KTH Royal Institute of Technology

Omogen Heap

Simon Lindholm, Johan Sannemo, Mårten Wiman

2023-11-23

Strings (1)

KMP.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: $\mathcal{O}(n)$

d41d8c, 16 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  rep(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vi match(const string& s, const string& pat) {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
  return res;
}
```

Zfunc.h

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

d41d8c, 12 lines

```
vi Z(const string& S) {
  vi z(sz(S));
  int l = -1, r = -1;
  rep(i,1,sz(S)) {
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < sz(S) && S[i + z[i]] == S[z[i]])
      z[i]++;
    if (i + z[i] > r)
      l = i, r = i + z[i];
  }
  return z;
}
```

Manacher.h

Description: For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down).
Time: $\mathcal{O}(N)$

d41d8c, 13 lines

```
array<vi, 2> manacher(const string& s) {
  int n = sz(s);
  array<vi,2> p = {vi(n+1), vi(n)};
  rep(z,0,2) for (int i=0,l=0,r=0; i < n; i++) {
    int t = r-i+!z;
    if (i<r) p[z][i] = min(t, p[z][l+t]);
    int L = i-p[z][i], R = i+p[z][i]-!z;
    while (L>=1 && R+1<n && s[L-1] == s[R+1])
      p[z][i]++, L--, R++;
    if (R>r) l=L, r=R;
  }
  return p;
}
```

MinRotation.h

Description: Finds the lexicographically smallest rotation of a string.
Usage: `rotate(v.begin(), v.begin()+minRotation(v), v.end());`
Time: $\mathcal{O}(N)$

d41d8c, 8 lines

```
int minRotation(string s) {
  int a=0, N=sz(s); s += s;
  rep(b,0,N) rep(k,0,N) {
    if (a+k == b || s[a+k] < s[b+k]) {b += max(0, k-1); break;}
    if (s[a+k] > s[b+k]) { a = b; break; }
  }
  return a;
}
```

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

d41d8c, 23 lines

```
struct SuffixArray {
  vi sa, lcp;
  SuffixArray(string& s, int lim=256) { // or basic_string<int>
    int n = sz(s) + 1, k = 0, a, b;
    vi x(all(s)+1), y(n), ws(max(n, lim)), rank(n);
    sa = lcp = y, iota(all(sa), 0);
    for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
      p = j, iota(all(y), n - j);
      rep(i,0,n) if (sa[i] >= j) y[p++] = sa[i] - j;
      fill(all(ws), 0);
      rep(i,0,n) ws[x[i]]++;
      rep(i,1,lim) ws[i] += ws[i - 1];
      for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      rep(i,1,n) a = sa[i - 1], b = sa[i], x[b] =
        (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
    }
    rep(i,1,n) rank[sa[i]] = i;
    for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
      for (k && k--, j = sa[rank[i] - 1];
          s[i + k] == s[j + k]; k++);
  }
};
```

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).
Time: $\mathcal{O}(26N)$

d41d8c, 50 lines

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
      if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
        p[m++]=v; v=s[v]; q=r[v];  goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i;  p[m+1]=m;  l[m]=l[v];  r[m]=q;
      p[m]=p[v];  t[m][c]=m+1;  t[m][toi(a[q])]=v;
      l[v]=q;  p[v]=m;  t[p[m]][toi(a[l[m]])]=m;
      v=s[p[m]];  q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])];  q+=r[v]-l[v]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }

  SuffixTree(string a) : a(a) {
    fill(r,r+N,sz(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
    rep(i,0,sz(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA = 28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
    rep(c,0,ALPHA) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

Hashing.h

Description: Self-explanatory methods for string hashing.

d41d8c, 44 lines

```
// Arithmetic mod 2^64-1. 2x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse, where
// ABBA... and BAAB... of length 2^10 hash the same mod 2^64).
// "typedef ull H;" instead if you think test data is random,
// or work mod 10^9+7 if the Birthday paradox is not a problem.
typedef uint64_t ull;
struct H {
  ull x; H(ull x=0) : x(x) {}
  H operator+(H o) { return x + o.x + (x + o.x < x); }
  H operator-(H o) { return *this + ~o.x; }
  H operator*(H o) { auto m = (__uint128_t)x * o.x;
    return H((ull)m) + (ull)(m >> 64); }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (ll)1e11+3; // (order ~ 3e9; random also ok)

struct HashInterval {
  vector<H> ha, pw;
  HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
```

```
    pw[0] = 1;
    rep(i,0,sz(str))
      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 (sz(str) < length) return {};
  H h = 0, pw = 1;
  rep(i,0,length)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  rep(i,length,sz(str)) {
    ret.push_back(h = h * C + str[i] - pw * str[i-length]);
  }
  return ret;
}

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

AhoCorasick.h

Description: Aho-Corasick automaton, used for multiple pattern matching. Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0. find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll($-$, word) finds all words (up to $N\sqrt{N}$ many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input. For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.
Time: construction takes $\mathcal{O}(26N)$, where N = sum of length of patterns. find(x) is $\mathcal{O}(N)$, where N = length of x. findAll is $\mathcal{O}(NM)$. d41d8c, 66 lines

```
struct AhoCorasick {
  enum {alpha = 26, first = 'A'}; // change this!
  struct Node {
    // (nmatches is optional)
    int back, next[alpha], start = -1, end = -1, nmatches = 0;
    Node(int v) { memset(next, v, sizeof(next)); }
  };
  vector<Node> N;
  vi backp;
  void insert(string& s, int j) {
    assert(!s.empty());
    int n = 0;
    for (char c : s) {
      int& m = N[n].next[c - first];
      if (m == -1) { n = m = sz(N); N.emplace_back(-1); }
      else n = m;
    }
    if (N[n].end == -1) N[n].start = j;
    backp.push_back(N[n].end);
    N[n].end = j;
    N[n].nmatches++;
  }
  AhoCorasick(vector<string>& pat) : N(1, -1) {
    rep(i,0,sz(pat)) insert(pat[i], i);
    N[0].back = sz(N);
    N.emplace_back(0);

    queue<int> q;
    for (q.push(0); !q.empty(); q.pop()) {
      int n = q.front(), prev = N[n].back;
      rep(i,0,alpha) {
        int &ed = N[n].next[i], y = N[prev].next[i];
        if (ed == -1) ed = y;
        else {
          N[ed].back = y;
          (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
            = N[y].end;
          N[ed].nmatches += N[y].nmatches;
          q.push(ed);
        }
      }
    }
  }
  vi find(string word) {
    int n = 0;
    vi res; // ll count = 0;
    for (char c : word) {
      n = N[n].next[c - first];
      res.push_back(N[n].end);
      // count += N[n].nmatches;
    }
    return res;
  }
  vector<vi> findAll(vector<string>& pat, string word) {
    vi r = find(word);
    vector<vi> res(sz(word));
    rep(i,0,sz(word)) {
      int ind = r[i];
      while (ind != -1) {
        res[i - sz(pat[ind]) + 1].push_back(ind);
        ind = backp[ind];
      }
    }
    return res;
  }
};
```

DeBruijnSequence.h

Description: Calculate length-L DeBruijn sequence.
Usage: Returns 1-base index. K is the number of alphabet, N is the length of different substring, L is the return length ($0 <= L <= K\hat{}N$). vector<int> seq = de_bruijn(K, N, L);
Time: $\mathcal{O}(L)$, $N = L = 10^5, K = 10$ in 12ms. d41d8c, 23 lines

```
vector<int> de_bruijn(int K, int N, int L) {
    vector<int> ans, tmp;
    function<void(int)> dfs = [&](int T) {
        if((int)ans.size() >= L) return;
        if((int)tmp.size() == N) {
            if(N%T == 0)
                for(int i = 0; i < T && (int)ans.size() < L; i
                    ++)
                    ans.push_back(tmp[i]);
        } else {
            int k = ((int)tmp.size()-T >= 0 ? tmp[(int)tmp.size
                ()-T] : 1);
            tmp.push_back(k);
            dfs(T);
            tmp.pop_back();
            for(int i = k+1; i <= K; i++) {
                tmp.push_back(i);
                dfs((int)tmp.size());
                tmp.pop_back();
            }
        }
    };
    dfs(1);
    return ans;
}
```

Graph (2)

2.1 Fundamentals

Bridge.h

Description: Undirected connected graph, no self-loop. Find every bridges. Usual graph representation. dfs(here, par): returns fastest vertex which connected by some node in subtree of here, except here-parent edge.
Time: $\mathcal{O}(V + E)$, 180ms for $V = 10^5$ and $E = 10^6$ graph. d41d8c, 23 lines

```
const int MAX_N = 1e5 + 1;

vector<int> adj[MAX_N];
vector<pii> bridges;
int in[MAX_N];
int cnt = 0;

int dfs(int here, int parent = -1) {
    in[here] = cnt++;
    int ret = 1e9;
    for (int there: adj[here]) {
        if (there != parent) {
            if (in[there] == -1) {
                int subret = dfs(there, here);
                if (subret > in[here]) bridges.push_back({here,
                    there});
                ret = min(ret, subret);
            } else {
                ret = min(ret, in[there]);
            }
        }
    }
    return ret;
}
```

