Tiny KTH ACM Contest Template Library

tinyKACTL version 2007-03-01

TOKYO EDITION

Contents

Chapter 1: Contest

Listing 1.1: Template.cc

23 lines, <cmath>, <cstdio>, <algorithm>, <string>, <map>, <vector>

```
// Contest, Location, Date
//
// Template for KTH-NADA, Team Name
//   Team Captain, Team Member, Team Member
//
// Problem: ___

using namespace std;
const enum {SIMPLE, FOR, WHILE} mode = NO;
#define dprintf debug && printf
bool debug = false;

void init() {
  // when reading from/writing to files
  freopen("X.in", "r", stdin);
  freopen("X.out", "w", stdout);
  // rebinding cin/cout requires include <fstream>
  cin.rdbuf((new ifstream("Y.in"))->rdbuf());
  cout.rdbuf((new ofstream("Y.out"))->rdbuf());
}

bool solve(int P) {
}

int main() {
  init();
  int n = mode == SIMPLE ? 1 : 1<<30;
  if (mode == FOR) scanf("%d", &n);
  for (int i = 0; i < n && solve(i); ++i);
  return 0;
}
```

Listing 1.2: script.cc

9 lines,

```
echo 'g++ -ansi -Wall -o $1 $1.cc' > c
echo 'cat > $1.in' > i
echo 'cat > $1.ans' > o
echo 'diff $1.out $1.ans' > d
echo 'a2ps --line-numbers=1 $1' > p
echo 'cp Template.cc $1.cc' > n

chmod +x c i o d p n
```

Listing 1.3: contest-keys.el.cc

26 lines,

```
(defun kactl-compile () (interactive)
  (shell-command (concat "g++ -ansi -lm -Wall -o "
                         (file-name-sans-extension buffer-file-name)
                         " " buffer-file-name)))

(defun kactl-new-file (N) (interactive "FCFF: ")
  (find-file N) (or (file-exists-p N)
                    (not (string-equal (file-name-extension N) "cc"))
                    (insert-file "Template.cc")))

(defun kactl-test () (interactive)
  (let ((N (file-name-sans-extension buffer-file-name)))
    (shell-command (concat N " < " N ".in &"))))
;; This line replaces the above two lines on input
;; from file instead of stdin
;;(shell-command (file-name-sans-extension buffer-file-name)))

(defun kactl-send () (interactive)
  (and (string-equal (file-name-extension buffer-file-name) "cc")
       (y-or-n-p "Send? ") (shell-command
                            (concat "submit " buffer-file-name))))

(global-set-key "\C-x\C-f" 'kactl-new-file)
(global-set-key "\C-cc" 'kactl-compile)
(global-set-key "\C-ct" 'kactl-test)
(global-set-key "\C-cs" 'kactl-send)
```

Listing 1.4: contest-extras.el.cc

23 lines,

```
(defun kactl-print () (interactive)
  (shell-command (concat "a2ps --line-numbers=1 "
                         buffer-file-name) " &"))

(defun kactl-diff () (interactive)
  (let ((N (file-name-sans-extension buffer-file-name)))
    (shell-command
     (concat N " < " N ".in > " N
             ".temp && diff " N ".out " N ".temp &"))))

(global-set-key "\C-cp" 'kactl-print)
(global-set-key "\C-cd" 'kactl-diff)
(global-set-key "\C-cg" 'goto-line)

# .xmodmap stuff
# copy |<> to Caps Lock
remove Lock = Caps_Lock
keycode 0x40 = less greater bar

# move Alt Graph to Right Meta
remove Mod4 = Meta_R
keycode 0xEE = Mode_switch
add Mod2 = Mode_switch
```

Chapter 2: Useful mathematical identities

2.1 Equations

$$\begin{array}{l} ax+by=e \\ cx+dy=f \end{array} \Rightarrow \begin{array}{l} x=\frac{ed-bf}{ad-bc} \\ y=\frac{af-ec}{ad-bc} \end{array}$$

2.2 Trigonometry

$$\begin{aligned}
\sin(v+w) &= \sin v\cos w + \cos v \sin w \\
\cos(v+w) &= \cos v\cos w - \sin v \sin w \\
\tan(v+w) &= \frac{\tan v + \tan w}{1-\tan v \tan w} \\
\sin v + \sin w &= 2\sin\frac{v+w}{2}\cos\frac{v-w}{2} \\
\cos v + \cos w &= 2\cos\frac{v+w}{2}\cos\frac{v-w}{2} \\
(V+W)\tan(v-w)/2 &= (V-W)\tan(v+w)/2
\end{aligned}$$

where V is the length of the side opposite the angle v, and similar for W.

$$\left\{ \begin{array}{lcl} a\cos x + b\sin x &=& r\cos(x-\phi) \\ a\sin x + b\cos x &=& r\sin(x+\phi) \end{array} \right.$$

where $r=\sqrt{a^2+b^2}$, $\phi = \arctan(b,a)$.

2.2.1 Spherical trigonometry

a, b, c = sides, α, β, γ = angles, all six anglemeasured and less than π.

$$\begin{aligned}
\cos a &= \cos b\cos c + \sin b \sin c \cos\alpha \\
\cos\alpha &= -\cos\beta\cos\gamma + \sin\beta\sin\gamma\cos a \\
\sin\alpha/\sin a &= \sin\beta/\sin b = \sin\gamma/\sin c
\end{aligned}$$

2.3 Geometry

2.3.1 Triangles

Side lengths a, b, c.
Semiperimeter $p = \frac{a+b+c}{2}$.
Area $A = \sqrt{p(p-a)(p-b)(p-c)}$.
Circumradius $R = \frac{abc}{4A}$.
Inradius $r = \frac{A}{p}$.
Median (divides triangle into two equal-sized triangles) $m_a = \frac{1}{2}\sqrt{2b^2+2c^2-a^2}$.

Bisector (divides angle in two) $s_a = \sqrt{bc\left[1-\left(\frac{a}{b+c}\right)^2\right]}$.

$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c} = \frac{1}{2R}$$

$$a^2 = b^2+c^2-2bc\cos\alpha$$

$$\frac{a+b}{a-b} = \frac{\tan\frac{\alpha+\beta}{2}}{\tan\frac{\alpha-\beta}{2}}$$

2.3.2 Quadrilaterals

Side lengths a, b, c, d.
Diagonals $e(ad \leftrightarrow bc), f(ab \leftrightarrow cd)$.
Diagonal angle θ.
Magic flux $F = b^2+d^2-a^2-c^2$.
Area $4A = 2ef\sin\theta = F\tan\theta = \sqrt{4e^2f^2-F^2}$.

2.3.3 Regular Polyhedra

Vertices v.
Edges e. Faces $f = e-v+2$. Volume $V = k_V a^3$.
Surface area $S = k_S a^2$.
Circumradius $R = k_R a$.
Inradius $r = k_r a$.

Tetrahedron ($v=4, e=6$)

$$k_V = \frac{\sqrt{2}}{12} \qquad k_S = \sqrt{3}$$
$$k_R = \frac{\sqrt{6}}{4} \qquad k_r = \frac{\sqrt{6}}{12}$$

Octahedron ($v=6, e=12$)

$$k_V = \frac{\sqrt{2}}{3} \qquad k_S = 2\sqrt{3}$$
$$k_R = \frac{1}{\sqrt{2}} \qquad k_r = \frac{1}{\sqrt{6}}$$

Dodecahedron ($v=20, e=30$)

$$k_V = \frac{15+7\sqrt{5}}{4} \qquad k_S = 3\sqrt{5(5+2\sqrt{5})}$$
$$k_R = \frac{(1+\sqrt{5})\sqrt{3}}{4} \qquad k_r = \frac{\sqrt{10+22/\sqrt{5}}}{4}$$

Icosahedron ($v=12, e=30$)

$$k_V = \frac{5(3+\sqrt{5})}{12} \qquad k_S = 5\sqrt{3}$$
$$k_R = \frac{\sqrt{2(5+\sqrt{5})}}{4} \qquad k_r = \frac{1}{2}\sqrt{\frac{7+3\sqrt{5}}{6}}$$

2.4 Derivatives/Integrals

$$\arcsin x \to \frac{1}{\sqrt{1-x^2}} \qquad \arccos x \to -\frac{1}{\sqrt{1-x^2}}$$
$$\tan x \to 1+\tan^2 x \qquad \arctan x \to \frac{1}{1+x^2}$$
$$\int \tan ax = -\frac{\ln|\cos ax|}{a} \qquad \int x\sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$

