# ここのところのリファレンス

Last Update: 2008/09/12

by kkntkr / unknown

にょろーんの精神でがんばってください — 桜庭 俊

## 0 基礎

### 0.1 いつものマクロ

```
#define REP(i,n) for(int i = 0; i < (int)(n); i
    ++)
#define FOR(i,c) for(__typeof((c).begin()) i = (
    c).begin(); i != (c).end(); ++i)
#define ALLOF(c) (c).begin(), (c).end()
```

### 0.2 間違ったかな？と思ったら

- **とりあえず深呼吸。**
- **これフローじゃね？**
- **これ二分探索じゃね？**
- **これ凸じゃね？**
- **最小/最大/特異ケースのテスト**
- -Wall, -Wextra はつけているか？
- 問題の条件をよみなおせ！
- 入力の形式はあっているか？
  - 入力ファイルを修正してそのままにしていないか
  - スペースを含む文字列
  - 0-origin, 1-origin
  - (i,j) <-> (x,y) 変換をはじめとする座標変換
  - 改行コード www
- 出力の形式はあっているか？
  - スペルミス
  - -0.000
  - 空行のはさみ方
  - デバッグ出力の削りのこし
  - 解が無い時
- オーバーフロー
  - 答えが int に収まらない
  - numeric_limits<int>::max() * 2
  - bool 型の変数に int/double を代入、関数の返り値型が bool (特に、DP/メモの型を後で変更した場合)
- ポインタの進め忘れ
- 参照/イテレータの無効化
- 実数
  - EPS を使わずに比較していないか
  - 問題に合った EPS の設定
  - NaN (/, sqrt, asin, acos, etc...)
- switch
  - break し忘れていないか
  - default に assert(false)
- ライブラリ関数
  - 全順序関係でないような operator< を使ったソート
  - accumulate/inner_product の初期値は必ず型をキャストで明示すること
- 配列
  - サイズは十分か？
  - 初期化を忘れていないか？
  - 番兵のせいでサイズ・インデックス・初期化が間違っていないか？
  - 添字を間違っていないか？ (j と書くべきところに i、など)
- コピペ
  - コピペ修正ミスは無いか？ (y += dx みたいな)
- 考え方
  - 問題を分解したときに独立にとけないこともあるので柔軟に考えるといいよ

# 1 幾何

## 1.1 基礎

```
typedef complex<double> P;
struct L { P pos, dir; };
typedef vector<P> G;
struct C { P p; double r; };

inline double inp(const P& a, const P& b) {
  return (conj(a)*b).real();
}
inline double outp(const P& a, const P& b) {
  return (conj(a)*b).imag();
}

inline int ccw(const P& p, const P& r, const P&
    s) {
  P a(r-p), b(s-p);
  int sgn = signum(outp(a, b));
  if (sgn != 0)
    return sgn;
  if (a.real()*b.real() < -EPS || a.imag()*b.
      imag() < -EPS)
    return -1;
  if (norm(a) < norm(b) - EPS)
    return 1;
  return 0;
}

// ベクトルp をベクトル b に射影したベクトルを計算する
inline P proj(const P& p, const P& b) {
  return b*inp(p,b)/norm(b);
}
// 点p から直線 l に引いた垂線の足となる点を計算する
inline P perf(const L& l, const P& p) {
  L m = {l.pos - p, l.dir};
  return (p + (m.pos - proj(m.pos, m.dir)));
}
// 線分s を直線 b に射影した線分を計算する
inline L proj(const L& s, const L& b) {
   return (L){perf(b, s.pos), proj(s.dir, b.dir
       )};
}
```

## 1.2 面積・体積

### 1.2.1 多角形の面積

```
double area(G& g) {
  int n = g.size();
  double s = 0.0;
  for(int i = 0; i < n; i++) {
    int j = (i+1)%n;
    s += outp(g[i], g[j])/2;
  }
  return abs(s);
}
```

### 1.2.2 円と円の交差面積

```
double cc_area(const C& c1, const C& c2) {
  double d = abs(c1.p - c2.p);
  if (c1.r + c2.r <= d + EPS) {
    return 0.0;
  } else if (d <= abs(c1.r - c2.r) + EPS) {
    double r = c1.r <? c2.r;
    return r * r * PI;
  } else {
    double rc = (d*d + c1.r*c1.r - c2.r*c2.r) /
        (2*d);
    double theta = acos(rc / c1.r);
    double phi = acos((d - rc) / c2.r);
    return c1.r*c1.r*theta + c2.r*c2.r*phi - d*c1
        .r*sin(theta);
  }
}
```

## 1.3 交差

### 1.3.1 各種交差判定

```
bool ls_intersects(const L& l, const L& s) {
  return (signum(outp(l.dir, s.pos-l.pos)) *
     signum(outp(l.dir, s.pos+s.dir-l.pos)) <=
         0);
}
bool sp_intersects(const L& s, const P& p) {
  return ( abs(s.pos - p) + abs(s.pos + s.dir -
      p) - abs(s.dir) < EPS );
}
bool ss_intersects(const L& s, const L& t) {
  return (ccw(s.pos, s.pos+s.dir, t.pos) *
     ccw(s.pos, s.pos+s.dir, t.pos+t.dir) <= 0
         &&
     ccw(t.pos, t.pos+t.dir, s.pos) *
     ccw(t.pos, t.pos+t.dir, s.pos+s.dir) <= 0);
}
```

### 1.3.2 多角形と点の包含判定

辺と点が重なれば包含。そうでなければ、$\mathrm{abs}(\sum_i \arg((g_{i+1}-p)/(g_i-p))) > 1$ なら包含。

### 1.3.3 円と円の交点

**交点が 0 個または無限個だとうまく動かない。**

```
pair<P, P> cc_cross(const C& c1, const C& c2) {
  double d = abs(c1.p - c2.p);
  double rc = (d*d + c1.r*c1.r - c2.r*c2.r) /
      (2*d);
  double rs = sqrt(c1.r*c1.r - rc*rc);
  P diff = (c2.p - c1.p) / d;
  return make_pair(c1.p + diff * P(rc, rs), c1.p
       + diff * P(rc, -rs));
}
```

### 1.3.4 円と直線の交点

```
vector<P> cl_cross(const C& c, const L& l) {
  P h = perf(l, c.p);
  double d = abs(h - c.p);
  vector<P> res;
  if(d < c.r - EPS){
  P x = l.dir / abs(l.dir) * sqrt(c.r*c.r - d*d
      );
  res.push_back(h + x);
  res.push_back(h - x);
  }else if(d < c.r + EPS){
  res.push_back(h);
  }
  return res;
}
```

### 1.3.5 凸多角形と線分の包含判定

線分と凸多角形が内部で交差しているか判定する。接しているだけの場合には交差しているとみなさない。

**多角形は凸でなければいけない。多角形は反時計回りに与えられなければならない。**

```
bool cgs_contains_exclusive(G& r, P from, P to)
    {
  int m = r.size();

  P dir0 = to - from;

  double lower = 0.0, upper = 1.0;
  REP(i, m) {
    int j = (i+1)%m;
    P a = r[i], b = r[j];
    P ofs = a - from;
    P dir1 = b - a;
    double num = (conj(ofs)*dir1).imag();
    double denom = (conj(dir0)*dir1).imag();
    if (abs(denom) < EPS) {
      if (num < EPS)
        lower = upper = 0.0;
    }
    else {
      double t = num / denom;
      if (denom > 0)
        upper <?= t;
      else
        lower >?= t;
    }
  }

  return (upper-lower > EPS);
}
```

### 1.3.6 直線と直線の交点

**直線が並行となるときに注意すること。**denom = 0 となるケース、すなわち**直線が平行となるとき**に注意すること。**片方または両方が線分のとき**に使用する場合は、`{ls,ss}_intersects()` 等と併用して、交点が valid なものか確認すること。

```
P line_cross(const L& l, const L& m) {
  double num = outp(m.dir, m.pos-l.pos);
  double denom = outp(m.dir, l.dir);
  return P(l.pos + l.dir*num/denom);
}
```

## 1.4 距離

### 1.4.1 各種距離

```
double lp_distance(const L& l, const P& p) {
  return abs(outp(l.dir, p-l.pos) / abs(l.dir));
}
double ll_distance(const L& l, const L& m) {
  return (ll_intersects(l, m) ? 0 : lp_distance(
      l, m.pos));
}
double ls_distance(const L& l, const L& s) {
  if (ls_intersects(l, s))
    return 0;
  return min(lp_distance(l, s.pos), lp_distance(
      l, s.pos+s.dir));
}
double sp_distance(const L& s, const P& p) {
  const P r = perf(s, p);
  const double pos = ((r-s.pos)/s.dir).real();
  if (-EPS <= pos && pos <= 1 + EPS)
    return abs(r - p);
  return min(abs(s.pos - p),
       abs(s.pos+s.dir - p));
}
double ss_distance(const L& s, const L& t) {
  if (ss_intersects(s, t))
    return 0;
  return (sp_distance(s, t.pos) <?
      sp_distance(s, t.pos+t.dir) <?
      sp_distance(t, s.pos) <?
      sp_distance(t, s.pos+s.dir));
}
```

## 1.5 最遠点対

**点集合は凸包であり、反時計回りに与えられること。**

```
double farthest(const vector<P>& v) {
  int n = v.size();

  int si = 0, sj = 0;
  REP(t, n) {
    if (v[t].imag() < v[si].imag())
      si = t;
    if (v[t].imag() > v[sj].imag())
      sj = t;
  }

  double res = 0;
  int i = si, j = sj;
  do {
    res >?= abs(v[i]-v[j]);
    P di = v[(i+1)%n] - v[i];
    P dj = v[(j+1)%n] - v[j];
    if (outp(di, dj) >= 0)
      j = (j+1)%n;
    else
      i = (i+1)%n;
  } while(!(i == si && j == sj));

  return res;
}
```