KthShortestPath.h

Description: Calculate Kth shortest path from s to t.
Usage: 0-base index. Vertex is 0 to n-1. KthShortestPath g(n); g.add_edge(s, e, cost); g.run(s, t, k);
Time: $\mathcal{O}(E \log V + K \log K)$, $V = E = K = 3 \times 10^5$ in 312ms, 144MB at yosupo. d41d8c, 75 lines

```
struct KthShortestPath {
    struct node{
        array<node*, 2> son; pair<ll, ll> val;
        node() : node(make_pair(-1e18, -1e18)) {}
        node(pair<ll, ll> val) : node(nullptr, nullptr, val) {}
        node(node *l, node *r, pair<ll, ll> val) : son({l,r}),
            val(val) {}
    };
    node* copy(node *x){ return x ? new node(x->son[0], x->son
        [1], x->val) : nullptr; }
    node* merge(node *x, node *y){ // precondition: x, y both
        points to new entity
        if(!x || !y) return x ? x : y;
        if(x->val > y->val) swap(x, y);
        int rd = rnd(0, 1);
        if(x->son[rd]) x->son[rd] = copy(x->son[rd]);
        x->son[rd] = merge(x->son[rd], y); return x;
    }

    struct edge{
        ll v, c, i; edge() = default;
        edge(ll v, ll c, ll i) : v(v), c(c), i(i) {}
    };

    vector<vector<edge>> gph, rev;
    int idx;
    vector<int> par, pae; vector<ll> dist; vector<node*> heap;
```

```
    KthShortestPath(int n) {
        gph = rev = vector<vector<edge>>(n);
        idx = 0;
    }

    void add_edge(int s, int e, ll x){
        gph[s].emplace_back(e, x, idx);
        rev[e].emplace_back(s, x, idx);
        assert(x >= 0); idx++;
    }

    void dijkstra(int snk){ // replace this to SPFA if edge weight is negative
        int n = gph.size();
        par = pae = vector<int>(n, -1);
        dist = vector<ll>(n, 0x3f3f3f3f3f3f3f3f);
        heap = vector<node*>(n, nullptr);
        priority_queue<pair<ll,ll>, vector<pair<ll,ll>>, greater<>> pq;
        auto enqueue = [&](int v, ll c, int pa, int pe){
            if(dist[v] > c) dist[v] = c, par[v] = pa, pae[v] = pe, pq.emplace(c, v);
        }; enqueue(snk, 0, -1, -1); vector<int> ord;
        while(!pq.empty()){
            auto [c,v] = pq.top(); pq.pop(); if(dist[v] != c) continue;
            ord.push_back(v); for(auto e : rev[v]) enqueue(e.v, c+e.c, v, e.i);
        }
        for(auto &v : ord){
            if(par[v] != -1) heap[v] = copy(heap[par[v]]);
            for(auto &e : gph[v]){
                if(e.i == pae[v]) continue;
                ll delay = dist[e.v] + e.c - dist[v];
                if(delay < 1e18) heap[v] = merge(heap[v], new node(make_pair(delay, e.v)));
            }
        }
    }
    vector<ll> run(int s, int e, int k){
        using state = pair<ll, node*>; dijkstra(e); vector<ll> ans;
        priority_queue<state, vector<state>, greater<state>> pq;
        if(dist[s] > 1e18) return vector<ll>(k, -1);
        ans.push_back(dist[s]);
        if(heap[s]) pq.emplace(dist[s] + heap[s]->val.first, heap[s]);
        while(!pq.empty() && ans.size() < k){
            auto [cst, ptr] = pq.top(); pq.pop(); ans.push_back(cst);
            for(int j=0; j<2; j++) if(ptr->son[j])
                                     pq.emplace(cst-ptr->val.first + ptr->son[j]->val.first, ptr->son[j]);
            int v = ptr->val.second;
            if(heap[v]) pq.emplace(cst + heap[v]->val.first, heap[v]);
        }
        while(ans.size() < k) ans.push_back(-1);
        return ans;
    }
};
```

TreeIsomorphism.h

Description: Calculate hash of given tree.
Usage: `1-base index. t.init(n); t.add_edge(a, b); (size, hash) = t.build(void); // size may contain dummy centroid.`
Time: $\mathcal{O}(N \log N)$, $N = 30$ and $\sum N \le 10^6$ in 256ms. d41d8c, 74 lines

```
const int MAX_N = 33;
ull A[MAX_N], B[MAX_N];

struct Tree {
    int n;
    vector<int> adj[MAX_N];
    int sz[MAX_N];
    vector<int> cent; // sz(cent) <= 2
    Tree() {}

    void init(int n) {
        this->n = n;
        for (int i=0; i<n+2; ++i) adj[i].clear();
        fill(sz, sz+n+2, 0);
        cent.clear();
    }

    void add_edge(int s, int e) {
        adj[s].push_back(e);
        adj[e].push_back(s);
    }

    int get_cent(int v, int b = -1) {
        sz[v] = 1;
        for (auto i: adj[v]) {
            if (i != b) {
                int now = get_cent(i, v);
                if (now <= n/2) sz[v] += now;
                else break;
            }
        }
        if (n - sz[v] <= n/2) cent.push_back(v);
        return sz[v];
    }

    int init() {
        get_cent(1);
        if (cent.size() == 1) return cent[0];
        int u = cent[0], v = cent[1], add = ++n;
        adj[u].erase(find(adj[u].begin(), adj[u].end(), v));
        adj[v].erase(find(adj[v].begin(), adj[v].end(), u));
        adj[add].push_back(u); adj[u].push_back(add);
        adj[add].push_back(v); adj[v].push_back(add);
        return add;
    }

    pair<int, ull> build(int v, int p = -1, int d = 1) {
        vector<pair<int, ull>> ch;
        for (auto i: adj[v]) {
            if (i != p) ch.push_back(build(i, v, d+1));
        }
        if (ch.empty()) return { 1, d };

        sort(ch.begin(), ch.end());
        ull ret = d;
        int tmp = 1;
        for (int j=0; j<ch.size(); ++j) {
            ret += A[d] ^ B[j] ^ ch[j].second;
            tmp += ch[j].first;
        }
        return { tmp, ret };
    }

    pair<int, ull> build() {
        return build(init());
    }
};

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
uniform_int_distribution<ull> urnd;

void solve() {
    for (int i=0; i<MAX_N; ++i) A[i] = urnd(rng), B[i] = urnd(rng);
}
```

2.2 Network flow

MinCostMaxFlow.h

Description: Set MAXN. Overflow is not checked.
Usage: `MCMF g; g.add_edge(s, e, cap, cost); g.solve(src, sink, total_size);`
Time: 216ms on almost K_n graph, for $n = 300$. d41d8c, 91 lines

```
// https://github.com/koosaga/olympiad/blob/master/Library/codes/combinatorial_optimization/flow_cost_dijkstra.cpp
const int MAXN = 800 + 5;

struct MCMF {
    struct Edge{ int pos, cap, rev; ll cost; };
    vector<Edge> gph[MAXN];
    void clear(){
        for(int i=0; i<MAXN; i++) gph[i].clear();
    }
    void add_edge(int s, int e, int x, ll c){
        gph[s].push_back({e, x, (int)gph[e].size(), c});
        gph[e].push_back({s, 0, (int)gph[s].size()-1, -c});
    }
    ll dist[MAXN];
    int pa[MAXN], pe[MAXN];
    bool inque[MAXN];
    bool spfa(int src, int sink, int n){
        memset(dist, 0x3f, sizeof(dist[0]) * n);
        memset(inque, 0, sizeof(inque[0]) * n);
        queue<int> que;
        dist[src] = 0;
        inque[src] = 1;
        que.push(src);
        bool ok = 0;
        while(!que.empty()){
            int x = que.front();
            que.pop();
            if(x == sink) ok = 1;
            inque[x] = 0;
            for(int i=0; i<gph[x].size(); i++){
                Edge e = gph[x][i];
                if(e.cap > 0 && dist[e.pos] > dist[x] + e.cost){
                    dist[e.pos] = dist[x] + e.cost;
                    pa[e.pos] = x;
                    pe[e.pos] = i;
                    if(!inque[e.pos]){
                        inque[e.pos] = 1;
                        que.push(e.pos);
                    }
                }
            }
        }
        return ok;
    }
    ll new_dist[MAXN];
    pair<bool, ll> dijkstra(int src, int sink, int n){
```

```
        priority_queue<pii, vector<pii>, greater<pii> > pq;
        memset(new_dist, 0x3f, sizeof(new_dist[0]) * n);
        new_dist[src] = 0;
        pq.emplace(0, src);
        bool isSink = 0;
        while(!pq.empty()) {
            auto tp = pq.top(); pq.pop();
            if(new_dist[tp.second] != tp.first) continue;
            int v = tp.second;
            if(v == sink) isSink = 1;
            for(int i = 0; i < gph[v].size(); i++){
                Edge e = gph[v][i];
                ll new_weight = e.cost + dist[v] - dist[e.pos];
                if(e.cap > 0 && new_dist[e.pos] > new_dist[v] +
                     new_weight){
                    new_dist[e.pos] = new_dist[v] + new_weight;
                    pa[e.pos] = v;
                    pe[e.pos] = i;
                    pq.emplace(new_dist[e.pos], e.pos);
                }
            }
        }
        return make_pair(isSink, new_dist[sink]);
    }

    pair<ll, ll> solve(int src, int sink, int n){
        spfa(src, sink, n);
        pair<bool, ll> path;
        pair<ll,ll> ret = {0,0};
        while((path = dijkstra(src, sink, n)).first){
            for(int i = 0; i < n; i++) dist[i] += min(ll(2e15),
                 new_dist[i]);
            ll cap = 1e18;
            for(int pos = sink; pos != src; pos = pa[pos]){
                cap = min(cap, (ll)gph[pa[pos]][pe[pos]].cap);
            }
            ret.first += cap;
            ret.second += cap * (dist[sink] - dist[src]);
            for(int pos = sink; pos != src; pos = pa[pos]){
                int rev = gph[pa[pos]][pe[pos]].rev;
                gph[pa[pos]][pe[pos]].cap -= cap;
                gph[pos][rev].cap += cap;
            }
        }
        return ret;
    }
};
```