Chapter 3: Data Structures

Listing 3.1: sets.cc

27 lines, <vector>

```
struct sets {
  struct set_elem {
    int h, rank; // rank is a pseudo-height with height“¡=rank
    set_elem(int elem) : h(elem), rank(0) {}
  };
  vector<set_elem> elems;

  sets(int nElems) {
    for(int i = 0; i < nElems; i++) elems.push_back(set_elem(i));
  }

  int get_head(int i) { // Find set-head with path-compression
    if (i != elems[i].h) elems[i].h = get_head(elems[i].h);
    return elems[i].h;
  }

  bool equal(int a, int b){ return (get_head(a)==get_head(b)); }

  void link(int a, int b) { // union sets
    a = get_head(a); b = get_head(b);
    if(elems[a].rank > elems[b].rank) elems[b].h = a;
    else {
      elems[a].h = b;
      if(elems[a].rank == elems[b].rank) elems[b].rank++;
    }
  }
};
```

Listing 3.2: suffix array.cc

79 lines, <cstring>

```
struct suffix_array {
  int length, *suffixes, *position, *count;
  char *text, *border;

  suffix_array(int maxlen):
    length(0), suffixes(new int[maxlen]),
    position(new int[maxlen]), count(new int[maxlen]),
    border(new int[maxlen]) {}

  void set_text(char *_text) {
    text = _text;
    length = strlen(text);
    sort_suffixes();
  }

  void sort_init() {
    int pos[257], i;
    char *p;

    memset(pos, 0, sizeof(pos));
    for(p = text; p < text + a->length; ++p) ++pos[*p+1];

    for(int i = 1; i < 256; ++i) {
      if((pos[i] += pos[i-1]) >= a->length) break;
      border[pos[i]] = 1;
    }
    *border = 1;

    for(p = text; p < text + length; ++p)
      suffixes[pos[(int) *p]++] = p - text;
    return 1;
  }

  void sort_suffixes() {
   int H, i, N = length;
   memset(border, 0, N);

   for(H = sa_sort_init(); H < length; H *= 2) {
     int left = 0, done = 1;

     for(i = 0; i < N; ++i) {
       if(border[i]) left = i;
       position[suffixes[i]] = left;
       count[i] = 0;
     }

     left = 0;
     for( i = 0; i < N; ++i) {
       int suff = suffixes[i];
       if(suff >= H) {
         position[suff - H] += count[position[suff - H]]++;
         border[position[suff - H]] |= 2;
       }
       if(suff >= N - H) {
         position[suff] += count[position[suff]]++;
         border[position[suff]] |= 2;
       }

       if(i == N - 1 || (border[i+1] & 1)) {
         for( ; left <= i; ++left) {
           suff = suffixes[left] - H;
           if(suff < 0 || !(border[position[suff]] & 2))
            continue;
           suff = position[suff];
           for (++suff; suff < N && (border[suff]^2) == 0; ++suff)
             border[suff] &= ~2;
         }
       }
     }

     for(i = 0; i < N; ++i) {
       suffixes[position[i]] = i;
       done &= (border[i] = !!border[i]);
     }

     if (done) break;
   }
  }
};
```

Chapter 4: Numerical

4.1 Numerical datastructures

Listing 4.1: bigint.cc

150 lines, <iostream>, <iomanip>, <string>, <vector>

```
/* if long longs are disallowed:
 * #define LSIZE 10000
 * #define LIMBDIGS 4
 * typedef int limb;    */
#define LSIZE 1000000000 /* 10^9 */
#define LIMBDIGS 9

typedef long long limb;
typedef vector<limb> bigint;
typedef bigint::const_iterator bcit;
typedef bigint::reverse_iterator brit;
typedef bigint::const_reverse_iterator bcrit;
typedef bigint::iterator bit;

bigint BigInt(limb i) {
  bigint res;
  do res.push_back(i % LSIZE);
  while (i /= LSIZE);
  return res;
}

istream& operator>>(istream& i, bigint& n) {
  string s; i >> s;
  int l = s.length();
  n.clear();
  while (l > 0) {
    int j = 0;
    for (int k = l > LIMBDIGS ? l-LIMBDIGS: 0; k < l; ++k)
      j = 10*j + s[k]-'0';
    n.push_back(j);
    l -= LIMBDIGS;
  }
  return i;
}

/* Warning: the ostream must be configured to print things
 * with right justification, lest output be ooky */
ostream& operator<<(ostream& o, const bigint& n) {
  int began = 0, ofill = o.fill();
  o.fill('0');
  for (bcrit i = n.rbegin(); i != n.rend(); ++i) {
    if (began) o << setw(LIMBDIGS);
    if (*i) began = 1;
    if (began) o << *i;
  }
  if (!began) o << "0";
  o.fill(ofill);
  return o;
}

/* The base comparison function. semantics like strcmp(...) */
int cmp(const bigint& n1, const bigint& n2) {
  int x = n2.size() - n1.size();
  bcit i = n1.end() - 1, j = n2.end() - 1;
  while (x-- > 0) if (*j--) return -1;
  while (++x < 0) if (*i--) return 1;
  for (; i + 1 != n1.begin(); --i, --j)
    if (*i != *j) return *i-*j;
  return 0;
}

/* The other operators will be automatically defined by STL */
bool operator==(const bigint& n1, const bigint& n2) {
  return !cmp(n1,n2); }
bool operator<(const bigint& n1, const bigint& n2) {
  return cmp(n1,n2) < 0; }

bigint& operator+=(bigint& a, const bigint& b) {
  if (a.size() < b.size()) a.resize(b.size());
  limb cy = 0; bit i = a.begin();
  for (bcit j = b.begin(); i != a.end() &&
         (cy || j < b.end()); ++j, ++i)
    cy += *i + (j < b.end() ? *j : 0),
      *i = cy % LSIZE, cy /= LSIZE;
  if (cy) a.push_back(cy);
  return a;
}

bool sub(bigint& a, const bigint& b) { /* Ret sign changed */
  if (a.size() < b.size()) a.resize(b.size());
  limb cy = 0; bit i = a.begin();
  for (bcit j = b.begin(); i != a.end() &&
         (cy || j < b.end()); ++j, ++i) {
    *i -= cy + (j < b.end() ? *j : 0);
    if ((cy = *i < 0)) *i += LSIZE;
  } /* Line below only if sign may change. */
  if (cy) while (i-- > a.begin()) *i = LSIZE - *i;
  return cy;
}

bigint& operator-=(bigint& a, const bigint& b) {
  sub(a, b); return a; }

bigint& operator*=(bigint& a, limb b) {
  limb cy = 0;
  for (bit i = a.begin(); i != a.end(); ++i)
    cy = cy / LSIZE + *i * b, *i = cy % LSIZE;
  while (cy /= LSIZE) a.push_back(cy % LSIZE);
  return a;
}

bigint& operator*=(bigint& a, const bigint& b) {
  bigint x = a, y, bb = b;
  a.clear();
  for (bcit i = bb.begin(); i != bb.end(); ++i)
    (y = x) *= *i, a += y, x.insert(x.begin(), 0);
  return a;
}

/* a will hold floor(a/b), rest will hold a % b */
bigint& divmod(bigint& a, limb b, limb* rest = NULL) {
  limb cy = 0;
  for (brit i = a.rbegin(); i != a.rend(); ++i)
    cy += *i, *i = cy / b, cy = (cy % b) * LSIZE;
  if (rest) *rest = cy / LSIZE;
  return a;
}

/* returns a, holding a % b, quo will hold floor(a/b).
 * NB!! different semantics from one-limb divmod!!
 * NB!! quo should be different from a!! */
bigint& divmod(bigint& a, const bigint& b, bigint* quo=NULL) {
  bigint den = b;
  brit j = den.rbegin(), i = a.rbegin();
  for ( ; j != den.rend() && !*j; ++j);
  for ( ; i != a.rend() && !*i; ++i);
  int n = a.rend() - i, m = den.rend() - j;
  if (!m) { /* Division by zero! */ abort(); }
  if (m == 1) {
    bigint q;
    return (quo ? *quo : q) = a, a.resize(1),
      divmod(quo ? *quo : q, *j, &a.front()), a;
  }

  bigint tmp;
  limb den0 = (*++j + *--j * LSIZE) + 1;
  if (quo) quo->clear();
  while (a >= den) { /* Loop invariant: quo * den + a = num */
    limb num0 = (*++i + *--i * LSIZE), z = num0 / den0, cy = 0;
    if (z == 0 && n == m) z = 1;/* Silly degenerate case */
    tmp.resize(n - m - !z);
    if (!z) z = num0 / (*j + 1);/* Non-silly degenerate case*/
    if (quo) tmp.push_back(z), *quo += tmp, tmp.pop_back();
    for (bcit j = den.begin(); j != den.end(); ++j)
      cy += *j * z, tmp.push_back(cy % LSIZE), cy /= LSIZE;
    if (cy) tmp.push_back(cy);
    if (tmp.size() > a.size()) tmp.resize(a.size());
    sub(a, tmp);
    while (i != a.rend() && !*i) --n, ++i;
  }
  return a;
}

bigint& operator/=(bigint& a, const bigint& b) {
  bigint q; return divmod(a, b, &q), a = q; }

bigint& operator%=(bigint& a, const bigint& b) {
  return divmod(a, b, NULL); }

bigint& operator/=(bigint& a, limb b) { return divmod(a, b); }
limb operator%(const bigint& a, limb b) {
  limb res;
  bigint fubar = a;
  return divmod(fubar, b, &res), res;
}
```