## 1.6 凸包

コードは最小の点数からなる凸包を求める場合。凸包の辺上にある点を全て取りたい場合は `ccw` の比較を `-EPS` に変更する。**最大点数を取るように変更した場合、一直線に並んだ点群を処理すると凸包が元の点数より大きくなることに注意。点がひとつのときに注意。**

```
bool xy_less(const P& a, const P& b) {
  if (abs(a.imag()-b.imag()) < EPS) return (a.
      real() < b.real());
  return (a.imag() < b.imag());
}
double ccw(P a, P b, P c) {
  return (conj(b-a)*(c-a)).imag();
}
```

```
template<class IN>
void walk_rightside(IN begin, IN end, vector<P>&
     v) {
  IN cur = begin;
  v.push_back(*cur++);
  vector<P>::size_type s = v.size();
  v.push_back(*cur++);
  while(cur != end) {
    if (v.size() == s || ccw(v[v.size()-2], v.
        back(), *cur) > EPS)
      v.push_back(*cur++);
    else
      v.pop_back();
  }
  v.pop_back();
}
vector<P> convex_hull(vector<P> v) {
  if (v.size() <= 1)
    return v; // EXCEPTIONAL
  sort(ALLOF(v), xy_less);
  vector<P> cv;
  walk_rightside(v.begin(), v.end(), cv);
  walk_rightside(v.rbegin(), v.rend(), cv);
  return cv;
}
```

## 1.7 凸多角形のクリッピング

直線の右側が切り取られ、左側が残される。**多角形は counterclockwise に与えられていること。**

```
G cut_polygon(const G& g, L cut) {
  int n = g.size();
  G res;

  REP(i, n) {
    P from(g[i]), to(g[(i+1)%n]);
    double p1 = (conj(cut.dir)*(from-cut.pos)).
        imag();
    double p2 = (conj(cut.dir)*(to-cut.pos)).imag
        ();
    if (p1 > -EPS) {
      res.push_back(from);
      if (p2 < -EPS && p1 > EPS)
        res.push_back(from - (to-from)*p1/(conj(
            cut.dir)*(to-from)).imag());
    }
    else if (p2 > EPS) {
      res.push_back(from - (to-from)*p1/(conj(cut
          .dir)*(to-from)).imag());
    }
  }

  return res;
}
```

## 1.8 アレンジメント

与えられた直線の集合に対し、直線同士の交点からなるグラフを作成する。線分から作成したいときは lp_intersects を sp_intersects に置き換える。ただし、端点もノードにしたい場合は適宜追加すること。**重なる直線が存在してはいけない。lp_intersects は NaN である点に対しても正しく動くこと。**

```
typedef map<P, vector<P> > PGraph;
PGraph arrange(const vector<L>& lines) {
  int m = lines.size();

  set<P> points;
  REP(j, m) REP(i, j) {
    L a = lines[i], b = lines[j];
    double r = outp(b.dir, b.pos-a.pos) / outp(b.
        dir, a.dir);
    P p = a.pos + a.dir * r;
    if (lp_intersects(a, p) && lp_intersects(b, p
        ))
      points.insert(p);
  }

  PGraph g;
  REP(k, m) {
    vector< pair<double, P> > onlines;
    FOR(it, points) if (lp_intersects(lines[k], *
        it))
      onlines.push_back(make_pair(inp(*it-lines[k
          ].pos, lines[k].dir), *it));
    sort(ALLOF(onlines));
    REP(i, (int)onlines.size()-1) {
      g[onlines[i+0].second].push_back(onlines[i
          +1].second);
      g[onlines[i+1].second].push_back(onlines[i
          +0].second);
    }
  }

  return g;
}
```

## 1.9 ダイス

```
struct die_t {
  int top, front, right, left, back, bottom;
};

#define rotate_swap(x,a,b,c,d) swap(x.a, x.b);
    swap(x.b, x.c); swap(x.c, x.d);
void rotate_right(die_t& x) { rotate_swap(x,top,
    left,bottom,right); }
void rotate_left(die_t& x) { rotate_swap(x,top,
    right,bottom,left); }
void rotate_front(die_t& x) { rotate_swap(x,top,
    back,bottom,front); }
void rotate_back(die_t& x) { rotate_swap(x,top,
    front,bottom,back); }
void rotate_cw(die_t& x) { rotate_swap(x,back,
    left,front,right); }
void rotate_ccw(die_t& x) { rotate_swap(x,back,
    right,front,left); }

void generate_all() {
  REP(i, 6) {
    REP(j, 4) {
      emit();
      rotate_cw(x);
    }
    (i%2 == 0 ? rotate_right : rotate_front)(x);
  }
}
```

## 1.10 三次元幾何

### 1.10.1 直線と直線の距離

```
double ll_distance(L& a, L& b) {
  P pa = a.from;
  P pb = b.from;
  P da = a.to - a.from;
  P db = b.to - b.from;
  double num = inp( (db * inp(da, db) - da * inp
      (db, db)) , pb - pa );
  double denom = inp(da, db) * inp(da, db) - inp
      (da, da) * inp(db, db);
  double ta;
  if (abs(denom) < EPS)
    ta = 0;
  else
    ta = num / denom;
  P p = pa + da * ta;
  return lp_distance(b, p);
}
```

# 2 グラフ

## 2.1 最短路

### 2.1.1 Dijkstra

一対全最短路問題。非負の重みのみであるときに使用する。

```
Graph g;
vector<int> trace;

Weight shortest(int start, int goal) {
  int n = g.size();
  trace.assign(n, -2); // UNREACHABLE

  priority_queue<Edge, vector<Edge>, greater<
      Edge> > q;
  q.push((Edge){-1, start, 0}); // TERMINAL

  while(!q.empty()) {
    Edge e = q.top();
    q.pop();
    if (trace[e.dest] >= 0)
      continue;
    trace[e.dest] = e.src;
    if (e.dest == goal)
      return e.weight;
    FOR(it, g[e.dest])
      if (trace[it->dest] == -2)
        q.push((Edge){it->src, it->dest, e.weight
            + it->weight});
  }
  return -1;
}

vector<int> buildRoute(int goal) {
  if (trace[goal] == -2)
    return vector<int>(); // UNREACHABLE
  vector<int> route;
  for(int i = goal; i >= 0; i = trace[i])
    route.push_back(i);
  reverse(route.begin(), route.end());
  return route;
}
```

### 2.1.2 Bellman-Ford

一対全最短路問題。負の重みがあるときに使用する。$n$ 回やっても緩和が起こるようなら negative cycle が存在する。

枝の重みが非負であるグラフにおいて、一対全最短経路を求めるアルゴリズム。負の重みを持つ枝が存在するなら Bellman-Ford 法などのアルゴリズムを用いる。**パスは非負の重みであること。**

```
Graph g;

bool shortest(int start) {
  int n = g.size();
  vector<Weight> costs(n, WEIGHT_INFTY);
  vector<int> trace(n, -2);
  costs[start] = 0;
  trace[start] = -1; // TERMINAL
  REP(k, n) REP(a, n) if (costs[a] !=
      WEIGHT_INFTY) FOR(it, g[a]) {
    int b = it->dest;
    Weight w = costs[a] + it->weight;
    if (w < costs[b]) {
      costs[b] = w;
      trace[b] = a;
    }
  }
  REP(a, n) FOR(it, g[a]) // negative
      cycle のチェック
    if (costs[a] + it->weight < costs[it->dest])
      return false;
  return true;
}
```

### 2.1.3 Warshall-Floyd

全対全最短路問題。**自身への距離を 0 とするかどうかに応じて隣接行列の対角成分を初期化すること。**道順がいるときは再帰的に。

```
Weight g[N][N];
int trace[N][N];

void shortest() {
  REP(i, n) REP(j, n)
    trace[i][j] = -1;
  REP(j, n) REP(i, n) REP(k, n) {
    if (g[i][k] > g[i][j] + g[j][k]) {
      g[i][k] = g[i][j] + g[j][k];
      trace[i][k] = j;
    }
  }
}
void buildRoute(vector<int>& route, int src, int
      dest, bool rec = false) {
  if (!rec)
    route.clear();
  int inter = trace[src][dest];
  if (inter < 0) {
    route.push_back(src);
  }
  else {
    buildRoute(route, src, inter, true);
    buildRoute(route, inter, dest, true);
  }
  if (!rec)
    route.push_back(dest);
}
```

## 2.2 最小広域木

任意の頂点集合分割について、それらの間にある枝のうち最小の重みを持つ枝を含む最小広域木が存在する。

枝を重みの順でソートして Union-Find するのが Kruskal。任意の点から始めて、最小コストで取り込める点を取っていくのが Prim。Prim は隣接行列表現を用いれば $O(V^3)$ で走る。

```
pair<Weight, Edges> prim(Graph& g) {
  int n = g.size();
  priority_queue<Edge, vector<Edge>, greater<
      Edge> > q;
  vector<bool> visited(n, false);
  Edges tree;
```

```
  Weight weight = 0;

  q.push((Edge){-1, 0, 0});
  while(!q.empty()) {
    Edge e = q.top();
    q.pop();
    if (visited[e.dest])
      continue;
    visited[e.dest] = true;
    if (e.src >= 0) {
      tree.push_back(e);
      weight += e.weight;
    }
    FOR(it, g[e.dest])
      if (!visited[it->dest])
        q.push(*it);
  }
  return make_pair(weight, tree);
}
```

```
pair<Weight, Edges> kruskal(Graph& g) {
  int n = g.size();
  vector<Edge> edges;
  REP(a, n) FOR(it, g[a])
    if (a < it->dest)
      edges.push_back(*it);
  sort(ALLOF(edges));
  UnionFind uf(n);
  Edges tree;
  Weight weight = 0;
  REP(i, edges.size()) {
    if ((int)tree.size() >= n-1)
      break;
    Edge& e = edges[i];
    if (uf.link(e.src, e.dest)) {
      tree.push_back(e);
      weight += e.weight;
    }
  }
  return make_pair(weight, tree);
}
```