Dinic.h

Description: 0-indexed. cf) $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite matching.
Usage: `Dinic g(n); g.add_edge(u, v, cap_uv, cap_vu); g.max_flow(s, t); g.clear_flow();`

d41d8c, 79 lines

```
struct Dinic {
    struct Edge {
        int a;
        ll flow;
        ll cap;
        int rev;
    };

    int n, s, t;
    vector<vector<Edge>> adj;
    vector<int> level;
    vector<int> cache;
    vector<int> q;
    Dinic(int _n) : n(_n) {
        adj.resize(n);
        level.resize(n);
        cache.resize(n);
        q.resize(n);
    }

    bool bfs() {
        fill(level.begin(), level.end(), -1);
        level[s] = 0;
        int l = 0, r = 1;
        q[0] = s;
        while (l < r) {
            int here = q[l++];
            for (auto[there, flow, cap, rev]: adj[here]) {
                if (flow < cap && level[there] == -1) {
                    level[there] = level[here] + 1;
                    if (there == t) return true;
                    q[r++] = there;
                }
            }
        }
        return false;
    }

    ll dfs(int here, ll extra_capa) {
        if (here == t) return extra_capa;
        for (int& i=cache[here]; i<adj[here].size(); ++i) {
            auto[there, flow, cap, rev] = adj[here][i];
            if (flow < cap && level[there] == level[here] + 1)
             {
                ll f = dfs(there, min(extra_capa, cap-flow));
                if (f > 0) {
                    adj[here][i].flow += f;
                    adj[there][rev].flow -= f;
                    return f;
                }
            }
        }
        return 0;
    }

    void clear_flow() {
        for (auto& v: adj) {
            for (auto& e: v) e.flow = 0;
        }
    }

    ll max_flow(int _s, int _t) {
        s = _s, t = _t;
        ll ret = 0;
        while (bfs()) {
            fill(cache.begin(), cache.end(), 0);
            while (true) {
                ll f = dfs(s, 2e18);
                if (f == 0) break;
                ret += f;
            }
        }
        return ret;
    }

    void add_edge(int u, int v, ll uv, ll vu) {
        adj[u].push_back({ v, 0, uv, (int)adj[v].size() });
        adj[v].push_back({ u, 0, vu, (int)adj[u].size()-1 });
    }
};
```

Hungarian.h

Description: Bipartite minimum weight matching. 1-base indexed. A[1..n][1..m] and $n \le m$ needed. pair(cost, matching) will be returned.
Usage: `auto ret = hungarian(A);`
Time: $\mathcal{O}\left(n^2 m\right)$, and 100ms for n = 500.

d41d8c, 41 lines

```
const ll INF = 1e18;

pair<ll, vector<int>> hungarian(const vector<vector<ll>>& A) {
    int n = (int)A.size()-1;
    int m = (int)A[0].size()-1;
    vector<ll> u(n+1), v(m+1), p(m+1), way(m+1);
    for (int i=1; i<=n; ++i) {
        p[0] = i;
        int j0 = 0;
        vector<ll> minv (m+1, INF);
        vector<char> used (m+1, false);
        do {
            used[j0] = true;
            int i0 = p[j0], j1;
            ll delta = INF;
            for (int j=1; j<=m; ++j) {
                if (!used[j]) {
                    ll cur = A[i0][j]-u[i0]-v[j];
                    if (cur < minv[j])
                        minv[j] = cur, way[j] = j0;
                    if (minv[j] < delta)
                        delta = minv[j], j1 = j;
                }
            }
            for (int j=0; j<=m; ++j)
                if (used[j])
                    u[p[j]] += delta, v[j] -= delta;
                else
                    minv[j] -= delta;
            j0 = j1;
        } while (p[j0] != 0);
        do {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    vector<int> match(n+1);
    for (int i=1; i<=m; ++i) match[p[i]] = i;
    return { -v[0], match };
}
```

GlobalMinCut.h

Description: Undirected graph with adj matrix. No edge means $adj[i][j] = 0$. 0-based index, and expect $N \times N$ adj matrix.
Time: $\mathcal{O}\left(V^3\right)$, $\sum V^3 = 5.5 \times 10^8$ in 640ms.

d41d8c, 24 lines

```
const int INF = 1e9;
int getMinCut(vector<vector<int>> &adj) {
    int n = adj.size();
    vector<int> used(n);
    int ret = INF;
    for (int ph=n-1; ph>=0; --ph) {
        vector<int> w = adj[0], added = used;
        int prev, k = 0;
        for (int i=0; i<ph; ++i) {
            prev = k;
            k = -1;
            for (int j = 1; j < n; j++) {
                if (!added[j] && (k == -1 || w[j] > w[k])) k =
                     j;
            }
            if (i+1 == ph) break;
            for (int j = 0; j < n; j++) w[j] += adj[k][j];
```

```
        added[k] = 1;
      }
      for (int i=0; i<n; ++i) adj[i][prev] = (adj[prev][i] +=
          adj[k][i]);
      used[k] = 1;
      ret = min(ret, w[k]);
    }
    return ret;
}
```

GomoryHu.h

Description: Given a list of edges representing an undirected flow graph, returns edges of the Gomory-Hu tree. The max flow between any pair of vertices is given by minimum edge weight along the Gomory-Hu tree path.
Usage: `0-base index. GomoryHuTree t; auto ret = t.solve(n, edges); 0 is root, ret[i] for` $i > 0$ `contains (cost, par)`
Time: $\mathcal{O}(V)$ Flow Computations, $V = 3000, E = 4500$ and special graph that flow always terminate in $\mathcal{O}(3(V + E))$ time in 4036ms. d41d8c, 33 lines

```
struct Edge {
    int s, e, x;
};

const int MAX_N = 500 + 1;

bool vis[MAX_N];

struct GomoryHuTree {
    vector<pii> solve(int n, const vector<Edge>& edges) { // i
        - j cut : i - j minimum edge cost. 0 based.
      vector<pii> ret(n); // if i > 0, stores pair(cost,
          parent)
      for(int i=1; i<n; i++){
        Dinic g(n);
        for (auto[s, e, x]: edges) g.add_edge(s, e, x, x);
        ret[i].first = g.max_flow(i, ret[i].second);

        memset(vis, 0, sizeof(vis));
        function<void(int)> dfs = [&](int x) {
          if (vis[x]) return;
          vis[x] = 1;
          for (auto& i: g.adj[x]) {
            if (i.cap - i.flow > 0) dfs(i.a);
          }
        };

        dfs(i);
        for (int j=i+1; j<n; j++) {
          if (ret[j].second == ret[i].second && vis[j])
              ret[j].second = i;
        }
      }
      return ret;
    }
};
```

2.3 Matching

2.3.1 Random notes on matching (and bipartite)

In general graph, complement of independent set is vertex cover, and reverse holds too.
In bipartite graph, cardinality of minimum vertex cover is equal to card of maximum matching (konig).
In poset (DAG), card of maximum anti chain is equal to minimum path cover (dilworth).
Poset is DAG which satisfy i-¿j and j-¿k edge means i-¿k (transitivity).