Listing 4.2: sign.cc

36 lines,

```
template <class T>
struct sign {
  static const T zero;
  T x; bool n;
  operator sign(T _x = zero, bool _n = false): x(_x), n(_n) {}
  bool operator <(const sign &s) const {
    return n==s.n ? n ? x>s.x : x<s.x : n && !(x==zero&&s.x== \
zero);
  }
  bool operator ==(const sign &s) const {
    return n==s.n ? x==s.x : x==zero&&s.x==zero;
  }
  sign operator -() { return sign(x, !n); }
  sign &as(bool add) {
    if (add) x+=s.x;
    else if (x<s.x) { T t=s.x; x = t-=x; n = !n; }
    else x-=s.x;
    return *this;
  }
  sign &operator+=(const sign &s){return as(n == s.n); }
  sign &operator-=(const sign &s){return as(n != s.n); }
  sign &operator*=(const sign &s){x*=s.x,n^=s.n; return *this;}
  sign &operator/=(const sign &s){x/=s.x,n^=s.n; return *this;}
};

template <class T>
sign<T> abs(const sign<T> &s) { return sign<T>(s.x, false); }

template <class T>
istream &operator >>(istream &in, sign<T> &s) {
  char c; in>>c; s.n = c=='-'; if (!s.n) in.unget(); in>>s.x;
}

template <class T>
ostream &operator <<(ostream &out, const sign<T> &s) {
  if (s.n && s.x != s.zero) out << '-'; out << s.x;
}
```

Listing 4.3: rational.cc

81 lines, "gcd.cpp"

```
template <class T>
struct rational {
  typedef rational<T> rT;
  typedef const rT & R;
  T n, d;
  rational(T _n=T(), T _d=T(1)) : n(_n), d(_d) { normalize(); }
  void normalize() {
    T f = gcd(n, d); n /= f; d /= f;
    if (d < T()) n *= -1, d *= -1;
  }
  bool operator < (R r) const { return n * r.d < d * r.n; }
  bool operator ==(R r) const { return n * r.d == d * r.n; }

  rT operator -() { return rT(-n, d); }

  rT operator +(R r) { return rT(n*r.d + r.n*d, d*r.d); }
  rT operator -(R r) { return rT(n*r.d - r.n*d, d*r.d); }
  rT operator *(R r) { return rT(n*r.n, d*r.d); }
  rT operator /(R r) { return rT( n*r.d,/**/d*r.n); }
  T/**//**/ div(R r) { return/**/(n*r.d) / (d*r.n); }
  rT operator %(R r) { return rT((n*r.d) % (d*r.n), d*r.d); }

  rT operator <<(int b) { return b<0 ? a>>-b : rT(n<<b, d); }
  rT operator >>(int b) { return b<0 ? a<<-b : rT(n, d<<b); }

  ostream &print_frac(ostream &out) {
    out << n; if (d != T(1)) out << '/' << d;
    return out;
  }

  istream &read_frac(istream &in) {
    in >> n;
    if (in.peek()=='/') { char c; in >> c >> d; } else d = T(1);
    normalize();
    return in;
  }
};

template <class T>
ostream &print_dec(ostream &out, const rational<T> &r,
                   int precision = 15, int radix = 10) {
  T n = r.n, d = r.d;
  if (n < T()) out << '-', n *= -1;
  out << n/d; n %= d;
  if (T() < n) {
    out << '.';
    for (int i = 0; n && i < precision; ++i) {
      n *= radix;
      out << n/d; n %= d;
    }
  }
  return out;
}

template <class T>
ostream &operator<<(ostream &out, const rational<T> &r) {
  return print_dec(out, r); //or return r.print_frac(out);
}

template <class T>
istream &read_dec(istream &in, rational<T> &r) {
  T i, f(0), z(1);
  in >> i;
  if (in.peek() == '.') {
    char c; in >> c;
    while (in.peek() == '0') { in >> c; z *= 10; }
    if (in.peek() >= '0' && in.peek() <= '9') in >> f;
  }
  r.d = T(1);
  while (r.d <= f) r.d *= 10;
  r.d *= z;
  r.n = i*r.d + f;
  r.normalize();
  return in;
}

template <class T>
istream& operator>>(istream &in, rational<T> &r) {
  return read_dec(in, r); // or return r.read_frac(in);
}
```

Listing 4.4: polynomial.cc

23 lines, <vector>

```
struct polynomial {
  int n;
  vector<double> a;
  polynomial(int _n): n(_n), a(n+1) {}

  double operator()(double x) const { // Calc value at x
    double val = 0;
    for(int i = n; i >= 0; --i) (val *= x) += a[i];
    return val;
  }

  void diff() { // differentiate
    for (int i = 1; i <= n; ++i) a[i-1] = i*a[i];
    a.pop_back(); --n;
  }

  void divroot(double x0) { // divide by (x-x0), ignore remainder
    double b = a.back(), c;
    a.back() = 0;
    for (int i = n--; i--; ) c = a[i], a[i] = a[i+1]*x0+b, b = c;
    a.pop_back();
  }
};
```

4.2 Equation solving

Listing 4.5: solve linear.cc

51 lines,

```
const double undefined = 1.0/0.0;
const double eps = 1e-12;

// Solves A*x = b, or as much of x as possible. Returns rank.
// Data in A and b is lost.
template <int N> int
solve_linear(int n, double A[N][N], double b[N], double x[N]) {
  int row[N], col[N], undef[N], invrow[N], invcol[N], rank = 0;
  for (int i = 0; i < n; ++i)
    row[i] = col[i] = i, undef[i] = false;

  for (int i = 0; i < n; rank = ++i) {
    int br = i, bc = i;
    double v, bv = abs(A[row[i]][col[i]]);
    for (int r = i; r < n; ++r)
      for (int c = i; c < n; ++c)
        if ((v = abs(A[row[r]][col[c]])) > bv)
          br = r, bc = c, bv = v;
    if (bv < eps) break;
    if (i != br) row[i] ^= row[br] ^= row[i] ^= row[br];
    if (i != bc) col[i] ^= col[bc] ^= col[i] ^= col[bc];
    for (int j = i + 1; j < n; ++j) {
      double fac = A[row[j]][col[i]] / bv;
      A[row[j]][col[i]] = 0;
      b[row[j]] -= fac * b[row[i]];
      for (int k = i + 1; k < n; ++k)
        A[row[j]][col[k]] -= fac * A[row[i]][col[k]];
    }
  }

  for (int i = rank; i-- ; ) {
    b[row[i]] /= A[row[i]][col[i]];
    A[row[i]][col[i]] = 1;
    for (int j = rank; j < n; ++j)
      if (abs(A[row[i]][col[j]]) > eps)
```

```
        undef[i] = true;
    for (int j = i - 1; j >= 0; --j) {
      if (undef[i] && abs(A[row[j]][col[i]]) > eps)
        undef[j] = true;
      else {
        b[row[j]] -= A[row[j]][col[i]] * b[row[i]];
        A[row[j]][col[i]] = 0;
      }
    }
  }

  for (int i = 0; i < n; ++i)
    invrow[row[i]] = i, invcol[col[i]] = i;
  for (int i = 0; i < n; ++i)
    if (invrow[i] >= rank || undef[invrow[i]]) b[i] = undefined;
  for (int i = 0; i < n; ++i) x[i] = b[row[invcol[i]]];
  return rank;
}
```