## 2.3 無向オイラー路

グラフの一筆書き (オイラー路) を求めるアルゴリズム。無向グラフが辺連結なとき、奇点が存在しない場合無向オイラー閉路が存在し、奇点が 2 個だけ存在する場合無向オイラー路が存在する。**グラフ g は隣接行列で与えられること。多重辺・セルフループを許す。**

```
int g[N][N];
int adj[N][N];

void dfs(int a, vector<int>& route) {
  REP(b, N){
    if(adj[a][b]){
      adj[a][b]--;
      adj[b][a]--;
      dfs(b, route);
    }
  }
  route.push_back(a);
}

vector<int> euler(int start) {
  int odd = 0;
  int nEdges = 0;
  REP(i, N){
    int deg = accumulate(g[i], g[i] + N, 0);
    if(deg % 2 == 1)
      odd++;
    nEdges += deg;
  }
  nEdges /= 2;

  vector<int> route;
  int startdeg = accumulate(g[start], g[start] +
       N, 0);
  if(odd == 0 || (odd == 2 && startdeg % 2 ==
      1)){
    memcpy(adj, g, sizeof(g));
    dfs(start, route);
  }

  if((int)route.size() != nEdges + 1) // 非連結だった場合
    route.clear();
  reverse(route.begin(), route.end());
  return route;
}
```

## 2.4 有向オイラー路

グラフの一筆書き (オイラー路) を求めるアルゴリズム。有向グラフが辺連結なとき、全ての点において出次数と入次数が等しい場合有向オイラー閉路が存在し、出次数 1 の点と入次数 1 の点が各 1 個でそれ以外の全ての点で出次数と入次数が等しい場合オイラー路が存在する。**グラフは隣接リストで与えられること。多重辺・セルフループを許す。**

```
void dfs(Graph& g, int a, vector<int>& route) {
  while(!g[a].empty()){
    int b = g[a].back();
    g[a].pop_back();
    dfs(g, b, route);
  }
  route.push_back(a);
}

vector<int> euler(Graph& g, int start) {
  int nNodes = g.size();
  int nEdges = 0;
  vector<int> deg(nNodes, 0);
  REP(i, nNodes){
    nEdges += g[i].size();
    REP(j, g[i].size())
      deg[g[i][j]]--; // 入次数
    deg[i] += g[i].size(); // 出次数
  }

  vector<int> route;
  int nonzero = nNodes - count(deg.begin(), deg.
      end(), 0);
  if(nonzero == 0 || (nonzero == 2 && deg[start]
       == 1)) {
    Graph g_(g);
    dfs(g_, start, route);
  }

  if((int)route.size() != nEdges + 1) // 非連結だった場合
    route.clear();
  reverse(route.begin(), route.end());
  return route;
}
```

## 2.5 強連結成分分解

出力される強連結成分は DAG に縮約したときのトポロジカル順序に従っている。

```
Graph g;
vector<bool> visited;

vector< vector<int> > scc() {
  int n = g.size();
  Graph r(n); // make reversed graph
  REP(a, n) FOR(it, g[a])
    r[it->dest].push_back((Edge){it->dest, it->
        src});
  vector<int> order;
  visited.assign(n, false);
  REP(i, n) dfs(i, order);
  reverse(order.begin(), order.end());
  g.swap(r);
  vector< vector<int> > components;
  visited.assign(n, false);
  REP(i, n)
    if (!visited[order[i]])
      components.push_back(vector<int>()) ,
        dfs(order[i], components.back());
  g.swap(r);
  return components;
}
void dfs(int a, vector<int>& order) {
  if (visited[a])
    return;
  visited[a] = true;
  FOR(it, g[a]) dfs(it->dest, order);
  order.push_back(a);
}
```

## 2.6 二重頂点連結成分

点 $a$ を除去したときに $m$ 個の連結成分に分割される時、arts に $m-1$ 個の数 $a$ が代入される。bccs には、二重頂点連結成分を誘導する頂点集合の集合が代入される。**arts, bccs は空にして渡すこと。**

```
int n;
bool g[N][N];

int bcc_decompose(int a, int depth, vector<int>&
     labels, vector<int>& comp,
              vector<int>& arts, vector< vector<
                  int> >& bccs) {
   int children = 0, upward = labels[a] = depth;
   REP(b, n) if (g[a][b]) {
      int u = labels[b];
      if (u < 0) {
         int k = comp.size();
         u = bcc_decompose(b, depth+1, labels,
             comp, arts, bccs);
         if (u >= depth) {
            comp.push_back(a);
            bccs.push_back(vector<int>(comp.
                begin()+k, comp.end()));
            comp.erase(comp.begin()+k, comp.end
                ());
            if (depth > 0 || children > 0)
               arts.push_back(a);
         }
      }
      upward <?= u;
      children++;
   }
   comp.push_back(a);
   if (depth == 0 && children == 0)
      bccs.push_back(comp);
   return upward;
}

void bcc_decompose(vector<int>& arts, vector<
    vector<int> >& bccs) {
   vector<int> comp, labels(n, -1);
   REP(r, n) if (labels[r] < 0) {
      comp.clear();
      bcc_decompose(r, 0, labels, comp, arts,
          bccs);
   }
}
```

## 2.7 橋

```
int bridge(Graph& g, int here, int parent,
    vector<int>& labels,
     vector<int>& current, vector< vector<int>
         >& comps, int& id) {
  int& myid = labels[here];
  assert(myid < 0);
  myid = id++;

  int res = myid;
  Edges& v = g[here];
  REP(i, v.size()) {
    int there = v[i];
    if (there == parent)
      continue;
    int child;
    if (labels[there] < 0) {
      child = bridge(g, there, here, labels,
          critical, id);
      if (child > myid)
        critical[here][there] = critical[there][
            here] = true;
    }
    else {
      child = labels[there];
    }
    res <?= child;
  }

  return res;
}
```

## 2.8 Union-Find

```
int n;
vector<int> data(n, -1);
bool link(int a, int b) { // 新たな併合を行うとtrue
  int ra = find_root(a);
  int rb = find_root(b);
  if (ra != rb) {
    if (data[rb] < data[ra])
      swap(ra, rb);
    data[ra] += data[rb];
    data[rb] = ra;
  }
  return (ra != rb);
}
bool check(int a, int b) { // 同じ集合ならtrue
  return (find_root(a) == find_root(b));
}
```

```
int find_root(int a) { // 代表元を返す
  return ((data[a] < 0) ? a : (data[a] =
      find_root(data[a])));
}
```

## 2.9 二部グラフのマッチング

DFS で増大路を求めていく。片側の辺でメモすれば $O(V(V+E))$. 二部グラフにおいて、最大マッチングの大きさは最小点被覆の大きさに一致する。以下はもっと速い Hopcroft-Karp. 計算量は $O(\sqrt{V}(V+E))$.

```
vector< vector<int> > g;
int n, m;

vector<bool> visited, matched;
vector<int> levels, matching;

bool augment(int left) {
  if (left == n)
    return true;
  if (visited[left])
    return false;
  visited[left] = true;
  REP(i, g[left].size()) {
    int right = g[left][i];
    int next = matching[right];
    if (levels[next] > levels[left] && augment(
        next)) {
      matching[right] = left;
      return true;
    }
  }
  return false;
}
int match() {
  matching.assign(m, n);
  matched.assign(n, false);
  bool cont;
  do {
    levels.assign(n+1, -1);
    levels[n] = n;
    queue<int> q;
    REP(left, n) {
      if (!matched[left]) {
        q.push(left);
        levels[left] = 0;
      }
    }
    while(!q.empty()) {
      int left = q.front();
      q.pop();
      REP(i, g[left].size()) {
        int right = g[left][i];
        int next = matching[right];
        if (levels[next] < 0) {
          levels[next] = levels[left] + 1;
          q.push(next);
        }
      }
    }
    visited.assign(n, false);
    cont = false;
    REP(left, n)
      if (!matched[left] && augment(left))
        matched[left] = cont = true;
  } while(cont);
  return count(ALLOF(matched), true);
}
```

## 2.10 ハンガリアン法

$O(n^2m)$ $(n < m)$.

```
#define residue(i,j) (adj[i][j] + ofsleft[i] +
    ofsright[j])

vector<int> min_cost_match(vector< vector<int> >
    adj) {
  int n = adj.size();
  int m = adj[0].size();
  vector<int> toright(n, -1), toleft(m, -1);
  vector<int> ofsleft(n, 0), ofsright(m, 0);

  REP(r, n) {
    vector<bool> left(n, false), right(m, false);
    vector<int> trace(m, -1), ptr(m, r);
    left[r] = true;

    for(;;) {
      int d = numeric_limits<int>::max();
      REP(j, m) if (!right[j])
        d <?= residue(ptr[j], j);
      REP(i, n) if (left[i])
        ofsleft[i] -= d;
      REP(j, m) if (right[j])
        ofsright[j] += d;
      int b = -1;
      REP(j, m) if (!right[j] && residue(ptr[j],
          j) == 0)
        b = j;
      trace[b] = ptr[b];
      int c = toleft[b];
      if (c < 0) {
        while(b >= 0) {
          int a = trace[b];
          int z = toright[a];
          toleft[b] = a;
          toright[a] = b;
          b = z;
        }
        break;
      }
      right[b] = left[c] = true;
      REP(j, m) if (residue(c, j) < residue(ptr[j
          ], j))
        ptr[j] = c;
    }
  }
  return toright;
}
```

