line sweeping
  Discrete second derivatives
Strings
  Longest common substring
  Palindrome subsequences
  Knuth-Morris-Pratt
  Tries
  Rolling polynom hashes
  Suffix array
  Suffix tree
  Aho-Corasick
  Manacher's algorithm
  Letter position lists
Combinatorial search
  Meet in the middle
  Brute-force with pruning
  Best-first (A*)
  Bidirectional search
  Iterative deepening DFS / A*
Data structures
  LCA (2^k-jumps in trees in general)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```