Listing 4.6: matrix inverse.cc

49 lines,

```
template <int N>
bool matrix_inverse(double A[N][N], int n) {
  bool singular = false;
  int row[N], col[N];
  double tmp[N][N] ;
  memset(tmp, 0, sizeof(tmp));

  for (int i = 0; i < n; ++i)
    tmp[i][i] = 1, row[i] = col[i] = i;

  for (int i = 0; i < n; ++i) { // forward pass:
    int r = i, c = i; // find pivot
    for (int j = i; j < n; ++j)
      for (int k = i; k < n; ++k)
        if (fabs(A[row[j]][col[k]]) > fabs(A[row[r]][col[c]]))
          r = j, c = k;
    if (fabs(A[row[r]][col[c]]) < 1e-12) // pivot found?
      return false; // if singular
    if (i != r) row[i] ^= row[r] ^= row[i] ^= row[r]; // pivot
    if (i != c) col[i] ^= col[c] ^= col[i] ^= col[c];
    double v = A[row[i]][col[i]]; // eliminate forward
    for (int j = i+1; j < n; ++j) {
      double f = A[row[j]][col[i]] / v;
      A[row[j]][col[i]] = 0;
      for (int k = i+1; k < n; ++k)
        A[row[j]][col[k]] -= f*A[row[i]][col[k]];
      for (int k = 0; k < n; ++k)
        tmp[row[j]][col[k]] -= f*tmp[row[i]][col[k]];
    } // normalize row
    for (int j = i+1; j < n; ++j) A[row[i]][col[j]] /= v;
    for (int j = 0; j < n; ++j) tmp[row[i]][col[j]] /= v;
    A[row[i]][col[i]] = 1;
  }

  for (int i = n-1; i > 0; --i) // backward pass:
    for (int j = i-1; j >= 0; --j) {
      double v = A[row[j]][col[i]];
      // forget A at this point, just eliminate tmp backward
      for (int k = 0; k < n; ++k)
        tmp[row[j]][col[k]] -= v*tmp[row[i]][col[k]];
    }

  int invcol[n];
  for (int i = 0; i < n; ++i) invcol[col[i]] = i;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      A[i][j] = tmp[row[invcol[i]]][j];
  return true;
}
```

Listing 4.7: poly roots.cc

28 lines, "polynomial.cpp"

```
const double eps = 1e-8;

void poly_roots(const polynomial& p, double xmin, double xmax,
                vector<double>& roots) {
  if (p.n == 1) { roots.push_back(-p.a.front()/p.a.back()); }
  else {
    polynomial d = p;
    vector<double> droots;
    d.diff();
    poly_roots(d, xmin, xmax, droots);
    droots.push_back(xmin-1);
    droots.push_back(xmax+1);
    sort(droots.begin(), droots.end());
    for (vector<double>::iterator i = droots.begin(), j = i++;
         i != droots.end(); j = i++) {
      double l = *j, h = *i, m, f;
      bool sign = p(l) > 0;
      if (sign ^ p(h) > 0) {
        while (h - l > eps) {
          m = (l + h) / 2, f = p(m);
          if (f <= 0 ^ sign) l = m;
          else h = m;
        }
        roots.push_back((l + h) / 2);
      }
    }
  }
}
```

4.2.1 Calculating determinant

`determinant` and `int_determinant` both reduces the matrix to an upper diagonal form using elementary row operations. There could be an overflow in the integral variant and in that case the double variant can be used instead, rounding the answer at the end. The strength of `int_determinant` is that it can be used for `long long` or `BigInt`. Note that it uses `euclid` which could be rather slow in the `BigInt` case.

Listing 4.8: determinant.cc

24 lines,

```
template< int N >
double determinant( double m[N][N], int n ) {
  for( int c=0; c<n; c++ ) {
    for( int r=c; r<n; r++ ) {
      if( abs(m[r][c]) >= 1e-8 ) {
        if( r!=c ) {   // Eliminate column c with row r
          for( int j=0; j<n; j++ ) {
            swap( m[c][j], m[r][j] );
            m[r][j] = -m[r][j];
          }
        }
        for( r++; r<n; r++ ) {
          double mul = m[r][c]/m[c][c];
          for( int j=0; j<n; j++ )
            m[r][j] -= m[c][j]*mul;
        }
      }
    }
  }
  // Matrix is now in upper-diagonal form
  double det = 1;
  for( int i=0; i<n; i++ ) det *= m[i][i];
  return det;
}
```

Listing 4.9: int determinant.cc

30 lines, "euclid.cpp"

```
template< class T, int N >
T int_determinant( T m[N][N], int n ) {
  for( int c=0; c<n; c++ ) {
    for( int r=c; r<n; r++ ) {
      if( m[r][c] !=0 ) {
        if( r!=c ) {          // Eliminate column c with row r
          for( int j=0; j<n; j++ ) {
            swap( m[c][j], m[r][j] );
            m[r][j] = -m[r][j];
          }
        }
        for( r++; r<n; r++ ) {
          T x,y;
          T d = euclid( m[c][c], m[r][c], x,y );
          T x2 = -m[r][c]/d, y2 = m[c][c]/d;

          for( int j=c; j<n; j++ ) {
            T u = x*m[c][j]+y*m[r][j];
            T v = x2*m[c][j]+y2*m[r][j];
            m[c][j] = u; m[r][j] = v;
          }
        }
      }
    }
  }
  // Matrix is now in upper-diagonal form
  T det = 1;
  for( int i=0; i<n; i++ ) det *= m[i][i];
  return det;
}
```

Listing 4.10: sqrt.cc

11 lines,

```
// Newton-Raphson's method. y' = y-(y^2-x)/(2y) = (y+x/y)/2.
// ans should contain a guess >= actual answer, e.g. x0/2.
template< class T >
void sqrt(const T &x0, T &ans) {
  T x = x0;
  if (x0 > 1) {
    while(true) {
      T d = x0; d /= ans; d += ans; d /= 2;
      if(!(d < ans)) break;
      ans = d;
    }
  }
}
```

4.3 Optimization

4.3.1 Simplex method

Solves a linear minimization problem. The first row of the input matrix is the objective function to be minimized. The first column is the maximum allowed value for each linear row.

Listing 4.11: simplex.cc

57 lines,

```
enum simplex_res { OK, UNBOUNDED, NO_SOLUTION };

template <class M, class I>
simplex_res simplex(M &a, I &var, int m, int n, int twophase=0) {
  while (true) {
    // Choose a variable to enter the basis
    int idx = 0;
    for (int j = 1; j <= n; ++j)
      if (a[0][j] > 0 && (idx == 0 || a[0][j] > a[0][idx]))
        idx = j;
    // Done if all a[m][j]<=0
    if (idx == 0) return OK;
    // Find the variable to leave the basis
    int j = idx; idx = 0;
    for (int i = 1; i <= m; ++i)
      if (a[i][j] > 0 &&
          (idx == 0 || a[i][0]/a[i][j] < a[idx][0]/a[idx][j]))
        idx = i;
    // Problem unbounded if all a[i][j]<=0
    if (idx == 0) return UNBOUNDED;
    // Pivot on a[i][j]
    int i = idx;
    for (int l = 0; l <= n; ++l)
      if (l != j) a[i][l] /= a[i][j];
    a[i][j] = 1;
    for (int k = 0; k <= m + twophase; ++k)
      if (k != i) {
        for (int l = 0; l <= n; ++l)
          if (l != j) a[k][l] -= a[k][j] * a[i][l];
        a[k][j] = 0;
      }
    // Keep track of the variable change
    var[i] = j;
  }
}

template <class M, class I>
simplex_res twophase_simplex(M &a, I &var, int m, int n,
                             int artificial) {
  // Save primary objective, clear phase I objective
  for (int j = 0; j <= n + artificial; ++j)
    a[m + 1][j] = a[0][j], a[0][j] = 0;
  // Express phase I objective in terms of non-basic variables
  for (int i = 1; i <= m; ++i)
    for (int j = n + 1; j <= n + artificial; ++j)
      if (a[i][j] == 1)
        for (int l = 0; l <= n; ++l)
          if (l != j) a[0][l] += a[i][l];
  simplex(a, var, m, n + artificial, 1); // Simplex phase I
  // Check solution
  for (int j = n + 1; j <= n + artificial; ++j)
    if (a[0][j] >= 0) return NO_SOLUTION;
  // Restore primary objective
  for (int j = 0; j <= n; ++j)
    a[0][j] = a[m + 1][j];
  return simplex(a, var, m, n); // Simplex phase II
}
```

4.3.2 Conjugated gradient method

Search direction in step i is $d_i = g_i + \beta_i d_{i-1}$, where g_i is the gradient in step i and $\beta_i = \frac{|g_i|^2}{|g_{i-1}|^2}$ $(d_1 = g_1)$.