## 2.11 安定結婚

任意の希望リストについて、安定なマッチングが少なくとも 1 つ存在することが知られている。以下は男性優先の解を求める。返り値は `res[womanID] = manID` なる割り当て。各人について、優先度の高い相手から順に並べる。`_man[manID][rank] = womanID`, `_woman[womanID][rank] = manID`. $O(n^2)$.

```
inline vector<int> stable_marriage(const vector<
    vector<int> >& _man,
                const vector<vector<int> >& _woman
                    ) {
  int n = _man.size();
  vector<vector<int> > man(n, n), woman(n, n);
```

```
  vector<int> res(n, -1);

  REP(i, n) REP(j, n){
    man[i][j] = _man[i][n-j-1];
    woman[i][_woman[i][j]] = n - j - 1;
  }
  REP(i, n){
    for(int manID = i; manID >= 0; ){
      int womanID = man[manID].back();
      man[manID].pop_back();
      int prevman = res[womanID];
      if(prevman < 0 || woman[womanID][prevman] <
           woman[womanID][manID]){
        res[womanID] = manID;
        manID = prevman;
      }
    }
  }
  return res;
}
```

## 2.12 最大流

重み無しグラフの意味での最短路に沿ってフローを流し続けるのが Edmonds-Karp。それにレベルグラフを導入し改善を図ったのが以下の Dinic. 計算時間は $O(V^4)$. 隣接リスト表現を使えば $O(V^2E)$.

```
int n, s, t;
int cap[N][N], flow[N][N];
int levels[N];
bool finished[N];
#define residue(i,j) (cap[i][j] - flow[i][j])

void levelize() {
  memset(levels, -1, sizeof(levels));
  queue<int> q;
  levels[s] = 0; q.push(s);
  while(!q.empty()) {
    int here = q.front(); q.pop();
    REP(there, n) if (levels[there] < 0 &&
        residue(here, there) > 0) {
      levels[there] = levels[here] + 1;
      q.push(there);
    }
  }
}
int augment(int here, int cur) {
  if (here == t || cur == 0)
    return cur;
  if (finished[here])
    return 0;
  finished[here] = true;
  REP(there, n) if (levels[there] > levels[here
      ]) {
    int f = augment(there, min(cur, residue(here,
         there)));
    if (f > 0) {
      flow[here][there] += f;
      flow[there][here] -= f;
      finished[here] = false;
      return f;
    }
  }
  return 0;
}
int maxflow() {
  memset(flow, 0, sizeof(flow));
  int total = 0;
  for(bool cont = true; cont; ) {
    cont = false;
    levelize();
    memset(finished, 0, sizeof(finished));
    for(int f; (f = augment(s, INF)) > 0; cont =
        true)
      total += f;
  }
  return total;
}
```

## 2.13 最小カット

Stoer-Wagner. $O(V^3)$. グラフの隣接リスト表現と priority queue を使えば $O(VE \log E)$. **非負の重みであること。隣接行列`adj` は破壊される。**

```
Weight adj[N][N];
int n;

int min_cut() {
  Weight res = WEIGHT_INFTY;

  vector<int> v;
  REP(i, n)
    v.push_back(i);
  for(int m = n; m > 1; m--) {
    vector<Weight> ws(m, 0);
    int s, t;
    Weight w;
    REP(k, m) {
      s = t;
      t = max_element(ws.begin(), ws.end()) - ws.
          begin();
      w = ws[t];
      ws[t] = -1;
      REP(i, m)
        if (ws[i] >= 0)
          ws[i] += adj[v[t]][v[i]];
    }
    REP(i, m) {
      adj[v[i]][v[s]] += adj[v[i]][v[t]];
      adj[v[s]][v[i]] += adj[v[t]][v[i]];
    }
    v.erase(v.begin()+t);
    res <?= w;
  }
  return res;
}
```

## 2.14 最小費用流

Primal-Dual. **コスト行列は反対称であること。キャパシティ行列は非負であること。キャパシティが正の枝はコストが非負であること。**

```
#define residue(i,j) (cap[i][j] - flow[i][j])
#define rcost(i,j) (cost[i][j] + pot[i] - pot[j
    ])
Weight min_cost_flow(vector< vector<Flow> > cap,
     vector< vector<Weight> > cost, int s, int t
    ) {
  int n = cap.size();

  vector< vector<Flow> > flow(n, n);
  vector<Weight> pot(n);

  Weight res = 0;
  for(;;) {
```

```
  vector<Weight> dists(n+1, WEIGHT_INFTY);
  vector<bool> visited(n, false);
  vector<int> trace(n, -1);
  dists[s] = 0;

  for(;;) {
    int cur = n;
    REP(i, n) if (!visited[i] && dists[i] <
        dists[cur])
      cur = i;
    if (cur == n)
      break;
    visited[cur] = true;
    REP(next, n) if (residue(cur, next) > EPS)
        {
      if (dists[next] > dists[cur] + rcost(cur,
          next) && !visited[next]) {
        dists[next] = dists[cur] + rcost(cur,
            next);
        trace[next] = cur;
      }
    }
  }
  if (!visited[t])
    break;

  Flow f = FLOW_INFTY;
  for(int c = t; c != s; c = trace[c])
    f <?= residue(trace[c], c);
  for(int c = t; c != s; c = trace[c]) {
    flow[trace[c]][c] += f;
    flow[c][trace[c]] -= f;
    res += cost[trace[c]][c] * f;
  }

  REP(i, n)
    pot[i] += dists[i];
 }

 return res;
}
```

## 2.15 二部グラフの辺彩色

二部グラフの辺に彩色をし、同一ノードに繋がっている辺が同じ色を持たないようにする。最小彩色数はグラフの最大次数に等しい。

```
int n, m; // 左右のノード数
vector< vector<int> > g; // グラフ (index -> [
    index])
vector< vector<int> > cl; // 色づけ (index ->
    color -> index)

BGEC(int n, int m) : n(n), m(m), g(n+m) {}
void add_edge(int a, int b) { // 左
    a、右 b の間に辺を張る
 g[a].push_back(n+b);
 g[n+b].push_back(a);
}
int color() {
 int d = 0; // 彩色数
 REP(i, n+m)
   d = max<int>(d, g[i].size());
 cl.assign(n+m, vector<int>(d, -1));
 REP(a, n) REP(i, g[a].size()) {
   int b = g[a][i];
   int ca = min_element(ALLOF(cl[a])) - cl[a].
       begin();
   int cb = min_element(ALLOF(cl[b])) - cl[b].
       begin();
   if (ca != cb) {
     augment(b, ca, cb);
     cb = ca;
   }
   cl[a][ca] = b;
   cl[b][cb] = a;
 }
 return d;
}
void augment(int a, int c1, int c2) {
 int b = cl[a][c1];
 if (b >= 0) {
   augment(b, c2, c1);
   cl[b][c2] = a;
   cl[a][c1] = -1;
 }
 cl[a][c2] = b;
}
```

### 2.15.1 最小平均長閉路

平均長が最小である単純閉路を探すアルゴリズム。Karp による Bellman-Ford アルゴリズムの変形で、次の事実を用いる:

$$\mathrm{minavg} = \min_{v \in V} \max_{0 \le i < n} \frac{d_n(v) - d_i(v)}{n - i}.$$

このコードで求まるのは閉路の平均長。閉路自体を求めるには、Bellman-Ford の過程で最小値に至るルートを保存して後で辿る。最小費用循環を効率よく求める際に用いる。O($nm$).

```
Edges edges;

Weight dp[N+1][N] = { { 0 } };
REP(k, n) {
   REP(i, n)
      dp[k+1][i] = INF;
   REP(i, m) {
      Edge& e = edges[i];
      if (dp[k][e.src] < INF)
         dp[k+1][e.dest] <?= dp[k][e.src] + e.
             weight;
   }
}
Weight res = INF;
REP(i, n) {
   Weight lo = -INF;
   REP(k, n) if (dp[n][i] < INF && dp[k][i] <
       INF)
      lo >?= (dp[n][i] - dp[k][i]) / (n - k);
   if (lo > -INF)
      res <?= lo;
}
```