hopcroftKarp.h

Description: It contains several application of bipartite matching.
Usage: `Both left and right side of node number starts with 0. HopcraftKarp(n, m); g.add_edge(s, e);`
Time: $\mathcal{O}\left(E\sqrt{V}\right)$, min path cover $V = 10^4, E = 10^5$ in 20ms. d41d8c, 92 lines

```
struct HopcroftKarp{
    int n, m;
    vector<vector<int>> g;
    vector<int> dst, le, ri;
    vector<char> visit, track;
    HopcroftKarp(int n, int m) : n(n), m(m), g(n), dst(n), le(n
        , -1), ri(m, -1), visit(n), track(n+m) {}

    void add_edge(int s, int e){ g[s].push_back(e); }
    bool bfs(){
        bool res = false; queue<int> que;
        fill(dst.begin(), dst.end(), 0);
        for(int i=0; i<n; i++)if(le[i] == -1)que.push(i),dst[i
            ]=1;
        while(!que.empty()){
            int v = que.front(); que.pop();
            for(auto i : g[v]){
                if(ri[i] == -1) res = true;
                else if(!dst[ri[i]])dst[ri[i]]=dst[v]+1,que.
                    push(ri[i]);
            }
        }
        return res;
    }
    bool dfs(int v){
        if(visit[v]) return false; visit[v] = 1;
        for(auto i : g[v]){
            if(ri[i] == -1 || !visit[ri[i]] && dst[ri[i]] ==
                dst[v] + 1 && dfs(ri[i])){
                le[v] = i; ri[i] = v; return true;
            }
        }
        return false;
    }
    int maximum_matching(){
        int res = 0; fill(le.begin(), le.end(), -1); fill(ri.
            begin(), ri.end(), -1);
        while(bfs()){
            fill(visit.begin(), visit.end(), 0);
            for(int i=0; i<n; i++) if(le[i] == -1) res += dfs(i
                );
        }
        return res;
    }
    vector<pair<int,int>> maximum_matching_edges(){
        int matching = maximum_matching();
        vector<pair<int,int>> edges; edges.reserve(matching);
        for(int i=0; i<n; i++) if(le[i] != -1) edges.
            emplace_back(i, le[i]);
        return edges;
    }
    void dfs_track(int v){
        if(track[v]) return; track[v] = 1;
        for(auto i : g[v]) track[n+i] = 1, dfs_track(ri[i]);
    }
    tuple<vector<int>, vector<int>, int> minimum_vertex_cover()
        {
        int matching = maximum_matching(); vector<int> lv, rv;
        fill(track.begin(), track.end(), 0);
        for(int i=0; i<n; i++) if(le[i] == -1) dfs_track(i);
        for(int i=0; i<n; i++) if(!track[i]) lv.push_back(i);
        for(int i=0; i<m; i++) if(track[n+i]) rv.push_back(i);
        return {lv, rv, lv.size() + rv.size()}; // s(lv)+s(rv)=
            mat
    }
    tuple<vector<int>, vector<int>, int>
        maximum_independent_set(){
        auto [a,b,matching] = minimum_vertex_cover();
        vector<int> lv, rv; lv.reserve(n-a.size()); rv.reserve(
            m-b.size());
        for(int i=0, j=0; i<n; i++){
            while(j < a.size() && a[j] < i) j++;
            if(j == a.size() || a[j] != i) lv.push_back(i);
        }
        for(int i=0, j=0; i<m; i++){
            while(j < b.size() && b[j] < i) j++;
            if(j == b.size() || b[j] != i) rv.push_back(i);
        } // s(lv)+s(rv)=n+m-mat
        return {lv, rv, lv.size() + rv.size()};
    }
    vector<vector<int>> minimum_path_cover(){ // n == m
        int matching = maximum_matching();
        vector<vector<int>> res; res.reserve(n - matching);
        fill(track.begin(), track.end(), 0);
        auto get_path = [&](int v) -> vector<int> {
            vector<int> path{v}; // ri[v] == -1
            while(le[v] != -1) path.push_back(v=le[v]);
            return path;
        };
        for(int i=0; i<n; i++) if(!track[n+i] && ri[i] == -1)
            res.push_back(get_path(i));
        return res; // sz(res) = n-mat
    }
    vector<int> maximum_anti_chain(){ // n = m
        auto [a,b,matching] = minimum_vertex_cover();
        vector<int> res; res.reserve(n - a.size() - b.size());
        for(int i=0, j=0, k=0; i<n; i++){
            while(j < a.size() && a[j] < i) j++;
            while(k < b.size() && b[k] < i) k++;
            if((j == a.size() || a[j] != i) && (k == b.size()
                || b[k] != i)) res.push_back(i);
        }
        return res; // sz(res) = n-mat
    }
};
```

GeneralMatching.h

Description: Matching for general graphs.
Usage: `1-base index. match[] has real matching (maybe). GeneralMatching g(n); g.add_edge(a, b); int ret = g.run(void);`
Time: $\mathcal{O}(N^3)$, $N = 500$ in 20ms. d41d8c, 93 lines

```
const int MAX_N = 500 + 1;

struct GeneralMatching {
    int n, cnt;
    int match[MAX_N], par[MAX_N], chk[MAX_N], prv[MAX_N], vis[
        MAX_N];
    vector<int> g[MAX_N];
    GeneralMatching(int n): n(n) {
        // init
        cnt = 0;
        for (int i=0; i<=n; ++i) g[i].clear();
        memset(match, 0, sizeof match);
        memset(vis, 0, sizeof vis);
        memset(prv, 0, sizeof prv);
    }
```

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

    int lca(int u, int v) {
        for (cnt++; vis[u] != cnt; swap(u, v)) {
            if (u) vis[u] = cnt, u = find(prv[match[u]]);
        }
        return u;
    }

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

    void blossom(int u, int v, int rt, queue<int> &q) {
        for (; find(u) != rt; u = prv[v]) {
            prv[u] = v;
            par[u] = par[v = match[u]] = rt;
            if (chk[v] & 1) q.push(v), chk[v] = 2;
        }
    }

    bool augment(int u) {
        iota(par, par + MAX_N, 0);
        memset(chk, 0, sizeof chk);
        queue<int> q;
        q.push(u);
        chk[u] = 2;

        while (!q.empty()) {
            u = q.front();
            q.pop();
            for (auto v : g[u]) {
                if (chk[v] == 0) {
                    prv[v] = u;
                    chk[v] = 1;
                    q.push(match[v]);
                    chk[match[v]] = 2;
                    if (!match[v]) {
                        for (; u; v = u) {
                            u = match[prv[v]];
                            match[match[v] = prv[v]] = v;
                        }
                        return true;
                    }
                } else if (chk[v] == 2) {
                    int l = lca(u, v);
                    blossom(u, v, l, q);
                    blossom(v, u, l, q);
                }
            }
        }
        return false;
    }

    int run() {
        int ret = 0;
        vector<int> tmp(n-1); // not necessary, just for
             constant optimization
        iota(tmp.begin(), tmp.end(), 0);
        shuffle(tmp.begin(), tmp.end(), mt19937(0x1557));
        for (auto x: tmp) {
            if (!match[x]) {
                for (auto y: g[x]) {
                    if (!match[y]) {
                        match[x] = y;
                        match[y] = x;
                        ret++;
                        break;
                    }
                }
            }
        }
        for (int i=1; i<=n; i++) {
            if (!match[i]) ret += augment(i);
        }
        return ret;
    }
};
```

GeneralWeightedMatching.h

Description: Given a weighted undirected graph, return maximum matching.

Usage: 1-base index. `init(n); add_edge(a, b, w); (tot_weight, n_matches) = _solve(void); Note that get_lca function have a static variable.`

Time: $\mathcal{O}\left(N^3\right)$, $N = 500$ in 317ms at yosupo. d41d8c, 228 lines

```
static const int INF = INT_MAX;
static const int N = 500 + 1;

struct Edge {
    int u, v, w;
    Edge() {}
    Edge(int ui, int vi, int wi) : u(ui), v(vi), w(wi) {}
};

int n, n_x;
Edge g[N * 2][N * 2];
int lab[N * 2];
int match[N * 2], slack[N * 2], st[N * 2], pa[N * 2];
int flo_from[N * 2][N + 1], s[N * 2], vis[N * 2];
vector<int> flo[N * 2];
queue<int> q;

int e_delta(const Edge &e) {
    return lab[e.u] + lab[e.v] - g[e.u][e.v].w * 2;
}

void update_slack(int u, int x) {
    if (!slack[x] || e_delta(g[u][x]) < e_delta(g[slack[x]][x])
         ) slack[x] = u;
}

void set_slack(int x) {
    slack[x] = 0;
    for (int u = 1; u <= n; ++u) {
        if (g[u][x].w > 0 && st[u] != x && s[st[u]] == 0)
            update_slack(u, x);
    }
}

void q_push(int x) {
    if (x <= n) {
        q.push(x);
    } else {
        for (size_t i = 0; i < flo[x].size(); i++) q_push(flo[x
            ][i]);
    }
}

void set_st(int x, int b) {
    st[x] = b;
    if (x > n) {
        for (size_t i = 0; i < flo[x].size(); ++i) set_st(flo[x
            ][i], b);
    }
}

int get_pr(int b, int xr) {
    int pr = find(flo[b].begin(), flo[b].end(), xr) - flo[b].
         begin();
    if (pr % 2 == 1) {
        reverse(flo[b].begin() + 1, flo[b].end());
        return (int)flo[b].size() - pr;
    } else {
        return pr;
    }
}

void set_match(int u, int v) {
    match[u] = g[u][v].v;
    if (u <= n) return;
    Edge e = g[u][v];
    int xr = flo_from[u][e.u], pr = get_pr(u, xr);
    for (int i = 0; i < pr; ++i) set_match(flo[u][i], flo[u][i
         ^ 1]);
    set_match(xr, v);
    rotate(flo[u].begin(), flo[u].begin() + pr, flo[u].end());
}

void augment(int u, int v) {
    for (;;) {
        int xnv = st[match[u]];
        set_match(u, v);
        if (!xnv) return;
        set_match(xnv, st[pa[xnv]]);
        u = st[pa[xnv]], v = xnv;
    }
}

int get_lca(int u, int v) {
    static int t = 0;
    for (++t; u || v; swap(u, v)) {
        if (u == 0) continue;
        if (vis[u] == t) return u;
        vis[u] = t;
        u = st[match[u]];
        if (u) u = st[pa[u]];
    }
    return 0;
}

void add_blossom(int u, int lca, int v) {
    int b = n + 1;
    while (b <= n_x && st[b]) ++b;
    if (b > n_x) ++n_x;
    lab[b] = 0, s[b] = 0;
    match[b] = match[lca];
    flo[b].clear();
    flo[b].push_back(lca);
    for (int x = u, y; x != lca; x = st[pa[y]]) {
        flo[b].push_back(x), flo[b].push_back(y = st[match[x]])
            , q_push(y);
    }
    reverse(flo[b].begin() + 1, flo[b].end());
    for (int x = v, y; x != lca; x = st[pa[y]]) {
        flo[b].push_back(x), flo[b].push_back(y = st[match[x]])
            , q_push(y);
    }
    set_st(b, b);
    for (int x = 1; x <= n_x; ++x) g[b][x].w = g[x][b].w = 0;
    for (int x = 1; x <= n; ++x) flo_from[b][x] = 0;
    for (size_t i = 0; i < flo[b].size(); ++i) {
        int xs = flo[b][i];
        for (int x = 1; x <= n_x; ++x)
```