4.3.3 Knapsack Heuristic

Complexity $\mathcal{O}(\min(bound, nC))$

Note! Exact if bound $\geq nC$

Listing 4.12: knapsack.cc

24 lines, <vector>

```
/* Templates:
 * R is the value type (needs to be constructable from "-1").
 * T is the cost type (needs to be multipliable with doubles).
 * W is a random access container of costs.
 * V is a random access container of values.
 */
template <class R, class T, class W, class V>
R knapsack(int n, const T& C, const W& costs, const V& values,
           int bound=500000) {
  double scale = bound / ((double) n * C);
  // Removed if the costs are all small-valued doubles.
  if (scale > 1) scale = 1;
  int C_max = (int) (scale * C) + 1;
  R max = R();
  vector<R> val_max(C_max, R(-1));
  val_max[0] = R();

  for (int i = 0; i < n; ++i) {
    int scaled_cost = (int) (scale * costs[i]);
    for (int j = C_max - 1; j >= scaled_cost; --j) {
      R v = val_max[j - scaled_cost];
      if (v != -1 && v + values[i] > val_max[j]) {
        val_max[j] = v + values[i];
        if (val_max[j] > max)
          max = val_max[j];
      }
    }
  }
  return max;
}
```

4.3.4 Hopcroft

Listing – hopcroft.cc, p. 7

Complexity $\mathcal{O}(Q^2\Sigma)$

Automata state minimization.

Listing 4.13: hopcroft.cc

40 lines, <set>

```
template <class T>
int count(T s, int Q) { // Count the number of bits in s
  int c(0); T zero(0), one(1); // Alternatively:
  for (int x = 0; x < Q; ++x) // use int bit manipulation
    if ((s & one << x) != zero) // or bitset<N>::count()!
      c++;
  return c;
}

template <class T, class F> // Hopcroft state minimization.
set<T> hopcroft(T q, T f, // all states q, finishing states f
                int S, int Q, // number of symbols S and states Q
                F &d) { // function d(state, symbol) -> next states
  set<T> w, p;
  w.insert(f); w.insert(q & ~f); // work list
  p.insert(f); p.insert(q & ~f); // initial partition
  while (!w.empty()) { // pick a set of states,
    T s = *w.begin(); w.erase(w.begin()); // s
    for (int a = 0; a < S; ++a) {
      T i(0), zero(0), one(1); // find the set of states i,
      for (int x = 0; x < Q; ++x) // that can reach a state in s,
        if ((s & d[x][a]) != zero) // given the symbol a
          i |= one << x;
      typename set<T>::iterator it = p.begin();
      while (it != p.end()) { // refine the partition around i
        T r = *it; T r1 = r & i, r2 = r & ~i;
        if (r1 != zero && r2 != zero) {
          if (w.count(r) == 0) // put either min{r1, r2}
            w.insert(count(r1, Q) < count(r2, Q) ? r1 : r2);
          else // or both{r1, r2} in worklist
            w.erase(r), w.insert(r1), w.insert(r2);
          p.erase(it++), p.insert(r1), p.insert(r2);
        }
        else
          ++it;
      }
    }
  }
  return p; // returns a state partition
}
/*Pseudo-code:
W,P <- {F,Q-F}
while W not empty
  take S from W
  for a in Symb
    I_a=deltainv_a(S)
    for R in P
      R1=R&I_a;R2=R&~I_a;
      R->{R1,R2} in P
      R->{R1,R2} in W OR min{R1,R2} to W, if R is not in W
*/
// if used with bitset<N>:
// operator < required by set, must be _in_ namespace std
```

Chapter 5: Number theory

5.1 Primality

5.1.1 Primes

1000th prime is 7919. First every 10000th primes are:

```
104729  224737  350377  479909  611953
746773  882377 1020379 1159523
```

Listing 5.1: prime sieve.cc

44 lines, <algorithm>, <cmath>

```
/* Prime sieve for generating all primes up to a certain n.
 * Decently fast for n up to ~50 millions. */
struct prime_sieve {
  typedef unsigned char uchar;
  typedef unsigned int uint;
  static const int pregen = 3*5*7*11*13;
  uint n, sqrtn;
  uchar *isprime;
  int *prime, primes; // prime[i] is i:th prime

  bool is_prime(int n) { // primality check
    if (n % 2 == 0 || n <= 2) return n == 2;
    return isprime[n-3 >> 4] & 1 << (n-3 >> 1 & 7);
  }

  prime_sieve(int _n): n(_n), sqrtn((int)ceil(sqrt(1.0*n))) {
    int n0 = max(n >> 4, (uint)pregen);
    prime = new int[max(2775,(int)(1.12*n/log(1.0*n)))];
    prime[0] = 2; prime[1] = 3; prime[2] = 5;
    prime[3] = 7; prime[4] = 11; prime[5] = 13;
    primes = 6;
    isprime = new uchar[n0];
    memset(isprime, 255, n0);

    for (int j = 1, p = prime[j]; j < 6; p = prime[++j])
      for (int i = p*p-3 >> 4, s = p*p-3 >> 1 & 7;
           i <= pregen; i += (s += p) >> 3, s &= 7)
        isprime[i] &= ~(1 << s);

    for (int d = pregen, b = pregen+1; b < n0; b += d, d <<= 1)
      memcpy(isprime + b, isprime + 1, (n0 < b + d) ? n0-b : d);

    for (uint p=17, i=0, s=7; p<n; p+=2, i += ++s>>3, s &= 7)
      if (isprime[i] & 1 << s) {
        prime[primes++] = p;
        if (p < sqrtn) {
          int ii = i, ss = s + (p-1)*p/2;
          for (uint pp = p*p; pp < n; pp += p<<1, ss += p) {
            ii += ss >> 3;
            ss &= 7;
            isprime[ii] &= ~(1 << ss);
          }
        } // end if
      } // end if
  } // end constructor
};
```

Listing 5.2: miller-rabin.cc

16 lines, "expmod.h", "mulmod.h"

```
template <class T> bool miller_rabin(T n, int tries = 15) {
  T n1 = n - 1, m = 1;
  int j, k = 0;
  if (n <= 3) return true;
  while (!(n1 & (m << k))) ++k;
  m = n1 >> k;
  for (int i = 0; i < tries; ++i) {
    T a = rand() % n1, b = expmod(++a, m, n);
    if (b == T(1)) continue;
    for (j = 0; j < k && b != n1; ++j)
      b = mulmod(b, b, n);
    if (j == k) return false;
  }
  return true;
}
```

5.2 Divisibility

Listing 5.3: euclid.cc

5 lines,

```
template <class Z> Z euclid(Z a, Z b, Z &x, Z &y) {
  if (b) { Z d = euclid(b, a % b, y, x);
           return y -= a/b * x, d; }
  return x = 1, y = 0, a;
}
```

5.2.1 Chinese remainder theorem

Solves the system $x = a \pmod{m}$, $x = b \pmod{n}$. `chinese` returns the unique solution with $0 \leq x < \mathrm{lcm}(m, n)$. If $\gcd(m, n) = 1$, `chinese` may be used, otherwise, `chinese_common` must be used, which returns -1 if there is no solution.

Listing 5.4: chinese.cc

12 lines, "euclid.cpp"

```
template <class Z> inline Z chinese(Z a, Z m, Z b, Z n) {
  Z x, y; euclid(m, n, x, y);
  return (a * n * (y < 0 ? y + m : y) +
         b * m * (x < 0 ? x + n : x)) % (m*n);
}

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

Listing 5.5: pollard-rho.cc

25 lines, "gcd and mulmod functions"

```
template <class T> inline T pollard_step(T x, T N) {
  T r = mulmod(x, x, N);
  if (++r == N) r = 0;
  return r; /* r = x^2+1 (mod N) */
}

/* Returns a non-trivial factor of N. (Note that if N is
 * prime, pollard_rho never returns) */
template <class T> T pollard_rho(T N, int maxiter = 500) {
  T x = random() % N, y = x, d;
  int iter = 0;
  if (!(N & 1)) return 2; /* Check factor 2 */
  /* Small factor check. */
  for (d = 3; d <= 997; d += 2)
    if (N % d == 0) return d;

  for (d = 1; d == 1; ) {
    /* Reseed if too many iterations passed. */
    if (iter++ == maxiter)
      x = y = random() % N, iter = 0;
    x = pollard_step(x, N);
    y = pollard_step(pollard_step(y, N), N);
    d = gcd(y - x, N);
    if (d == N) d = 1;
  }
  return d;
}
```

5.2.2 Perfect numbers

n is perfect iff $n = \frac{p(p+1)}{2}$, where $p = 2^k - 1$ is prime. First Mersenne primes are obtained for $k = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701$.

Chapter 6: Combinatorial

6.1 Permutations

6.1.1 Permutations to/from integers

Complexity $\mathcal{O}\left(n^2\right)$

Note! The bijection is order-preserving

Listing 6.1: intperm.cc

21 lines, <algorithm>