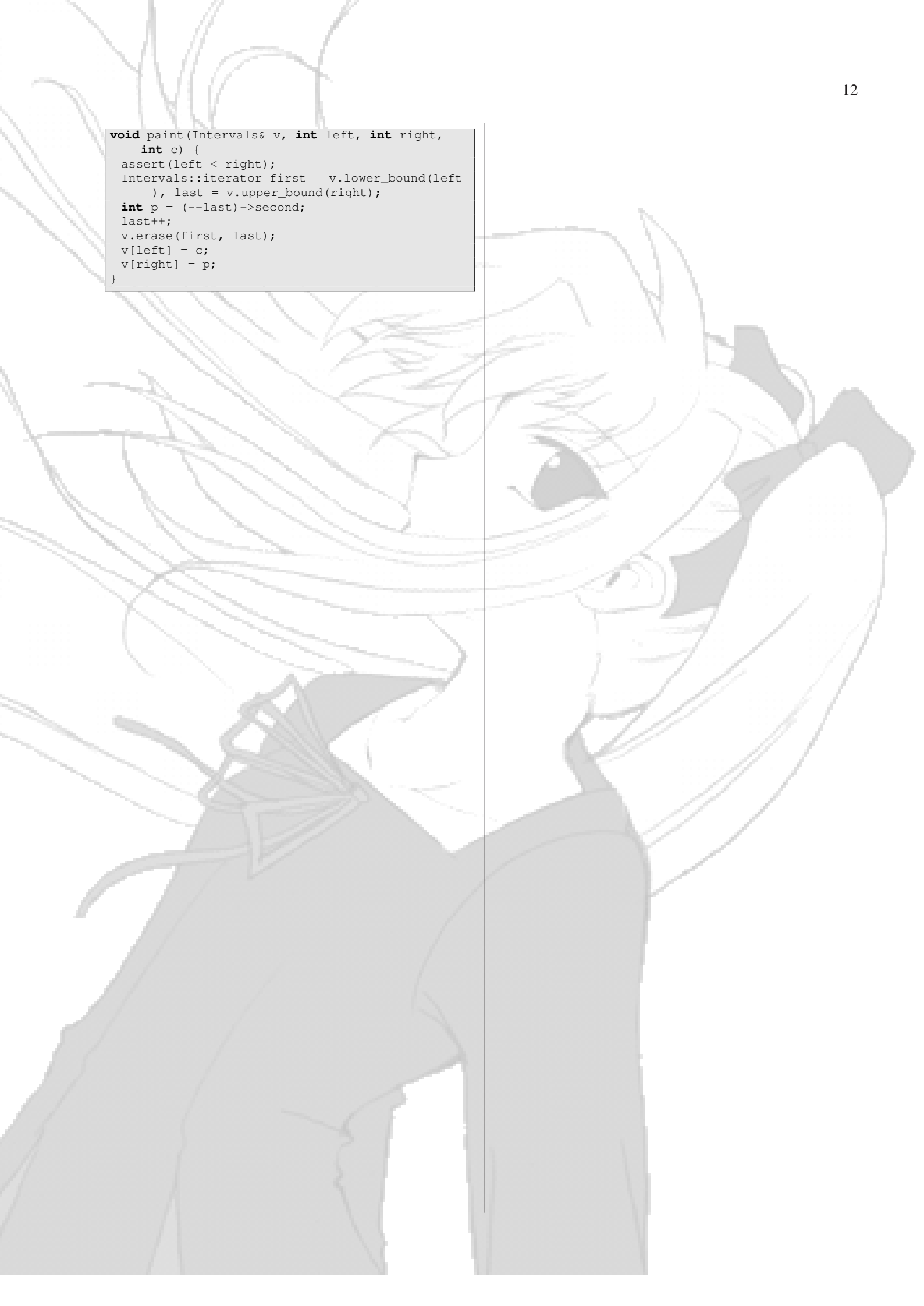
## 2.16 範囲の更新

**Intervals には初期値として「負無限大=無色」を代入しておくこと。**

```
typedef map<int, int> Intervals;

// paint [left, right) with color c
```

```
void paint(Intervals& v, int left, int right,
    int c) {
  assert(left < right);
  Intervals::iterator first = v.lower_bound(left
      ), last = v.upper_bound(right);
  int p = (--last)->second;
  last++;
  v.erase(first, last);
  v[left] = c;
  v[right] = p;
}
```



# 3 数論

## 3.1 数表

**■約数の個数**

| N | N 以下の数の約数の個数の max |
|---|---|
| 1e3 | 32 |
| 1e4 | 64 |
| 1e5 | 128 |
| 1e6 | 240 |
| 1e7 | 448 |
| 1e8 | 768 |
| 1e9 | 1344 |
| INT_MAX | 1600 |
| UINT_MAX | 1920 |

**■素数の個数**

| N | N 以下の素数の個数 |
|---|---|
| 1e1 | 4 |
| 1e2 | 25 |
| 1e3 | 168 |
| 1e4 | 1,229 |
| 1e5 | 9,592 |
| 1e6 | 78,498 |
| 1e7 | 664,579 |
| 1e8 | 5,761,455 |
| 1e9 | 50,847,534 |
| 1e10 | 455,052,511 |

**■分割数**

| N | N の分割数 |
|---|---|
| 10 | 42 |
| 20 | 627 |
| 30 | 5,604 |
| 40 | 37,338 |
| 50 | 204,226 |
| 60 | 966,467 |
| 70 | 4,087,968 |
| 80 | 15,796,476 |
| 90 | 56,634,173 |
| 100 | 190,569,292 |

## 3.2 定理

**■Bertrand の仮説**

Chebyshev が証明してるので、定理である。任意の自然数 n に対して n と 2n の間に必ず素数が存在する。

**■Wilson の定理**

$p$ が素数ならば、$(p-1)! \equiv -1 \pmod{p}$. $p > 1$ に対しては逆も成り立つ。

**■Ptolemy の定理**

円に内接する四角形の対角線の長さの積 = 向かい合う二組の辺の長さの積の和

**■五心の公式**

- 三点が与えられたときの五心の座標
- 重心
  - (a+b+c)/3
- 垂心 (TODO: complex で書き直す)

```
// (a,b),(c,d),(e,f)
a-=e,b-=f,c-=e,d-=f;
p=(b*d+a*c)/(a*d-b*c);
p*(d-b)+e,p*(a-c)+f;
```

- TODO: 残り三つ (傍心はパス?)
- 外接円の半径

```
// (a,b),(c,d),(e,f)
hypot(a-=c,b-=d)*hypot(c-=e,d-=f)*hypot(a+c,b+d
    )*acos(-1)/fabs(a*d-b*c)
```

- 外接円の半径 (complex) (未検証)

```
// a,b,c
norm(a-b)*norm(b-c)*norm(c-a)*acos(-1)/fabs(outp
    (a,b)+outp(b,c)+outp(c,a))
```

**■Pick の定理**

格子点上に頂点を持つ多角形の、面積 S、内部の格子点の数 i、辺上の格子点の数 b に対して、S=i+b/2-1

## 3.3 オイラーの $\phi$ 関数

$\phi(n)$ は自然数 $n$ と互いに素な $n$ 以下の数の個数

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

```
// phi(1)..phi(N) を求める
void phi_all(int N) {
  int a[N+1], b[N+1];
  REP(i, N+1) {
    a[i] = 1;
    b[i] = i;
  }

  REP(k, N+1) {
    if (b[k] < 2)
      continue;
    for(int n = k; n <= N; n += k) {
      for(int m = k-1; b[n]%k == 0; m = k) {
        a[n] *= m;
        b[n] /= k;
      }
    }
  }
  // a[n] == phi(n), b[n] == 1
}
```

## 3.4 中国剰余定理 1

中国剰余定理とは、$n$ 個の整数 $m_0, \ldots, m_{n-1}$ がどの 2 つの要素も互いに素ならば、与えられる $r_0, \ldots, r_{n-1}$ に対して $x = r_i \mod m_i$ を満たすような $x$ が、$\prod_i m_i$ を法として唯一つ存在する、というもの。そのような $x$ を $0 \le x \le \prod_i m_i$ の範囲で求めるアルゴリズム。但し実装時にはオーバーフローに注意すること。**m[i] はどの 2 つの要素も互いに素であること。**

```
integer chinese_remainder(const vector<int>& m,
    const vector<int>& r) {
  int n = m.size();
  integer prod = 1;
  REP(i, n)
  prod *= m[i];

  integer res = 0;
  REP(i, n){
  integer M = prod / m[i];
  integer a = divide(M, 1, m[i]);
  integer R = r[i] - r[i] / prod * prod;
  if(R < 0)
    R += prod;
  res = (res + M * a * r[i] % prod) % prod;
  }

  return res;
}
```

## 3.5 中国剰余定理 2

2 つの制約「K=ax+b」「K=cy+d」に対して、それらを同時に満たす K は「K=ez+f」という一つの式で表せる (か、もしくはそのような K は存在しない)。ただし、e=lcm(a,c)。a と c は互いに素でなくてもよい。

以下はそのような e,f を求めるアルゴリズム。f の値の正規化方法は臨機応変に。オーバーフローに注意すること。

N 個の制約について求めたい場合は N-1 回呼んでください。もしくは、1x+0 に対して N 回 fold。

**互いに素である必要はないが、答えがない場合に注意**

```
// ax+b の a と b
struct Constraint {
  int mult;
  int base;
};

// BE CAREFUL OF OVERFLOW!!
Constraint chinese_remainder(const Constraint& a
    , const Constraint& b) {
  Constraint r;
  int g = gcd(a.mult, b.mult);
  int d = b.base - a.base;

  // 解がない場合はこの
     assert で落ちるので、マズい場合は適宜修正すること
  assert(d % g == 0);
  d /= g;

  // このへんでオーバーフローに注意
  r.mult = a.mult / g * b.mult;
  r.base = a.mult * d * inv(a.mult/g, b.mult/g)
      + a.base;

  // ここから解の正規化
  // 以下のコードではK=ax+b=cy+d=ez+f に対して、
  // x>=0,y>=0の条件がついていた場合にz>=0を必要十分とする
     ようなf を構成している
  // while ループが長引きそうなら割り算にしたほうがいいかもね
  r.base %= r.mult;
  while(r.base < a.base || r.base < b.base)
    r.base += r.mult;

  return r;
}
```

## 3.6 逆数

$ax \equiv 1 \pmod p$ なる $x$ を求める。$p$ が素数でない場合、$a$ の逆元が存在すればそれを返す。$a$ に 0 を指定したり、逆元が存在しなければ division by zero で落ちる。

```
int inv(int a, int p) {
  return ( a == 1 ? 1 : (1 - p*inv(p%a, a)) / a
      + p );
}
```