```
            if (g[b][x].w == 0 || e_delta(g[xs][x]) < e_delta(g
                [b][x])) {
                g[b][x] = g[xs][x], g[x][b] = g[x][xs];
            }
        for (int x = 1; x <= n; ++x)
            if (flo_from[xs][x]) flo_from[b][x] = xs;
    }
    set_slack(b);
}

void expand_blossom(int b) {
    for (size_t i = 0; i < flo[b].size(); ++i) set_st(flo[b][i
        ], flo[b][i]);
    int xr = flo_from[b][g[b][pa[b]].u], pr = get_pr(b, xr);
    for (int i = 0; i < pr; i += 2) {
        int xs = flo[b][i], xns = flo[b][i + 1];
        pa[xs] = g[xns][xs].u;
        s[xs] = 1, s[xns] = 0;
        slack[xs] = 0, set_slack(xns);
        q_push(xns);
    }
    s[xr] = 1, pa[xr] = pa[b];
    for (size_t i = pr + 1; i < flo[b].size(); ++i) {
        int xs = flo[b][i];
        s[xs] = -1, set_slack(xs);
    }
    st[b] = 0;
}

bool on_found_edge(const Edge &e) {
    int u = st[e.u], v = st[e.v];
    if (s[v] == -1) {
        pa[v] = e.u, s[v] = 1;
        int nu = st[match[v]];
        slack[v] = slack[nu] = 0;
        s[nu] = 0, q_push(nu);
    } else if (s[v] == 0) {
        int lca = get_lca(u, v);
        if (!lca) return augment(u, v), augment(v, u), true;
        else add_blossom(u, lca, v);
    }
    return false;
}

bool matching() {
    memset(s + 1, -1, sizeof(int) * n_x);
    memset(slack + 1, 0, sizeof(int) * n_x);
    q = queue<int>();
    for (int x = 1; x <= n_x; ++x)
        if (st[x] == x && !match[x]) pa[x] = 0, s[x] = 0,
            q_push(x);
    if (q.empty()) return false;
    for (;;) {
        while (q.size()) {
            int u = q.front(); q.pop();
            if (s[st[u]] == 1) continue;
            for (int v = 1; v <= n; ++v)
                if (g[u][v].w > 0 && st[u] != st[v]) {
                    if (e_delta(g[u][v]) == 0) {
                        if (on_found_edge(g[u][v])) return true
                            ;
                    } else update_slack(u, st[v]);
                }
        }
        int d = INF;
        for (int b = n + 1; b <= n_x; ++b)
            if (st[b] == b && s[b] == 1) d = min(d, lab[b] / 2)
                ;
        for (int x = 1; x <= n_x; ++x)
            if (st[x] == x && slack[x]) {
                if (s[x] == -1) d = min(d, e_delta(g[slack[x]][
                    x]));
                else if (s[x] == 0) d = min(d, e_delta(g[slack[
                    x]][x]) / 2);
            }
        for (int u = 1; u <= n; ++u) {
            if (s[st[u]] == 0) {
                if (lab[u] <= d) return 0;
                lab[u] -= d;
            } else if (s[st[u]] == 1) lab[u] += d;
        }
        for (int b = n + 1; b <= n_x; ++b)
            if (st[b] == b) {
                if (s[st[b]] == 0) lab[b] += d * 2;
                else if (s[st[b]] == 1) lab[b] -= d * 2;
            }
        q = queue<int>();
        for (int x = 1; x <= n_x; ++x)
            if (st[x] == x && slack[x] && st[slack[x]] != x &&
                e_delta(g[slack[x]][x]) == 0)
                if (on_found_edge(g[slack[x]][x])) return true;
        for (int b = n + 1; b <= n_x; ++b)
            if (st[b] == b && s[b] == 1 && lab[b] == 0)
                expand_blossom(b);
    }
    return false;
}

pair<long long, int> _solve() {
    memset(match + 1, 0, sizeof(int) * n);
    n_x = n;
    int n_matches = 0;
    long long tot_weight = 0;
    for (int u = 0; u <= n; ++u) st[u] = u, flo[u].clear();
    int w_max = 0;
    for (int u = 1; u <= n; ++u)
        for (int v = 1; v <= n; ++v) {
            flo_from[u][v] = (u == v ? u : 0);
            w_max = max(w_max, g[u][v].w);
        }
    for (int u = 1; u <= n; ++u) lab[u] = w_max;
    while (matching()) ++n_matches;
    for (int u = 1; u <= n; ++u)
        if (match[u] && match[u] < u) tot_weight += g[u][match[
            u]].w;
    return make_pair(tot_weight, n_matches);
}

void add_edge(int ui, int vi, int wi) {
    g[ui][vi].w = g[vi][ui].w = wi;
}

void init(int _n) {
    n = _n;
    for (int u = 1; u <= n; ++u) {
        for (int v = 1; v <= n; ++v) g[u][v] = Edge(u, v, 0);
    }
}
```

2.4 DFS algorithms

2sat.h

Description: Every variable x is encoded to 2i, $!x$ is 2i+1. n of TwoSAT means number of variables.

Usage:
```
TwoSat g(number of vars);
g.addCNF(x, y); // x or y
g.atMostOne({ a, b, ... });
auto ret = g.solve(void); if impossible empty
```

Time: $\mathcal{O}(V+E)$, note that sort in atMostOne function. 10^5 simple cnf clauses 56ms.

d41d8c, 94 lines

```
struct TwoSAT {
    struct SCC {
        int n;
        vector<bool> chk;
        vector<vector<int>> E, F;
        SCC() {}

        void dfs(int x, vector<vector<int>> &E, vector<int> &st
            ) {
            if(chk[x]) return;
            chk[x] = true;
            for(auto i : E[x]) dfs(i, E, st);
            st.push_back(x);
        }

        void init(vector<vector<int>> &E) {
            n = E.size();
            this->E = E;
            F.resize(n);
            chk.resize(n, false);
            for(int i = 0; i < n; i++)
                for(auto j : E[i]) F[j].push_back(i);
        }

        vector<vector<int>> getSCC() {
            vector<int> st;
            fill(chk.begin(), chk.end(), false);
            for(int i = 0; i < n; i++) dfs(i, E, st);
            reverse(st.begin(), st.end());
            fill(chk.begin(), chk.end(), false);
            vector<vector<int>> scc;
            for(int i = 0; i < n; i++) {
                if(chk[st[i]]) continue;
                vector<int> T;
                dfs(st[i], F, T);
                scc.push_back(T);
            }
            return scc;
        }
    };

    int n;
    vector<vector<int>> adj;
    TwoSAT(int n): n(n) {
        adj.resize(2*n);
    }

    int new_node() {
        adj.push_back(vector<int>());
        adj.push_back(vector<int>());
        return n++;
    }

    void add_edge(int a, int b) {
        adj[a].push_back(b);
    }

    void add_cnf(int a, int b) {
        add_edge(a^1, b);
        add_edge(b^1, a);
    }

    // arr elements need to be unique
    // Add n dummy variable, 3n-2 edges
    // yi = x1 | x2 | ... | xi, xi->yi, yi->y(i+1), yi->!x(i+1)
    void at_most_one(vector<int> arr) {
```

```
        sort(arr.begin(), arr.end());
        assert(unique(arr.begin(), arr.end()) == arr.end());
        for (int i=0; i<arr.size(); ++i) {
            int now = new_node();
            add_cnf(arr[i]^1, 2*now);
            if (i == 0) continue;
            add_cnf(2*(now-1)+1, 2*now);
            add_cnf(2*(now-1)+1, arr[i]^1);
        }
    }

    vector<int> solve() {
        SCC g;
        g.init(adj);
        auto scc = g.getSCC();

        vector<int> rev(2*n, -1);
        for (int i=0; i<scc.size(); ++i) {
            for (int x: scc[i]) rev[x] = i;
        }
        for (int i=0; i<n; ++i) {
            if (rev[2*i] == rev[2*i+1]) return vector<int>();
        }

        vector<int> ret(n);
        for (int i=0; i<n; ++i) ret[i] = (rev[2*i] > rev[2*i
            +1]);
        return ret;
    }
};
```

2.5 Coloring

EdgeColoring.h

Description: Given a simple, undirected graph with max degree D, computes a $(D+1)$-coloring of the edges such that no neighboring edges share a color.
Usage: `1-base index. Vizing g; g.clear(V); g.solve(edges, V); answer saved in G.`
Time: $\mathcal{O}(VE)$, $\sum VE = 1.1 \times 10^6$ in 24ms.

d41d8c, 60 lines