```
int factorial[] = {1, 1, 2, 6, 24, 120, 720, 5040, etc...};

template <class Z, class It>
void perm_to_int(Z& val, It begin, It end) {
  int x = 0, n = 0;
  for (It i = begin; i != end; ++i, ++n)
    if (*i < *begin) ++x;
  if (n > 2) perm_to_int<Z>(val, ++begin, end);
  else val = 0;
  val += factorial[n-1]*x;
}

/* range [begin, end) does not have to be sorted. */
template <class Z, class It>
void int_to_perm(Z val, It begin, It end) {
  Z fac = factorial[end - begin - 1];
  // Note that the division result will fit in an integer!
  int x = val / fac;
  nth_element(begin, begin + x, end);
  swap(*begin, *(begin + x));
  if (end - begin > 2) int_to_perm(val % fac, ++begin, end);
}
```

6.1.2 Derangements

$$
\begin{aligned}
D_n &= (n-1)(D_{n-1}+D_{n-2}) = nD_{n-1} + (-1)^n \\
&= n!\left(\frac{1}{2!}-\frac{1}{3!}+\ldots+(-1)^n\frac{1}{n!}\right) = \left\lfloor\frac{n!}{e}\right\rceil
\end{aligned}
$$

derangements.cc generates the i:th (lexicographically) derangement of S_n.

Listing 6.2: derangements.cc

43 lines, <cstring>

```
template <class T, int N>
struct derangements {
  T dgen[N][N], choose[N][N], fac[N];
  derangements() {
    fac[0] = choose[0][0] = 1;
    memset(dgen, 0, sizeof(dgen));
    for (int m = 1; m < N; ++m) {
      fac[m] = fac[m-1] * m;
      choose[m][0] = choose[m][m] = 1;
      for (int k = 1; k < m; ++k) {
        choose[m][k] = choose[m-1][k-1] + choose[m-1][k];
      }
    }
  }
  T DGen(int n, int k) {
    T ans = 0;
    if (dgen[n][k]) return dgen[n][k];
    for (int i = 0; i <= k; ++i) {
      T t = choose[k][i] * fac[n-i];
      if (i & 1) ans -= t;
      else ans += t;
    }
    return dgen[n][k] = ans;
  }
  void generate(int n, T idx, int *res) {
    int vals[N];
    for (int i = 0; i < n; ++i) vals[i] = i;
    for (int i = 0; i < n; ++i) {
      int j, k = 0, m = n - i;
      for (j = 0; j < m; ++j)
        if (vals[j] > i) ++k;
      for (j = 0; j < m; ++j) {
        T l = 0;
        if (vals[j] > i)  l = DGen(m-1, k-1);
        else if (vals[j] < i) l = DGen(m-1, k);
        if (idx <= l) break;
        idx -= l;
      }
      res[i] = vals[j];
      memmove(vals + j, vals + j + 1, sizeof(int)*(m-j-1));
    }
  }
};
```

6.1.3 Involutions

An involution is a permutation with maximum cycle length 2, or equivalently, a permutation which is its own inverse. The number of involutions on $[n]$ is given by

$$s(n) = s(n-1) + (n-1)s(n-2) \qquad s(0) = s(1) = 1$$

6.1.4 Stirling numbers of the first kind

$s(n,k)$ is defined as $(-1)^{n-k}c(n,k)$, where $c(n,k)$ is the number of permutations on n items with k cycles.

$$s_{n,k} = s_{n-1,k-1} - (n-1)s_{n-1,k}$$

$$s_{n,k} = 1, n = k \qquad s_{n,k} = 0, n < 1$$

6.1.5 Eulerian numbers

The Eulerian number $e_{n,k}$ is the number of $\pi \in S_n$ with

- k j:s s.t. $\pi(j) > \pi(j+1)$
- $k+1$ j:s s.t. $\pi(j) \geq j$
- k j:s s.t. $\pi(j) > j$

$$
\begin{aligned}
e_{n,k} &= (n-k)e_{n-1,k-1} + (k+1)e_{n-1,k} \\
&= \sum_{j=0}^{k+1}(-1)^j\binom{n+1}{j}(k-j+1)^n
\end{aligned}
$$

$$e_{n,k} = 1, n = k = 0 \qquad e_{n,k} = 0, n < 1 \vee n = k \neq 0$$

6.1.6 Second-order Eulerian numbers

The second-order Eulerian number e_{nk} is the number of permutations $\pi_1\pi_2\cdots\pi_{2n}$ of the multiset $\{1,1,2,2,\cdots,n,n\}$ with the property that all numbers between the two occurences of m are greater than m that have k places where $\pi_j < \pi_{j+1}$. It is given by

$$e_{n,k} = (2n-1-k)e_{n-1,k-1} + (k+1)e_{n-1,k}$$

$$e_{n,k} = 1, n = k = 0 \qquad e_{n,k} = 0, n < 1 \vee n = k \neq 0$$

6.1.7 Married Couples Problem

In how many ways can n couples be seated around a round table such that men and women are alternated and such that no couple sits next to each other?

$$a_{n+1} = \frac{(n^2-1)a_n + (n+1)a_{n-1} + 4(-1)^n}{n-1}$$

for $n \geq 4$, with $a_0 = a_3 = 1$, $a_1 = a_2 = 0$. The first numbers are 1, 0, 0, 1, 2, 13, 80, 579, 4738, 43387, 439792, 4890741, 59216642.

6.2 Partitions and subsets

6.2.1 Binomial $\binom{n}{k}$

Complexity $\mathcal{O}(\min\{k, n-k\})$

Listing 6.3: choose.cc

11 lines, <algorithm>

```
template <class T>
T choose(int n, int k) {
  k = max(k, n-k);

  T c = 1;
  for (int i = 1; i <= n-k; ++i)
    c *= k+i, c /= i;

  return c;
}
```

6.2.2 Multinomial $\binom{\Sigma k_i}{k_1\, k_2\, \ldots\, k_n}$

Complexity $\mathcal{O}((\Sigma k_i) - k_1)$

Listing 6.4: multinomial.cc

10 lines,

```
template <class T, class V>
T multinomial(int n, V &k) {
  T c = 1;
  int m = k[0];
  for (int i = 1; i < n; ++i)
    for (int j = 1; j <= k[i]; ++j)
      c *= ++m, c /= j;

  return c;
}
```

6.2.3 Stirling numbers of the second kind

Partitions of n elements in exactly k boxes.

$$s_{n,k} = s_{n-1,k-1} + k s_{n-1,k}$$

$$s_{n,k} = 1, n = k \qquad s_{n,k} = 0, n < 1$$

6.2.4 Bell numbers

$B(n) = \sum_{k=1}^{n} \binom{n-1}{k-1} B(n-k) = \sum_{k=1}^{n} S(n,k)$, where $S(n,k)$ are the Stirling numbers of the second kind.

The Bell numbers count the ways n distinct elements can be partitioned. The first are 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597.

6.2.5 Partitions of non-distinct elements

The number of partitions of n non-distinct elements is

$$p(n) = \sum_{k \geq 1} (-1)^{k+1} \left[p\left(n - \frac{k(3k-1)}{2}\right) + p\left(n - \frac{k(3k+1)}{2}\right)\right]$$

for $n \geq 1$, and $p(n) = [n == 0]$ for $n < 1$.

The number of partitions into distinct, odd parts is

$$q(n) = \frac{1}{n} \sum_{k=1}^{n} (-1)^{k+1} \sigma(k) q(n-k)$$

where $\sigma(k)$ is the sum of the *odd* divisors of k (e.g. $\sigma(9) = 13$). If $(-1)^{k+1}$ is removed, we get the number of partitions into distinct parts.

6.2.6 Triangles

Given rods of length $1, \ldots, n$,

$$T(n) = \frac{1}{24} \begin{cases} n(n-2)(2n-5) & n \text{ even} \\ (n-1)(n-3)(2n-1) & n \text{ odd} \end{cases}$$

distinct triangles (positive area) can be constructed (in other words, the number of 3-subsets $\{x, y, z\}$ of $[n]$ s.t. $x < y < z$ and $z \neq x + y$).

6.3 General purpose numbers

6.3.1 Catalan numbers

$$C_n = \frac{2(2n-1)C_{n-1}}{n+1} = \frac{\binom{2n}{n}}{n+1}$$

The first Cataln numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900.

6.3.2 Super Catalan numbers

The Super Catalan numbers count the number of lattice paths with diagonal steps from (n,n) to $(0,0)$ which do not touch the diagonal line $x = y$. Also number of ways to insert parentheses in a string of n symbols. The parentheses must be balanced but there is no restriction on the number of pairs of parentheses. The number of letters inside a pair of parentheses must be at least 2. Parentheses enclosing the whole string are ignored.

$$S_n = \frac{3(2n-3)S_{n-1} - (n-3)S_{n-2}}{n}$$

With $S_1 = S_2 = 1$. The first are 1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, 518859, 2646723.

6.3.3 Motzkin Numbers

Counts for instance: number of ways of drawing any number of nonintersecting chords among n points on a circle, number of lattice paths from $(0,0)$ to $(n,0)$ never going below the x axis and using only steps NE, E and SE, number of $(3412, 2413)$-, $(3412, 3142)$-, and $(3412, 3412)$-avoiding involutions in S_n.

$$M_n = \frac{3(n-1)M_{n-2} + (2n+1)M_{n-1}}{n+2}$$

with $M_0 = M_1 = 1$. The first Motzkin numbers are 1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, 5798, 15511, 41835, 113634.

Chapter 7: Graph

7.1 Fundamentals

7.1.1 Bellman-Ford

Complexity $\mathcal{O}(VE)$

Listing 7.1: bellman ford.cc

31 lines,