## 3.7 拡張ユークリッド互除法

整数 $a, b$ に対して、$ax + by = \gcd(a, b)$ となる整数 $x, y$ を求める。

```
void xgcd(integer a, integer b, integer& x,
    integer& y) {
  if (b == 0) {
    x = 1;
    y = 0;
  }
  else {
    xgcd(b, a%b, y, x);
    y -= a/b*x;
  }
}
```

## 3.8 線形ディオファントス方程式

$an = b \mod m$ となる、非負でかつ最小の $n$ を求める。もしくは、$m$ を法とした剰余環上で $b/a$ を計算する。**$a, b, m$ は非負であること。**

```
struct no_solution {};
integer divide(integer a, integer b, integer m)
    {
  integer g = gcd(a, m);
  if (b%g != 0)
    throw no_solution();
  integer x, y;
  xgcd(a, m, x, y);
  assert(a*x+m*y == gcd(a,m));
  integer n = x*b/g;
  integer dn = m/g;
  n -= n/dn*dn;
  if (n < 0)
    n += dn;
  return n;
}
```

## 3.9 行列演算

```
typedef double* vector_t;
typedef vector_t* matrix_t;
matrix_t new_matrix(int n) {
  matrix_t x = new vector_t[n];
  for(int i = 0; i < n; i++)
    x[i] = new double[n];
  return x;
}
matrix_t dup_matrix(matrix_t x_, int n) {
  matrix_t x = new_matrix(n);
  for(int i = 0; i < n; i++)
    copy(x_[i], x_[i]+n, x[i]);
  return x;
}
void delete_matrix(matrix_t x, int n) {
  for(int i = 0; i < n; i++)
    delete[] x[i];
  delete[] x;
}
matrix_t multiply(matrix_t a, matrix_t b, int n)
    {
  matrix_t r = new_matrix(n);
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      r[i][j] = 0;
      for(int k = 0; k < n; k++)
        r[i][j] += a[i][k] * b[k][j];
    }
  }
  return r;
}
```

## 3.10 行列の高速べき乗

```
matrix_t pow(matrix_t& e, int n, int m) {
  matrix_t r = new_matrix(n);

  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      r[i][j] = ((m&1) == 0 ? (i == j ? 1 : 0) :
          e[i][j]);
  if (m >= 2) {
    matrix_t u = pow(e, n, m/2);
    matrix_t uu = multiply(u, u, n);
    matrix_t z = multiply(r, uu, n);
    delete_matrix(u, n);
    delete_matrix(uu, n);
    delete_matrix(r, n);
    r = z;
  }

  return r;
}
```

## 3.11 ガウスの消去法

連立方程式 $Ax = b$ を解く。$A, b$ は破壊され、答えが $b$ に代入される。invert, modulo を別途定義すること。$A, b$ は modulo による正規化が既に行われているものとする。

```
void gauss(matrix_t& A, vector_t& b, int n, int
    m) {
  int pi = 0, pj = 0;
  while(pi < n && pj < m) {
    for(int i = pi+1; i < n; i++) {
      if (abs(A[i][pj]) > abs(A[pi][pj])) {
        swap(A[i], A[pi]);
        swap(b[i], b[pi]);
      }
    }
    if (abs(A[pi][pj]) > 0) {
      int d = invert(A[pi][pj]);
      REP(j, m)
        A[pi][j] = modulo(A[pi][j] * d);
      b[pi] = modulo(b[pi] * d);
      for(int i = pi+1; i < n; i++) {
        int k = A[i][pj];
        REP(j, m)
          A[i][j] = modulo(A[i][j] - k * A[pi][j
              ]);
        b[i] = modulo(b[i] - k * b[pi]);
      }
      pi++;
    }
    pj++;
  }
  for(int i = pi; i < n; i++)
    if (abs(b[i]) > 0)
      throw Inconsistent();
  if (pi < m || pj < m)
    throw Ambiguous();
  for(int j = m-1; j >= 0; j--)
    REP(i, j)
      b[i] = modulo(b[i] - b[j] * A[i][j]);
}
```

## 3.12 LU 分解

a を破壊し LU 形式に変換する。p は行交換の情報を保持する。

```
bool lu_decompose(matrix_t& a, int* p, int n) {
  for(int i = 0; i < n; i++)
    p[i] = i;
  for(int k = 0; k < n; k++) {
    int pivot = k;
    for(int i = k+1; i < n; i++)
      if (abs(a[i][k]) > abs(a[pivot][k]))
```

```
      pivot = i;
    swap(a[k], a[pivot]);
    swap(p[k], p[pivot]);
    if (abs(a[k][k]) < EP)
      return false;
    for(int i = k+1; i < n; i++) {
      double m = (a[i][k] /= a[k][k]);
      for(int j = k+1; j < n; j++)
        a[i][j] -= a[k][j] * m;
    }
  }
  return true;
}
void lu_solve(matrix_t& a, int* p, vector_t& b,
    vector_t& x, int n) {
  for(int i = 0; i < n; i++)
    x[i] = b[p[i]];
  for(int k = 0; k < n; k++)
    for(int i = 0; i < k; i++)
      x[k] -= a[k][i] * x[i];
  for(int k = n-1; k >= 0; k--) {
    for(int i = k+1; i < n; i++)
      x[k] -= a[k][i] * x[i];
    x[k] /= a[k][k];
  }
}
```

## 3.13 単体法

minimize $\boldsymbol{cx}$ s.t. $\boldsymbol{Ax} = \boldsymbol{b}$. 解が存在しない/最適値が発散する場合は `vector_t()` を返す。単体法は最悪の場合で指数時間がかかるので、あくまで最終手段。これを使う前に、ほかの方法が使えないか十分考えること。特に、2 変数の不等式制約は二次元の凸多角形クリッピングとして捉えることができる。

```
const double INF = numeric_limits<double>::
    infinity();
typedef vector<double> vector_t;
typedef vector<vector_t> matrix_t;

vector_t simplex(matrix_t A, vector_t b,
    vector_t c) {
  const int n = c.size(), m = b.size();

  REP(i, m) if (b[i] < 0) {
    REP(j, n)
      A[i][j] *= -1;
    b[i] *= -1;
  }
  vector<int> bx(m), nx(n);
  REP(i, m)
    bx[i] = n+i;
  REP(i, n)
    nx[i] = i;
  A.resize(m+2);
  REP(i, m+2)
    A[i].resize(n+m, 0);
  REP(i, m)
    A[i][n+i] = 1;
  REP(i, m) REP(j, n)
    A[m][j] += A[i][j];
  b.push_back(accumulate(ALLOF(b), (double
      )0.0));
  REP(j, n)
    A[m+1][j] = -c[j];
  REP(i, m)
    A[m+1][n+i] = -INF;
  b.push_back(0);

  REP(phase, 2) {
    for(;;) {
      int ni = -1;
      REP(i, n)
        if (A[m][nx[i]] > EPS && (ni < 0 || nx[i]
            < nx[ni]))
          ni = i;
      if (ni < 0)
        break;
      int nv = nx[ni];
      vector_t bound(m);
      REP(i, m)
        bound[i] = (A[i][nv] < EPS ? INF : b[i] /
            A[i][nv]);
      if (!(*min_element(ALLOF(bound)) < INF))
        return vector_t(); // -infinity
      int bi = 0;
      REP(i, m)
        if (bound[i] < bound[bi]-EPS || (bound[i]
            < bound[bi]+EPS && bx[i] < bx[bi]))
          bi = i;
      double pd = A[bi][nv];
      REP(j, n+m)
        A[bi][j] /= pd;
      b[bi] /= pd;
      REP(i, m+2) if (i != bi) {
        double pn = A[i][nv];
        REP(j, n+m)
          A[i][j] -= A[bi][j] * pn;
        b[i] -= b[bi] * pn;
      }
      swap(nx[ni], bx[bi]);
    }
    if (phase == 0 && abs(b[m]) > EPS)
      return vector_t(); // no solution
    A[m].swap(A[m+1]);
    swap(b[m], b[m+1]);
  }
  vector_t x(n+m, 0);
  REP(i, m)
    x[bx[i]] = b[i];
  x.resize(n);
  return x;
}
```

# 4 文字列

## 4.1 KMP

作成されるテーブルは「パターン中の先頭 $n$ 文字を考えたときに、その prefix と suffix が一致するような部分長さ $L$ $(0 \leq L < n)$ のうち最長のもの」を保持する。
$O(n+m)$.

```
vector<int> kmp(string s) {
  int n = s.size();
  vector<int> table(n+1);
  int q = table[0] = -1;
  for(int p = 1; p <= n; p++) {
    while(q >= 0 && s[q] != s[p-1])
      q = table[q];
    table[p] = ++q;
  }
  return table;
}
```

## 4.2 Aho-Corasick

```
struct AC {
  AC* fail;
  AC* next[4];
  vector<int> accepts;

  AC() : fail(0) {
    memset(next, 0, sizeof(next));
  }
  void insert(const int* p, int id) {
    if (*p < 0)
      accepts.push_back(id);
    else
      (next[*p] ?: (next[*p] = new AC()))->insert
          (p+1, id);
  }
  void make() {
    this->fail = 0; // this is root
    queue<AC*> q;
    q.push(this);
    while(!q.empty()) {
      AC* cur = q.front();
      q.pop();
      REP(k, 4) {
        AC*& next = cur->next[k];
        if (!next) {
          next = (cur->fail ? cur->fail->next[k] :
                this);
        }
        else {
          AC* fail = cur->fail;
          while(fail && !fail->next[k])
            fail = fail->fail;
          fail = (fail ? fail->next[k] : this);
          next->fail = fail;
          next->accepts.insert(next->accepts.end
              (), ALLOF(fail->accepts));
          q.push(next);
        }
      }
    }
  }
};
```