```
const int MAX_N = 444 + 1;
struct Vizing { // returns edge coloring in adjacent matrix G.
    1 - based
    int C[MAX_N][MAX_N], G[MAX_N][MAX_N];

    void clear(int n){
        for (int i=0; i<=n; i++){
            for (int j=0; j<=n; j++) C[i][j] = G[i][j] = 0;
        }
    }

    void solve(vector<pii> &E, int n){
        int X[MAX_N] = {}, a;

        auto update = [&](int u) {
            for (X[u] = 1; C[u][X[u]]; X[u]++);
        };

        auto color = [&](int u, int v, int c) {
            int p = G[u][v];
            G[u][v] = G[v][u] = c;
            C[u][c] = v;
            C[v][c] = u;
            C[u][p] = C[v][p] = 0;
            if (p) X[u] = X[v] = p;
            else update(u), update(v);
            return p;
        };
        auto flip = [&](int u, int c1, int c2){
            int p = C[u][c1]; swap(C[u][c1], C[u][c2]);
            if (p) G[u][p] = G[p][u] = c2;
            if (!C[u][c1]) X[u] = c1;
            if (!C[u][c2]) X[u] = c2;
            return p;
        };

        for (int i=1; i <= n; i++) X[i] = 1;
        for (int t=0; t<E.size(); ++t) {
            auto[u, v0] = E[t];
            int v = v0, c0 = X[u], c=c0, d;
            vector<pii> L;
            int vst[MAX_N] = {};
            while (!G[u][v0]) {
                L.emplace_back(v, d = X[v]);
                if(!C[v][c]) for(a = (int)L.size()-1; a >= 0; a
                    --) c = color(u, L[a].first, c);
                else if (!C[u][d]) for(a=(int)L.size()-1;a>=0;a
                    --) color(u,L[a].first,L[a].second);
                else if (vst[d]) break;
                else vst[d] = 1, v = C[u][d];
            }

            if(!G[u][v0]) {
                for (; v; v = flip(v, c, d), swap(c, d));
                if(C[u][c0]){
                    for(a = (int)L.size()-2; a >= 0 && L[a].
                        second != c; a--);
                    for(; a >= 0; a--) color(u, L[a].first, L[a
                        ].second);
                } else t--;
            }
        }
    }
};
```

2.6 Heuristics

2.7 Trees

DirectedMST.h

Description: Directed MST for given root node. If no MST exists, returns -1.
Usage: `0-base index. Vertex is 0 to n-1. typedef ll cost_t.`
Time: $\mathcal{O}(E \log V)$, $V = E = 2 \times 10^5$ in 90ms at yosupo.

d41d8c, 87 lines

```
struct Edge{
    int s, e; cost_t x;
    Edge() = default;
    Edge(int s, int e, cost_t x) : s(s), e(e), x(x) {}
    bool operator < (const Edge &t) const { return x < t.x; }
};
struct UnionFind{
    vector<int> P, S;
    vector<pair<int, int>> stk;
    UnionFind(int n) : P(n), S(n, 1) { iota(P.begin(), P.end(),
        0); }
    int find(int v) const { return v == P[v] ? v : find(P[v]);
        }
    int time() const { return stk.size(); }
    void rollback(int t){
        while(stk.size() > t){
            auto [u,v] = stk.back(); stk.pop_back();
            P[u] = u; S[v] -= S[u];
        }
    }
    bool merge(int u, int v){
        u = find(u); v = find(v);
        if(u == v) return false;
        if(S[u] > S[v]) swap(u, v);
        stk.emplace_back(u, v);
        S[v] += S[u]; P[u] = v;
        return true;
    }
};
struct Node{
    Edge key;
    Node *l, *r;
    cost_t lz;
    Node() : Node(Edge()) {}
    Node(const Edge &edge) : key(edge), l(nullptr), r(nullptr),
        lz(0) {}
    void push(){
        key.x += lz;
        if(l) l->lz += lz;
        if(r) r->lz += lz;
        lz = 0;
    }
    Edge top(){ push(); return key; }
};
Node* merge(Node *a, Node *b){
    if(!a || !b) return a ? a : b;
    a->push(); b->push();
    if(b->key < a->key) swap(a, b);
    swap(a->l, (a->r = merge(b, a->r)));
    return a;
}
void pop(Node* &a){ a->push(); a = merge(a->l, a->r); }

// 0-based
pair<cost_t, vector<int>> DirectMST(int n, int rt, vector<Edge>
      &edges){
    vector<Node*> heap(n);
    UnionFind uf(n);
    for(const auto &i : edges) heap[i.e] = merge(heap[i.e], new
        Node(i));
    cost_t res = 0;
    vector<int> seen(n, -1), path(n), par(n);
    seen[rt] = rt;
    vector<Edge> Q(n), in(n, {-1,-1, 0}), comp;
    deque<tuple<int, int, vector<Edge>>> cyc;
    for(int s=0; s<n; s++){
        int u = s, qi = 0, w;
        while(seen[u] < 0){
            if(!heap[u]) return {-1, {}};
            Edge e = heap[u]->top();
            heap[u]->lz -= e.x; pop(heap[u]);
            Q[qi] = e; path[qi++] = u; seen[u] = s;
            res += e.x; u = uf.find(e.s);
            if(seen[u] == s){ // found cycle, contract
                Node* nd = 0;
                int end = qi, time = uf.time();
                do nd = merge(nd, heap[w = path[--qi]]); while(
                    uf.merge(u, w));
                u = uf.find(u); heap[u] = nd; seen[u] = -1;
                cyc.emplace_front(u, time, vector<Edge>{&Q[qi],
                    &Q[end]});
            }
        }
        for(int i=0; i<qi; i++) in[uf.find(Q[i].e)] = Q[i];
    }
    for(auto& [u,t,comp] : cyc){
        uf.rollback(t);
        Edge inEdge = in[u];
        for (auto& e : comp) in[uf.find(e.e)] = e;
        in[uf.find(inEdge.e)] = inEdge;
    }
    for(int i=0; i<n; i++) par[i] = in[i].s;
```

```
    return {res, par};
}
```

ManhattanMST.h

Description: Given 2d points, find MST with taxi distance.
Usage: `0-base index internally. taxiMST(pts); Returns mst's tree edges with (length, a, b); Note that union-find need return value.`
Time: $\mathcal{O}(N \log N)$, $N = 2 \times 10^5$ in 363ms at yosupo.

d41d8c, 26 lines

```
struct point { ll x, y; };

vector<tuple<ll, int, int>> taxiMST(vector<point> a){
    int n = a.size();
    vector<int> ind(n);
    iota(ind.begin(), ind.end(), 0);
    vector<tuple<ll, int, int>> edge;
    for(int k=0; k<4; k++){
        sort(ind.begin(), ind.end(), [&](int i,int j){return a[
            i].x-a[j].x < a[j].y-a[i].y;});
        map<ll, int> mp;
        for(auto i: ind){
            for(auto it=mp.lower_bound(-a[i].y); it!=mp.end();
                it=mp.erase(it)){
                int j = it->second; point d = {a[i].x-a[j].x, a
                    [i].y-a[j].y};
                if(d.y > d.x) break;
                edge.push_back({d.x + d.y, i, j});
            }
            mp.insert({-a[i].y, i});
        }
        for(auto &p: a) if(k & 1) p.x = -p.x; else swap(p.x, p.
            y);
    }
    sort(edge.begin(), edge.end());
    DisjointSet dsu(n);
    vector<tuple<ll, int, int>> res;
    for(auto [x, i, j]: edge) if(dsu.merge(i, j)) res.push_back
        ({x, i, j});
    return res;
}
```

2.8 Math

2.8.1 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).

2.8.2 Erdős–Gallai theorem

A simple graph with node degrees $d_1 \geq \cdots \geq d_n$ exists iff $d_1 + \cdots + d_n$ is even and for every $k = 1 \ldots n$,

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k).$$

Contest (3)

template.cpp

37 lines

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
typedef pair<int, pii> piii;
typedef pair<ll, ll> pll;
typedef pair<ll, pll> plll;
#define fi first
#define se second
const int INF = 1e9+1;
const int P = 1000000007;
const ll LLINF = (ll)1e18+1;
template <typename T>
ostream& operator<<(ostream& os, const vector<T>& v) { for(auto
    i : v) os << i << " "; os << "\n"; return os; }
template <typename T1, typename T2>
ostream& operator<<(ostream& os, const pair<T1, T2>& p) { os <<
    p.fi << " " << p.se; return os; }
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());
#define rnd(x, y) uniform_int_distribution<int>(x, y)(rng)

ll mod(ll a, ll b) { return ((a%b) + b) % b; }
ll ext_gcd(ll a, ll b, ll &x, ll &y) {
    ll g = a; x = 1, y = 0;
    if(b) g = ext_gcd(b, a % b, y, x), y -= a / b * x;
    return g;
}
ll inv(ll a, ll m) {
    ll x, y; ll g = ext_gcd(a, m, x, y);
    if(g > 1) return -1;
    return mod(x, m);
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    return 0;
}
```

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 data structures 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 teammate.
Ask the teammate 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 teammate 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 teammates think about your algorithm?