```
template <class E, class M, class P, class D>
bool bellman_ford(E &edges, M &min, P &path,
                  int start, int n, int m) {
  typedef typename M::value_type T;
  T inf(1<<29);

  for (int i = 0; i < n; i++) {
    min[i] = inf;
    path[i] = -1;
  }
  min[start] = T();

  bool changed = true;
  for (int i = 1; changed; ++i) { // V-1 times
    changed = false;
    for (int j = 0; j < m; ++j) {
      int node = edges[j].first.first;
      int dest = edges[j].first.second;
      T dist = min[node] + edges[j].second;

      if (dist < min[dest]) {
        if( i>=n )
          return false; // negative cycle!
        min[dest] = dist;
        path[dest] = node;
        changed = true;
      }
    }
  }
  return true; // graph is negative-cycle-free
}
```

7.1.2 Shortest Tour

Shortest tour from A to B to A again not using any edge twice, in an undirected graph: Convert the graph to a directed graph. Take the shortest path from A to B. Remove the paths used from A to B, but also negate the lengths of the reverse edges. Take the shortest path again from A to B, using an algorithm which can handle negative-weight edges, such as Bellman-Ford. Note that there is no negative-weight *cycles*. The shortest tour has the length of the two shortest paths combined.

7.1.3 Kruskal

Usage `kruskal( graph, tree, n );`

Complexity $\mathcal{O}(E \log E)$

NB! Requires sets.cc! The resulting tree which is returned in `tree` may be the same variable as the graph.

Listing – sets.cc, p. 3

Listing 7.2: kruskal.cc

26 lines, <algorithm>, <vector>, "sets.cpp"

```
template <class G>
void kruskal(const G *graph, G *tree, int n) {
  typedef typename G::const_iterator  G_iter;
  typedef typename G::value_type::second_type D;
  sets sets(n);
  vector<pair< D,pair<int,int> > > E;

  // Convert all edges into a single edge-list
  for( int i=0; i<n; i++ )
    for(G_iter it = graph[i].begin(); it!=graph[i].end(); ++it)
      if (i < it->first)//Undirected:only use half of the edges
        E.push_back(make_pair(it->second,
                              make_pair(i, it->first)));
  sort(E.begin(), E.end());
  for(int i=0; i<n; i++) tree[i].clear();

  int numEdges = edges.size();
  for(int i = 0; i < numEdges; ++i) {
    pair<int,int> &e = E[i].second;
    if(!sets.equal(e.first, e.second)) {
      sets.link(e.first, e.second);
      tree[e.first].push_back(make_pair(e.second, E[i].first));
      tree[e.second].push_back(make_pair(e.first, E[i].first));
    }
  }
}
```

7.1.4 Directed Minimum Spanning Tree

Chu-Liu/Edmonds Algorithm:

1. Discard the arcs entering the root if any; For each node other than the root, select the entering arc with the smallest cost; Let the selected $n-1$ arcs be the set S.
2. If no cycle formed, $G(N,S)$ is a MST. Otherwise, continue.
3. For each cycle formed, contract the nodes in the cycle into a pseudo-node (k), and modify the cost of each arc which enters a node (j) in the cycle from some node (i) outside the cycle according to the following equation:

   $$c(i,k) = c(i,j) - (c(x(j),j) - min_j(c(x(j),j))$$

   where $c(x(j),j)$ is the cost of the arc in the cycle which enters j.
4. For each pseudo-node, select the entering arc which has the smallest modified cost; Replace the arc which enters the same real node in S by the new selected arc.
5. Go to step 2 with the contracted graph.

Listing 7.3: topo sort.cc

20 lines, <vector>, <queue>

```
template <class E, class I>
bool topo_sort(const E *edges, I &idx, int n) {
  typedef typename E::const_iterator E_iter;
  vector<int> indeg(n);
  for (int i = 0; i < n; i++)
    for (E_iter e = edges[i].begin(); e != edges[i].end(); e++)
      indeg[*e]++;
  queue<int> q; // use priority_queue for lexic. smallest ans.
  for (int i = 0; i < n; i++)
    if (indeg[i] == 0) q.push(-i);
  int nr = 0;
  while (q.size() > 0) {
    int i = -q.front(); // top() for priority_queue
    idx[i] = ++nr;
    q.pop();
    for (E_iter e = edges[i].begin(); e != edges[i].end(); ++e)
      if (--indeg[*e] == 0) q.push(-*e);
  }
  return nr == n;
}
```

7.2 Euler walk

7.2.1 Euler walk

Complexity $\mathcal{O}(E)$

The algorithm *assumes* that there exists an eulerian walk. If it does not exists, it will return any maximal path, not neccessarily the longest. If the graph is not cyclic, the start node must be a node with $\deg_{out} - \deg_{in} = 1$. Set `cyclic=true` if the path found must be cyclic, this is mostly of internal use.

`edges` is a vector/array with V edge-containers. The edge-containers should contain vertex-indices, and may contain repeated indices (i.e. multiple edges). **WARNING!** `edges` is modified and emptied by the algorithm.

`path` should be empty prior to the call and contains the euler-path given as *vertex numbers*. The first vertex is `start` which also is the last vertex if the path is cyclic.

Lexicographic Path If the edges are sorted in lexicographic order for each vertex, the resulting path will be lexicographically ordered.

Listing 7.4: euler walk.cc

35 lines, <list>

```
template<class V>
void euler_walk(V &edges, int start, list<int> &path,
                bool cyclic = false) {
  int node = start, next_node;
  // Find a maximal path
  while(true) {
    typename V::value_type &s = edges[node];
    path.push_back(node);
    if(s.empty()) break;
    // Follow the first edge and remove it
    next_node = *s.begin();
    s.erase( s.begin() );
    node = next_node;
  }

  // If no cyclic path was found, return "empty" path.
  if(cyclic && node != start) {
    path.clear();
    path.push_back( node );
    return;
  }

  // Extend path with cycles
  for(list<int>::iterator it=--path.end(); it!=path.begin();) {
    list<int>::iterator iter2 = iter; iter2--;
    node = *iter;
    typename V::value_type &s = edges[node];
    while(!s.empty()) {
      list<int> add;
      euler_walk(edges, node, add, true /* must be cyclic */);
      path.splice(it, add, add.begin(), --add.end() );
    }
    it = iter2;
  }
}
```

7.2.2 Chinese postman

A generalised euler path/cycle problem, finding the shortest path/cycle that visits all edges even if some edges have to be traversed several times. There are several variations to this problem, e.g. for directed or undirected graphs, paths or cycles, or whether just a subset of the edges are interesting (the latter variations are called rural chinese postman, and are generally NP-complete).

Undirected chinese postman can be solved by computing a minimum weight perfect matching on the odd nodes of the graph.

Directed chinese postman can be solved by computing a minimum weight perfect bipartite matching between the positive and negative nodes of the graph. If a node v has surplus (outvalency - invalency) k, k copies of this node occurs in the bipartite graph. **NB!** The bipartite graph may have $2E$ vertices!

7.2.3 De Bruijn Sequences

Complexity $\mathcal{O}(N^L)$

Usage `deBruijn( int N, int L, char symbols[N] )`

Let Ω be an alphabet of size σ. A de Bruijn sequence is a sequence such that all words on L letters appear as a contiguous subrange of it. The shortest cyclic de Bruijn sequence is of length σ^L and the shortest non-cyclic de Bruijn sequence is of length $\sigma^L + L - 1$.

The lexicographically smallest, and shortest de Bruijn sequence on three letters and alphabet $\{0,1\}$ is 00011101 (cyclic), 0001110100 (non-cyclic).

`N` is the size of the alphabet and `symbols` the corresponding letters. `L` is the length of the words that should appear in the de Bruijn sequence.

The answer is stored in the `sol` string.

Listing 7.5: deBruijn fast.cc

48 lines,