## 4.3 Suffix Array

### 4.3.1 Larsson Sadakane

たぶん $O(n \log n)$.

```
int* suffix_array(unsigned char* str) {
  int n = strlen((char*)str);
  int *v, *u, *g, *b, r[256];
  v = new int[n+1]; u = new int[n+1]; g = new
      int[n+1]; b = new int[n+1];
  for(int i = 0; i <= n; i++) {
    v[i] = i;
    g[i] = str[i];
  }
  for(int h = 1; ; h *= 2) {
    int m = *max_element(g, g+n+1);
    for(int k = h; k >= 0; k -= h) {
      for(int ord = 0; m >> ord; ord += 8) {
        memset(r, 0, sizeof(r));
        for(int i = 0; i <= n; i++)
          r[(g[min(v[i]+k, n)] >> ord) & 0xff]++;
        for(int i = 1; i < 256; i++)
          r[i] += r[i-1];
        for(int i = n; i >= 0; i--)
          u[ --r[(g[min(v[i]+k, n)] >> ord) & 0xff
              ] ] = v[i];
        swap(u, v);
      }
    }
    b[0] = 0;
    for(int i = 1; i <= n; i++)
      b[i] = b[i-1] + ((g[v[i-1]] != g[v[i]] ? g[
          v[i-1]] < g[v[i]] : g[v[i-1]+h] < g[v[i
          ]+h]) ? 1 : 0);
    if (b[n] == n)
      break;
    for(int i = 0; i <= n; i++)
      g[v[i]] = b[i];
  }
  delete[] g; delete[] b; delete[] u;
  return v;
}
```

### 4.3.2 Kaŕkkaïnen Sanders

テキストサイズ $\geq 2$.

```
// Derived from:
// J. Karkkainen and P. Sanders: Simple Linear
    Work Suffix Array Construction.
// In 30th International Colloquium on Automata,
    Languages and Programming,
// number 2719 in LNCS, pp. 943--955, Springer,
    2003.

void radix_sort(int* src, int* dest, int* rank,
    int n, int m) {
  int* cnt = new int[m+1];
  memset(cnt, 0, sizeof(int)*(m+1));
  for(int i = 0; i < n; i++)
    cnt[rank[src[i]]]++;
  for(int i = 1; i <= m; i++)
    cnt[i] += cnt[i-1];
  for(int i = n-1; i >= 0; i--)
    dest[--cnt[rank[src[i]]]] = src[i];
}

bool triple_less(int a0, int a1, int a2, int b0,
    int b1, int b2) {
  if (a0 != b0)
    return (a0 < b0);
```

```
  if (a1 != b1)
    return (a1 < b1);
  return (a2 < b2);
}

void suffix_array(int* text, int* sa, int n, int
     m) {
  assert(n >= 2);

  int n0 = (n+2)/3, n1 = (n+1)/3, n2 = n/3, n02
      = n0 + n2;

  int* rank12 = new int[n02+3]; rank12[n02+0] =
      rank12[n02+1] = rank12[n02+2] = 0;
  int* sa12 = new int[n02+3]; sa12[n02+0] = sa12
      [n02+1] = sa12[n02+2] = 0;
  int* tmp0 = new int[n0];
  int* sa0 = new int[n0];

  for(int i = 0; i < n02; i++)
    rank12[i] = i*3/2+1;
  radix_sort(rank12, sa12, text+2, n02, m);
  radix_sort(sa12, rank12, text+1, n02, m);
  radix_sort(rank12, sa12, text+0, n02, m);

  int name = 0;
  for(int i = 0; i < n02; i++) {
    if (name == 0 ||
       text[sa12[i]+0] != text[sa12[i-1]+0] ||
       text[sa12[i]+1] != text[sa12[i-1]+1] ||
       text[sa12[i]+2] != text[sa12[i-1]+2])
      name++;
    rank12[sa12[i]/3 + (sa12[i]%3 == 1 ? 0 : n0)]
        = name;
  }

  if (name < n02) {
    suffix_array(rank12, sa12, n02, name);
    for(int i = 0; i < n02; i++)
      rank12[sa12[i]] = i+1;
  }
  else {
    for(int i = 0; i < n02; i++)
      sa12[rank12[i]-1] = i;
  }

  for(int i = 0, j = 0; i < n02; i++)
    if (sa12[i] < n0)
      tmp0[j++] = 3*sa12[i];
  radix_sort(tmp0, sa0, text, n0, m);

  for(int i = 0, j = n0-n1, k = 0; k < n; ) {
    int a = sa0[i];
    int b = (sa12[j] < n0 ? sa12[j]*3+1 : (sa12[j
        ]-n0)*3+2);
    if (b%3 == 1 ?
        triple_less(text[a], rank12[a/3], 0,
                    text[b], rank12[b/3+n0], 0) :
                    triple_less(text[a], text[a+1],
                         rank12[a/3+n0],
                             text[b], text[b+1],
                                 rank12[b/3+1])) {
      sa[k++] = a;
      if (++i == n0)
        while(k < n)
          sa[k++] = (sa12[j] < n0 ? sa12[j++]*3+1
              : (sa12[j++]-n0)*3+2);
    }
    else {
      sa[k++] = b;
      if (++j == n02)
        while(k < n)
          sa[k++] = sa0[i++];
    }
  }

  delete[] rank12; delete[] sa12; delete[] tmp0;
       delete[] sa0;
}
```

### 4.3.3 高さ配列

Kasai らによる、構築された Suffix Array について、順位が隣り合った二つの suffix の longest common prefix を求めるアルゴリズム。

```
int* height_array(char* text, int* sa, int n) {
  int* height = new int[n];
  int* rank = new int[n];
  REP(i, n)
    rank[sa[i]] = i;
  int h = 0;
  height[0] = 0;
  REP(i, n) {
    if (rank[i] > 0) {
      int k = sa[rank[i]-1];
      while(text[i+h] == text[k+h])
        h++;
      height[rank[i]] = h;
      if (h > 0)
        h--;
    }
  }
  delete[] rank;
  return height;
}
```

# 5 その他

## 5.1 ビット演算

```
unsigned int __builtin_popcount(unsigned int);
    // 1であるビットの数を数える。
unsigned long long __builtin_popcountll(unsigned
    long long);
unsigned int __builtin_ctz(unsigned int); // 末尾
    の 0であるビットの数を数える。
unsigned long long __builtin_ctzll(unsigned long
    long);
y = x&-x; // 最右ビットを抜き出す。
y = x&(x-1); // 最右ビットを落とす。
#define FOR_SUBSET(b, a) for(int b = (a)&-(a); b
    != 0; b = (((b|~(a))+1)&(a)))
#define FOR_SUBSET(b, a) for(int b = a; b != 0;
    b = (b-1)&a)
int next_combination(int p) {
  int lsb = p&-p;
  int rem = p+lsb;
  int rit = rem&~p;
  return rem|(((rit/lsb)>>1)-1);
}
```

## 5.2 平衡木

Treap. サブツリーに対する操作を行ってはいけない。

```
template<class Key, class Value, class Cache>
struct Treap {
  Key key;
  Value value;
  int prio;
  Treap* ch[2];
  bool cached;
  Cache cache;
  Treap(const Key& key, const Value& value) :
      key(key), value(value), prio(rand()),
      cached(false) {
    ch[0] = ch[1] = 0; // is this necessary?
  }
  Treap* insert(const Key& newkey, const Value&
      newvalue) {
    if (!this)
      return new Treap(newkey, newvalue);
    if (newkey == key)
      return this;
    int side = (newkey < key ? 0 : 1);
    ch[side] = ch[side]->insert(newkey, newvalue
        );
    cached = false;
    return rotate(side);
  }
  Treap* rotate(int side) {
    if (ch[side] && ch[side]->prio > prio) {
      Treap* rot = ch[side];
      this->ch[side] = rot->ch[side^1];
      rot->ch[side^1] = this;
      this->cached = rot->cached = false;
      return rot;
    }
    return this;
  }
  void clear() {
    if (this) {
      ch[0]->clear();
      ch[1]->clear();
      delete this;
    }
  }
  Treap* remove(const Key& oldkey) {
    if (!this)
      return this;
    if (key == oldkey) {
      if (!ch[1]) {
        Treap* res = ch[0];
        delete this;
        return res;
      }
      pair<Treap*, Treap*> res = ch[1]->
          delete_min();
      res.first->ch[0] = this->ch[0];
      res.first->ch[1] = res.second;
      res.first->cached = false;
      delete this;
      return res.first->balance();
    }
    int side = (oldkey < key ? 0 : 1);
    ch[side] = ch[side]->remove(oldkey);
    cached = false;
    return this;
  }
  pair<Treap*, Treap*> delete_min() {
    cached = false;
    if (!ch[0])
      return make_pair(this, ch[1]);
    pair<Treap*, Treap*> res = ch[0]->delete_min
        ();
    ch[0] = res.second;
    return make_pair(res.first, this);
  }
  inline int priority() {
    return (this ? prio : -1);
  }
  Treap* balance() {
    int side = (ch[0]->priority() > ch[1]->
        priority() ? 0 : 1);
    Treap* rot = rotate(side);
    if (rot == this)
      return this;
    return rot->balance();
  }
  Cache eval() {
    if (!this)
      return Cache();
    if (!cached)
      cache = Cache(key, value, ch[0]->eval(), ch
          [1]->eval());
    cached = true;
    return cache;
  }
};

// 小さいほうからn番目の要素をクエリできるようにする場合の例
struct SizeCache {
  int size;
  SizeCache() : size(0) {}
  template<class Key, class Value>
  SizeCache(const Key& key, const Value& value,
      const SizeCache& left, const SizeCache&
      right)
    : size(left.size+right.size+1) {}
};
template<class Key, class Value>
pair<Key, Value> nth(Treap<Key, Value, SizeCache
    >* root, int k) {
  int l = root->ch[0]->eval().size;
  if (k < l)
    return nth<Key, Value>(root->ch[0], k);
  if (k == l)
    return make_pair(root->key, root->value);
  return nth<Key, Value>(root->ch[1], k-(l+1));
}
```