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

Data structures (4)

LazySegmentTree.h

Description: 0-index, [l, r] interval
Usage: `SegmentTree seg(n); seg.query(l, r); seg.update(l, r, val);`

d41d8c, 64 lines

```
struct SegmentTree {
    int n, h;
    vector<int> arr;
    vector<int> lazy;
    SegmentTree(int _n) : n(_n) {
        h = Log2(n);
        n = 1 << h;
        arr.resize(2*n, 0);
        lazy.resize(2*n, 0);
    }

    void update(int l, int r, int c) {
        l += n, r += n;
        for (int i=h; i>=1; --i) {
            if (l >> i << i != l) push(l >> i);
            if ((r+1) >> i << i != (r+1)) push(r >> i);
        }
        for (int L=l, R=r; L<=R; L/=2, R/=2) {
            if (L & 1) apply(L++, c);
            if (~R & 1) apply(R--, c);
        }
        for (int i=1; i<=h; ++i) {
            if (l >> i << i != l) pull(l >> i);
            if ((r+1) >> i << i != (r+1)) pull(r >> i);
        }
    }

    int query(int l, int r) {
        l += n, r += n;
        for (int i=h; i>=1; --i) {
            if (l >> i << i != l) push(l >> i);
            if ((r+1) >> i << i != (r+1)) push(r >> i);
        }
```

```
        int ret = 0;
        for (; l <= r; l/=2, r/=2) {
            if (l & 1) ret = max(ret, arr[l++]);
            if (~r & 1) ret = max(ret, arr[r--]);
        }
        return ret;
    }

    void push(int x) {
        if (lazy[x] != 0) {
            apply(2*x, lazy[x]);
            apply(2*x+1, lazy[x]);
            lazy[x] = 0;
        }
    }

    void apply(int x, int c) {
        arr[x] = max(arr[x], c);
        if (x < n) lazy[x] = c;
    }

    void pull(int x) {
        arr[x] = max(arr[2*x], arr[2*x+1]);
    }

    static int Log2(int x){
        int ret = 0;
        while (x > (1 << ret)) ret++;
        return ret;
    }
};
```

ConvexHullTrick.h

Description: Max query, call init() before use.

d41d8c, 55 lines

```
struct Line{
  ll a, b, c; // y = ax + b, c = line index
  Line(ll a, ll b, ll c) : a(a), b(b), c(c) {}
  ll f(ll x){ return a * x + b; }
};
vector<Line> v; int pv;
void init(){ v.clear(); pv = 0; }
int chk(const Line &a, const Line &b, const Line &c) const {
  return (__int128_t)(a.b - b.b) * (b.a - c.a) <=
  (__int128_t)(c.b - b.b) * (b.a - a.a);
}
void insert(Line l){
  if(v.size() > pv && v.back().a == l.a){
    if(l.b < v.back().b) l = v.back(); v.pop_back();
  }
  while(v.size() >= pv+2 && chk(v[v.size()-2], v.back(), l))
  v.pop_back();
  v.push_back(l);
}
p query(ll x){ // if min query, then v[pv].f(x) >= v[pv+1].f(x)
  while(pv+1 < v.size() && v[pv].f(x) <= v[pv+1].f(x)) pv++;
  return {v[pv].f(x), v[pv].c};
}

// Container where you can add lines of the form kx+m, and
     query maximum values at points x.
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static 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()) return x->p = inf, 0;
    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());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

FenwickTree.h

Description: 0-indexed. (1-index for internal bit trick)
Usage: `FenwickTree fen(n); fen.add(x, val); fen.sum(x);`

d41d8c, 12 lines

```
struct FenwickTree {
    vector<int> tree;
    FenwickTree(int size) { tree.resize(size+1, 0); }
    int sum(int pos) {
        int ret = 0;
        for (int i=pos+1; i>0; i &= (i-1)) ret += tree[i];
        return ret;
    }
    void add(int pos, int val) {
        for (int i=pos+1; i<tree.size(); i+=(i & -i)) tree[i]
            += val;
    }
};
```

HLD.h

d41d8c, 54 lines

```
class HLD {
private:
    vector<vector<int>> adj;
    vector<int> in, sz, par, top, depth;
    void traverse1(int u) {
        sz[u] = 1;
        for (int &v: adj[u]) {
            adj[v].erase(find(adj[v].begin(), adj[v].end(), u))
                ;
            depth[v] = depth[u] + 1;
            traverse1(v);
            par[v] = u;
            sz[u] += sz[v];
            if (sz[v] > sz[adj[u][0]]) swap(v, adj[u][0]);
        }
    }
    void traverse2(int u) {
        static int n = 0;
        in[u] = n++;
        for (int v: adj[u]) {
            top[v] = (v == adj[u][0] ? top[u] : v);
            traverse2(v);
        }
    }
public:
    void link(int u, int v) { // u and v is 1-based
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    void init() { // have to call after linking
        top[1] = 1;
        traverse1(1);
        traverse2(1);
    }
    // u is 1-based and returns dfs-order [s, e) 0-based index
    pii subtree(int u) {
        return {in[u], in[u] + sz[u]};
    }
    // u and v is 1-based and returns array of dfs-order [s, e)
         0-based index
    vector<pii> path(int u, int v) {
        vector<pii> res;
        while (top[u] != top[v]) {
            if (depth[top[u]] < depth[top[v]]) swap(u, v);
            res.emplace_back(in[top[u]], in[u] + 1);
            u = par[top[u]];
        }
        res.emplace_back(min(in[u], in[v]), max(in[u], in[v]) +
            1);
        return res;
    }
    HLD(int n) { // n is number of vertexes
        adj.resize(n+1); depth.resize(n+1);
        in.resize(n+1); sz.resize(n+1);
        par.resize(n+1); top.resize(n+1);
    }
};
```

PBDS.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: $\mathcal{O}(\log N)$

d41d8c, 13 lines

```
#include <bits/extc++.h>
using namespace __gnu_pbds;
template<class T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
                 tree_order_statistics_node_update>;
int main() {
    ordered_set<int> X;
    for (int i=1; i<10; i+=2) X.insert(i); // 1 3 5 7 9
    cout << *X.find_by_order(2) << endl; // 5
    cout << X.order_of_key(6) << endl; // 3
    cout << X.order_of_key(7) << endl; // 3
    X.erase(3);
}
```

Rope.h

Description: 1 x y: Move SxSx+1...Sy to front of string. $(0 \le x \le y < N)$ 2 x y: Move SxSx+1...Sy to back of string. $(0 \le x \le y < N)$ 3 x: Print Sx. $(0 \le x < N)$ cf. rope.erase(index, count) : erase [index, index+count)

<ext/rope> d41d8c, 23 lines

```
using namespace __gnu_cxx;
int main() {
    string s; cin >> s;
    rope<char> R;
    R.append(s.c_str());
    int q; cin >> q;
    while(q--) {
        int t, x, y; cin >> t;
        switch(t) {
            case 1:
                cin >> x >> y; y++;
                R = R.substr(x, y-x) + R.substr(0, x) + R.
                    substr(y, s.size());
```

```
                break;
            case 2:
                cin >> x >> y; y++;
                R = R.substr(0, x) + R.substr(y, s.size()) + R.
                    substr(x, y-x);
                break;
            default:
                cin >> x;
                cout << R[x] << "\n";
        }
    }
}
```

PersistentSegmentTree.h

Description: Point update (addition), range sum query
Usage: Unknown, but just declare sufficient size. You should achieve root number manually after every query/update. d41d8c, 69 lines

```
struct PersistentSegmentTree {
    int size;
    int last_root;
    vector<ll> tree, l, r;

    PersistentSegmentTree(int _size) {
        size = _size;
        init(0, size-1);
        last_root = 0;
    }

    void add_node() {
        tree.push_back(0);
        l.push_back(-1);
        r.push_back(-1);
    }

    int init(int nl, int nr) {
        int n = tree.size();
        add_node();
        if (nl == nr) {
            tree[n] = 0;
            return n;
        }
        int mid = (nl + nr) / 2;
        l[n] = init(nl, mid);
        r[n] = init(mid+1, nr);
        return n;
    }

    void update(int ori, int pos, int val, int nl, int nr) {
        int n = tree.size();
        add_node();
        if (nl == nr) {
            tree[n] = tree[ori] + val;
            return;
        }

        int mid = (nl + nr) / 2;
        if (pos <= mid) {
            l[n] = tree.size();
            r[n] = r[ori];
            update(l[ori], pos, val, nl, mid);
        } else {
            l[n] = l[ori];
            r[n] = tree.size();
            update(r[ori], pos, val, mid+1, nr);
        }
        tree[n] = tree[l[n]] + tree[r[n]];
    }

    void update(int pos, int val) {
        int new_root = tree.size();
        update(last_root, pos, val, 0, size-1);
        last_root = new_root;
    }

    ll query(int a, int b, int n, int nl, int nr) {
        if (n == -1) return 0;
        if (b < nl || nr < a) return 0;
        if (a <= nl && nr <= b) return tree[n];
        int mid = (nl + nr) / 2;
        return query(a, b, l[n], nl, mid) + query(a, b, r[n],
            mid+1, nr);
    }

    ll query(int x, int root) {
        return query(0, x, root, 0, size-1);
    }
};
```

Geometry (5)

5.1 Analytic Geometry

Area $A = \sqrt{p(p-a)(p-b)(p-c)}$ when $p = (a+b+c)/2$
Circumscribed circle $R = abc/4A$, inscribed circle $r = A/p$
Middle line length $m_a = \sqrt{2b^2 + 2c^2 - a^2}/2$
Bisector line length $s_a = \sqrt{bc[1 - (\frac{a}{b+c})^2]}$

Name	α	β	γ	
R	$a^2\mathcal{A}$	$b^2\mathcal{B}$	$c^2\mathcal{C}$	$\mathcal{A} = b^2 + c^2 - a^2$
r	a	b	c	$\mathcal{B} = a^2 + c^2 - b^2$
G	1	1	1	$\mathcal{C} = a^2 + b^2 - c^2$
H	$\mathcal{BC}$	$\mathcal{CA}$	$\mathcal{AB}$	
Excircle(A)	$-a$	b	c	

$HG : GO = 1 : 2$. H of triangle made by middle point on arc of circumscribed circle is equal to inscribed circle center of original triangle.

5.2 Geometric primitives

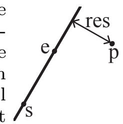
Point.h

Description: Maybe you can improvise it. Caution to overflow such as outer product. d41d8c, 28 lines