## 5.3 Fenwick Tree

配列の partial sum の取得と要素の書き換えをそれぞれ対数時間で行うデータ構造。別名 Binary Indexed Tree。n 次元にも容易に拡張できる。和の取得・要素の更新ともに O(log $n$).

```
T bitquery(const vector<T>& bit, int from, int
    to) { // [from, to)
  if (from > 0)
    return bitquery(bit, 0, to) - bitquery(bit,
        0, from);
  T res = T();
  for(int k = to-1; k >= 0; k = (k & (k+1)) - 1)
    res += bit[k];
  return res;
}

void bitupdate(vector<T>& bit, int pos, const T&
     delta) {
  for(const int n = bit.size(); pos < n; pos |=
      pos+1)
    bit[pos] += delta;
}
```

## 5.4 Range Minimum Query

列のある区間の最小値を対数時間で求めるデータ構造。**配列サイズは２のべき乗であること。**初期化に O($n$). 更新・クエリに O(log $n$).

```
// 配列を拡張してRMQに対応させる
void rmq_ext(vector<T>& v) {
  int n = v.size();
  assert(__builtin_popcount(n) == 1); // 長さは2の
      べき乗でなければならない
  v.resize(n*2);
  for(int i = n; i < 2*n; i++)
    v[i] = min(v[(i-n)*2+0], v[(i-n)*2+1]);
}
// 列の要素を書き換える
void rmq_update(vector<T>& rmq, int pos, const T
    & value) {
  int n = rmq.size() / 2;
  rmq[pos] = value;
  while(pos < 2*n-1) {
    rmq[pos/2+n] = min(rmq[pos], rmq[pos^1]);
    pos = pos/2+n;
  }
}
// [from, to)の最小値を取り出す
T rmq_query(const vector<T>& rmq, int from, int
    to) {
  int n = rmq.size() / 2;
  int p = min((from == 0 ? 32 : __builtin_ctz(
      from)), 31-__builtin_clz(to-from));
  T x = rmq[(from>>p)|((n*2*((1<<p)-1))>>p)];
  from += 1<<p;
  if (from < to)
    x = min(x, rmq_query(rmq, from, to));
  return x;
}
```

## 5.5 Range Tree

区間に対する加算、区間中の最小値のクエリ、各要素の値のクエリを対数程度の時間で行う木。range_add に $O((\log n)^2)$.range_min に $O(\log n)$. query に $O(\log n)$.

```
struct range_tree {
  int m;
  vector< pair<int, int> > tree;

  range_tree(int n) {
    m = n;
    while(m&(m-1))
      m += m&-m;
    tree.assign(2*m, make_pair(0, 0));
    for(int k = n; k < m; k++)
      tree[k] = make_pair(INF, INF);
    for(int k = m; k < 2*m-1; k++)
      tree[k] = make_pair(0,
              min(tree[((k&~m)<<1)^0].second,
                tree[((k&~m)<<1)^1].second));
  }
  void range_add(int a, int b, int d) {
    while(a < b) {
      int r = 1, k = a;
      while((a & r) == 0 && a + (r<<1) <= b) {
        r <<= 1;
        k = (k >> 1) | m;
      }
      tree[k].first += d;
      do {
        tree[k].second = tree[k].first +
          (k < m ? zero :
           min(tree[((k&~m)<<1)^0].second,
             tree[((k&~m)<<1)^1].second));
        k = (k >> 1) | m;
      } while(k < 2*m-1);
      a += r;
    }
  }
  int range_min(int a, int b) {
    int res = INF;
    while(a < b) {
      int r = 1, k = a;
      while((a & r) == 0 && a + (r<<1) <= b) {
        r <<= 1;
        k = (k >> 1) | m;
      }
      res = min(res, tree[k].second);
      a += r;
    }
    return res;
  }
  int query(int k) {
    int res = 0;
    while(k < 2*m-1) {
      res += tree[k].first;
      k = (k >> 1) | m;
    }
    return res;
  }
};
```

## 5.6 α-β枝狩り

なんだかんだで毎回書くのに苦労するあれ。

```
// [alpha..beta]の値以外を返しても意味を持たない、という意
    味
```

```
int search(int alpha = -INF, int beta = INF) {
  if (finished())
    return 0;
  if (enemy_turn())
    return -search(-beta, -alpha);
  for(move a : possible moves) {
    int p = gain(a);
    alpha >?= p + search(alpha-p, beta-p);
    if (alpha >= beta)
      break;
  }
  return alpha;
}
```

## 5.7 マトロイド交差

マトロイド交差の最大独立集合を求める。**M1, M2 について独立性判定ができること (M::appendable)。M2 について、独立な集合に要素を加えて非独立になったとき、存在する一意の回路を計算できること (M2::circuit)。**

```
template<class E, class M1, class M2>
set<E> augment(M1 m1, M2 m2, set<E> g, set<E> s)
     {
  map<E, E> trace;

  vector<E> rights;
  FOR(i, g) {
    if (m1.appendable(s, *i)) {
      rights.push_back(*i);
      trace[*i] = *i;
    }
  }
  while(!rights.empty()) {
    vector<E> lefts;
    REP(i, rights.size()) {
      E e = rights[i];
      if (m2.appendable(s, e)) {
        while(trace[e] != e) {
          s.insert(e); e = trace[e];
          s.erase(e); e = trace[e];
        }
        s.insert(e);
        return s;
      }
      vector<E> c = m2.circuit(s, e);
      REP(j, c.size()) {
        E f = c[j];
        if (trace.count(f) == 0) {
          trace[f] = e;
          lefts.push_back(f);
        }
      }
    }
    rights.clear();
    REP(i, lefts.size()) {
      E f = lefts[i];
      s.erase(f);
      FOR(i, g) {
        E e = *i;
        if (m1.appendable(s, e) && trace.count(e)
            == 0) {
          trace[e] = f;
          rights.push_back(e);
        }
      }
      s.insert(f);
    }
  }
  return s;
}
```

## 5.8 Zeller の公式

```
int zeller(int y, int m, int d) {
  if (m <= 2) {
    y--;
    m += 12;
  }
  return (y + y/4 - y/100 + y/400 + (13 * m + 8)
       / 5 + d) % 7;
}
```

## 5.9 Grundy 数

move できなくなった時に負けるゲームについて、勝てるかどうかの判定と必勝法の計算を行う道具。盤面ひとつひとつに数字を割り当てる。後手必勝が 0、先手必勝が 1 以上。選択は mex。平行していくつかのゲームを進める場合は xor。

# 6 Tips

## 6.1 std::string

```
// コンストラクタ/代入
string();
string(const basic_string& s, size_type pos = 0,
     size_type n = npos);
string(const char*);
string(const char* s, size_type n);
string(size_type n, char c);
string(InputIterator first, InIter last);

// 挿入
void insert(iterator pos, InIter f, InIter l);
string& insert(size_type pos, const string& s);
string& insert(size_type pos, const char* s);
string& append(const string& s);
string& append(size_type n, char c);
string& append(InIter first, InIter last);
void push_back(char c);

// 削除
iterator erase(iterator p);
iterator erase(iterator first, iterator last);
string& erase(size_type pos = 0, size_type n =
    npos);
void clear();
void resize(size_type n, char c = char());

// 置換
string& replace(size_type pos, size_type n,
    const string& s);
string& replace(size_type pos, size_type n,
    size_type n1, char c);
string& replace(iterator first, iterator last,
    const string& s);
string& replace(iterator first, iterator last,
    size_type n, char c);
string& replace(iterator first, iterator last,
    InIter f, InIter l);

// 切り出し
basic_string substr(size_type pos = 0, size_type
     n = npos);

// 検索
size_type find(const basic_string& s, size_type
    pos = 0);
size_type find(const char* s, size_type pos,
    size_type n);
size_type find(const char* s, size_type pos =
    0);
size_type find(char c, size_type pos = 0);
size_type find_first_of(const basic_string& s,
    size_type pos = 0);
size_type find_first_of(const char* s, size_type
     pos, size_type n);
size_type find_first_of(const char* s, size_type
     pos = 0);
size_type find_first_of(char c, size_type pos =
    0);
```

## 6.2 レアな algorithm たち

```
void nth_element(RandIter first, RandIter nth,
    RandIter last, Ordering comp);
bool includes(InIter1 first1, InIter1 last1,
    InIter2 first2, InIter2 last2, Ordering comp
    );
OutIter set_difference(InIter1 first1, InIter1
    last1, InIter2 first2, InIter2 last2,
   OutIter result, Ordering comp);
OutIter set_intersection(InIter1 first1, InIter1
     last1, InIter2 first2, InIter2 last2,
   OutIter result, Ordering comp);
OutIter set_symmetric_difference(InIter1 first1,
     InIter1 last1, InIter2 first2, InIter2
    last2,
   OutIter result, Ordering comp);
OutIter set_union(InIter1 first1, InIter1 last1,
     InIter2 first2, InIter2 last2,
   OutIter result, Ordering comp);
void inplace_merge(BidirIter first, BidirIter
    middle, BidirIter last, Ordering comp);
OutIter merge(InIter1 first1, InIter1 last1,
    InIter2 first2, InIter2 last2,
   OutIter result, Ordering comp);
ForIter partition(ForIter first, ForIter last,
    Predicate pred);
ForIter stable_partition(ForIter first, ForIter
    last, Predicate pred);
```