```
template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
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)); }
    friend istream& operator>>(istream& is, Point& p) { is >> p
        .x >> p.y; return is; }
    friend ostream& operator<<(ostream& os, P p) { return os <<
        "(" << p.x << "," << p.y << ")"; }
};
```

lineDistance.h

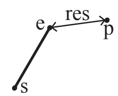
Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate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



"Point.h" d41d8c, 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

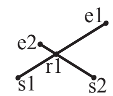


Description:
Returns the shortest distance between point p and the line segment from point s to e.
Usage: Point<double> a, b(2,2), p(1,1);
bool onSegment = segDist(a,b,p) < 1e-10;
"Point.h" d41d8c, 6 lines

```
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 intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> 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.



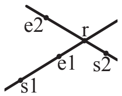
Usage: vector<P> inter = segInter(s1,e1,s2,e2);
if (sz(inter)==1)
cout << "segments intersect at " << inter[0] << endl;
"Point.h", "OnSegment.h" d41d8c, 13 lines

```
template<class P> vector<P> segInter(P a, P b, P c, P d) {
  auto oa = c.cross(d, a), ob = c.cross(d, b),
       oc = a.cross(b, c), od = a.cross(b, d);
  // Checks if intersection is single non-endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
    return {(a * ob - b * oa) / (ob - oa)};
  set<P> s;
  if (onSegment(c, d, a)) s.insert(a);
  if (onSegment(c, d, b)) s.insert(b);
```

```
  if (onSegment(a, b, c)) s.insert(c);
  if (onSegment(a, b, d)) s.insert(d);
  return {all(s)};
}
```

lineIntersection.h

Description:



If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> 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 ll.
Usage: `auto res = lineInter(s1,e1,s2,e2);`
`if (res.first == 1)`
`cout << "intersection point at " << res.second << endl;`

"Point.h" d41d8c, 8 lines

```
template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0), P(0, 0)};
  auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
  return {1, (s1 * p + e1 * q) / d};
}
```

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 ⇔ 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" d41d8c, 9 lines

```
template<class P>
int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }

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

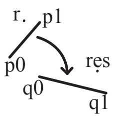
Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.

"Point.h" d41d8c, 3 lines

```
template<class P> bool onSegment(P s, P e, P p) {
  return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 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" d41d8c, 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. Sometimes also represents points or vectors.
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`

d41d8c, 35 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 b) const { return {x-b.x, y-b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
  Angle t180() const { return {-x, -y, t + half()}; }
  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.half(), a.y * (ll)b.x) <
         make_tuple(b.t, b.half(), a.x * (ll)b.y);
}

// 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) { // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a)};
}
```

5.3 Circles

CircleIntersection.h

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

"Point.h" d41d8c, 11 lines

```
typedef Point<double> P;
bool circleInter(P a,P b,double r1,double r2,pair<P, P>* out) {
  if (a == b) { assert(r1 != r2); return false; }
  P vec = b - a;
  double d2 = vec.dist2(), sum = r1+r2, dif = r1-r2,
         p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 = r1*r1 - p*p*d2;
  if (sum*sum < d2 || dif*dif > d2) return false;
  P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

CircleTangents.h

Description: Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

"Point.h" d41d8c, 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1, double r1, P c2, double r2) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

CirclePolygonIntersection.h

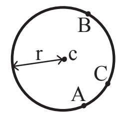
Description: Returns the area of the intersection of a circle with a ccw polygon.
Time: $\mathcal{O}(n)$

"../../content/geometry/Point.h" d41d8c, 19 lines

```
typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(P c, double r, vector<P> ps) {
  auto tri = [&](P p, P q) {
    auto r2 = r * r / 2;
    P d = q - p;
    auto a = d.dot(p)/d.dist2(), b = (p.dist2()-r*r)/d.dist2();
    auto det = a * a - b;
    if (det <= 0) return arg(p, q) * r2;
    auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det));
    if (t < 0 || 1 <= s) return arg(p, q) * r2;
    P u = p + d * s, v = p + d * t;
    return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;
  };
  auto sum = 0.0;
  rep(i,0,sz(ps))
    sum += tri(ps[i] - c, ps[(i + 1) % sz(ps)] - c);
  return sum;
}
```

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" d41d8c, 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" d41d8c, 17 lines

```
pair<P, double> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  rep(i,0,sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    rep(j,0,i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k,0,j) if ((o - ps[k]).dist() > r * EPS) {
        o = ccCenter(ps[i], ps[j], ps[k]);
        r = (o - ps[i]).dist();
      }
    }
  }
  return {o, r};
}
```

5.4 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.
Usage: `vector<P> v = {P{4,4}, P{1,2}, P{2,1}};`
`bool in = inPolygon(v, P{3, 3}, false);`
Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h" d41d8c, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = sz(p);
  rep(i,0,n) {
    P q = p[(i + 1) % n];
    if (onSegment(p[i], q, a)) return !strict;
    //or: if (segDist(p[i], q, a) <= eps) return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) > 0;
  }
  return cnt;
}
```

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" d41d8c, 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;
}
```

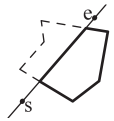
PolygonCenter.h

Description: Returns the center of mass for a polygon.
Time: $\mathcal{O}(n)$

"Point.h" d41d8c, 9 lines

```
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = sz(v) - 1; i < sz(v); j = i++) {
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
}
```

PolygonCut.h

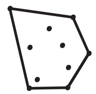


Description:
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.
Usage: `vector<P> p = ...;`
`p = polygonCut(p, P(0,0), P(1,0));`

"Point.h", "lineIntersection.h" d41d8c, 13 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.push_back(lineInter(s, e, cur, prev).second);
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

ConvexHull.h



Description:
Returns a vector of the points 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.
Time: $\mathcal{O}(n \log n)$

"Point.h" d41d8c, 13 lines

```
typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
  if (sz(pts) <= 1) return pts;
  sort(all(pts));
  vector<P> h(sz(pts)+1);
  int s = 0, t = 0;
  for (int it = 2; it--; s = --t, reverse(all(pts)))
    for (P p : pts) {
      while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t--;
      h[t++] = p;
    }
  return {h.begin(), h.begin() + t - (t == 2 && h[0] == h[1])};
}
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).
Time: $\mathcal{O}(n)$

"Point.h" d41d8c, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  int n = sz(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
  rep(i,0,j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}});
      if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >= 0)
        break;
    }
  return res.second;
}
```

PointInsideHull.h

Description: Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.
Time: $\mathcal{O}(\log N)$

"Point.h", "sideOf.h", "OnSegment.h" d41d8c, 14 lines

```
typedef Point<ll> P;

bool inHull(const vector<P>& l, P p, bool strict = true) {
  int a = 1, b = sz(l) - 1, r = !strict;
  if (sz(l) < 3) return r && onSegment(l[0], l.back(), p);
  if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
  if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<= -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
}
```

LineHullIntersection.h

Description: Line-convex polygon intersection. The polygon must be ccw and have no collinear points. lineHull(line, poly) 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. `extrVertex` returns the point of a hull with the max projection onto a line.
Time: $\mathcal{O}(\log n)$

"Point.h" d41d8c, 39 lines

```
#define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
template <class P> int extrVertex(vector<P>& poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
    (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
  }
  return lo;
}

#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P>& poly) {
  int endA = extrVertex(poly, (a - b).perp());
  int endB = extrVertex(poly, (b - a).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  rep(i,0,2) {
    int lo = endB, hi = endA, n = sz(poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
      (cmpL(m) == cmpL(endB) ? lo : hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
}
```

5.5 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.
Time: $\mathcal{O}(n \log n)$

"Point.h" d41d8c, 17 lines

```
typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
  assert(sz(v) > 1);
  set<P> S;
  sort(all(v), [](P a, P b) { return a.y < b.y; });
  pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
  int j = 0;
  for (P p : v) {
    P d{1 + (ll)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2(), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
}
```

kdTree.h

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

"Point.h" d41d8c, 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" d41d8c, 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;
}
```

5.6 3D

PolyhedronVolume.h

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

d41d8c, 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.

d41d8c, 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: $\mathcal{O}\left(n^2\right)$

"Point3D.h" d41d8c, 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.

d41d8c, 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);
}
```

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 contiguous subvector sum
  Invariants
  Huffman encoding
Graph theory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  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
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges and 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 programming
  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
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little 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 polynomial 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
```