```
template<class V>
void euler_walk_dB(V &edges, int start, list<int> &path,
                   int nSymb, int nNodes) {
  int node = start;
  while( true ) {
    int &s = edges[node];
    path.push_back( node );
    if( s == 0 ) break;
    for( int i=0; i<nSymb; i++ )
      if( s & (1<<i) ) {
        node = (node*nSymb)%nNodes + i;
        s ^= (1<<i);
        break;
      }
  }

  for(list<int>::iterator it=--path.end(); it!=path.begin();) {
    list<int>::iterator iter2 = iter; iter2--;
    node = *iter;
    int &s = edges[node];
    while( s != 0 ) {
      list<int> add;
      euler_walk_dB(edges, node, add, nSymb, nNodes);
      path.splice(iter, add, add.begin(), --add.end());
    }
    iter = iter2;
  }
}


void deBruijn(int nSymb, int L, char *symbols, char *sol) {
  int         nNodes = 1;
  vector<int> edges;
  list<int>  p;
  for(int i = 0; i < L-1; ++i) nNodes *= nSymb;
  edges.reserve(nNodes);
  for(int i = 0; i < nNodes; ++i)
    edges.push_back((1<<nSymb) - 1);

  euler_walk_dB(edges, 0, p, nSymb, nNodes);
  for(list<int>::iterator it = p.begin(); it != p.end(); ++it)
    if(iter == p.begin())
      for(int j=0; j<L-1; j++)
        *sol++ = symbols[*iter % nSymb];
    else
      *sol++ = symbols[*iter % nSymb];
  *sol = 0;
}
```

7.3 Network Flow

Listing 7.6: flow graph.cc

23 lines, <vector>

```
typedef int Flow;

struct flow_edge {
  int dest, back;// back is index of back-edge in graph[dest]
  Flow c, f; // capacity and flow
  Flow cost; //used by bellman ford
  Flow r() { return c - f; } // used by ford fulkerson
  flow_edge() {}
  flow_edge(int _dest, int _back, Flow _c, Flow _f = 0
            /*, Flow _cost = 0*/)
    : dest(_dest), back(_back), c(_c), f(_f)/*, cost(_cost)*/ { }
};

typedef vector<flow_edge> adj_list;
typedef adj_list::iterator adj_iter;

void flow_add_edge(adj_list *g, int s, int t, // add s -> t
                   Flow c, Flow back_c = 0/*, Flow cost = 0*/) {
  g[s].push_back(flow_edge(t, g[t].size(), c/*, 0, cost*/));
  g[t].push_back(flow_edge(s, g[s].size() - 1, back_c
                           /*, 0, cost*/));
}

#define MAXNODES 6000
```

7.3.1 Lift to Front

Complexity $\mathcal{O}\left(V^3\right)$

Listing 7.7: lift to front.cc

41 lines, "flow_graph.cpp"

```
void add_flow(adj_list *g, flow_edge &e, Flow f, Flow *exc) {
  flow_edge &back = g[e.dest][e.back];
  e.f += f; e.c -= f; exc[e.dest] += f;
  back.f -= f; back.c += f; exc[back.dest] -= f;
}

Flow lift_to_front(adj_list *g, int v, int s, int t) {
  int l[MAXNODES], hgt[MAXNODES]; // l == list, hgt == height
  Flow exc[MAXNODES]; // exc == excess
  adj_iter cur[MAXNODES];

  memset(hgt, 0, sizeof(int)*v);
  memset(exc, 0, sizeof(Flow)*v);
  hgt[s] = v - 2;
  for (adj_iter it = g[s].begin(); it != g[s].end(); it++)
    add_flow(g, *it, it->c, exc);
  int p = t; // make l a linked list from p to t (sink)
  for (int i = 0; i < v; i++) {
    if (i != s && i != t) l[i] = p, p = i;
    else l[i] = t;
    cur[i] = g[i].begin();
  }

  int r = 0, u = p; // lift-to-front loop
  while (u != t) {
    int oldheight = hgt[u];
    while (exc[u] > 0) // discharge u
      if (cur[u] == g[u].end()) {
        hgt[u] = 2 * v - 1; // lift u, find admissible edge
        for (adj_iter it = g[u].begin(); it!=g[u].end(); ++it)
          if (it->c > 0 && hgt[it->dest] + 1 < hgt[u])
            hgt[u] = hgt[it->dest]+1, cur[u] = it;
      } else if (cur[u]->c>0 && hgt[u] == hgt[cur[u]->dest]+1)
        add_flow(g, *cur[u], min(exc[u], (*cur[u]).c), exc);
      else ++cur[u];
    if (hgt[u] > oldheight && p != u) // lift-to-front!
      l[r] = l[u], l[u] = p, p = u; // u to front of list
    r = u, u = l[r];
  }
  return exc[t];
}
```

7.3.2 Ford Fulkerson

DFS and BFS Ford Fulkerson implementations. Min-cost-max-flow returns the minimal cost for the maximal flow.

Complexity $\mathcal{O}\left(E \cdot n_{aug\ paths}\right)$

Listing 7.8: ford fulkerson.cc

68 lines, <queue>, "flow_graph.cpp"

```
int mark[MAXNODES];

Flow inc_flow_dfs(adj_list *g, int s, int t, Flow maxf) {
  if (s == t) return maxf;
  Flow inc; mark[s] = 0;
  for (adj_iter it = g[s].begin(); it != g[s].end(); ++it)
    if (mark[it->dest] && it->r() &&
        (inc=inc_flow_dfs(g,it->dest,t,min(maxf, it->r()))))
      return it->f+=inc, g[it->dest][it->back].f -= inc, inc;
  return 0;
}

Flow inc_flow_bfs(adj_list *g, int s, int t, Flow inc) {
  queue<int> q; q.push(s);
  while (!q.empty() && mark[t] < 0) {
    int v = q.front(); q.pop();
    for (adj_iter it = g[v].begin(); it != g[v].end(); ++it)
      if (mark[it->dest] < 0 && it->r())
        mark[it->dest] = it->back, q.push(it->dest);
  }
  if (mark[t] < 0) return 0;
  flow_edge* e; int v = t;
  while (v != s)
    e = &g[v][mark[v]], v = e->dest, inc<?=g[v][e->back].r();
  v = t;
  while (v != s)
    e = &g[v][mark[v]], e->f -= inc,
      v = e->dest, g[v][e->back].f += inc;
  return inc;
}

Flow mindist[MAXNODES], inf = 1<<28;

// For min-cost-max-flow. NB! Needs "cost" field in flow_edge!
Flow inc_flow_bellman_ford(adj_list *g, int n, int s, int t) {
  for(int i = 0; i < n; ++i)
    mindist[i] = inf;
  mindist[s] = 0;
  bool changed = true;
  for (int i = 1; !(changed = !changed); ++i)
    for (int v = 0; v < n; ++v)
      for (adj_iter it = g[v].begin(); it != g[v].end(); ++it) {
        Flow dist = mindist[v]+(it->f<0 ? -it->cost : it->cost);
        if (it->r() > 0 && dist < mindist[it->dest]) {
          if (i >= n) assert(0); // negative cycle! shouldn't be!
          mindist[it->dest] = dist;
          mark[it->dest] = it->back;
          changed = true;
        }
      }
  if (mark[t] < 0) return 0;
  Flow inc = inf;
  flow_edge* e; int v = t;
  while (v != s)
    e = &g[v][mark[v]], v = e->dest, inc<?=g[v][e->back].r();
  v = t;
  while (v != s)
    e = &g[v][mark[v]], e->f -= inc,
      v = e->dest, g[v][e->back].f += inc;
  return inc * mindist[t];
}

Flow max_flow(adj_list *graph, int n, int s, int t) {
  Flow flow = 0, inc = 0;
  do flow += inc, memset(mark, 255, sizeof(int)*n);
  while ((inc = inc_flow_dfs(graph, s, t, 1<<28)));
  return flow;//inc_flow_bfs(graph, s, t, 1<<28)
  //inc_flow_bellman_ford(graph, n, s, t)
}
```

7.3.3 Flow constructions

Minimal cut of a graph, generalization of edge connectivity. A minimal cut is found by first finding a maximal flow. Then we consider the set A of all nodes that can be reached from the source using edges which has capacity left (i.e. edges in the residue network). The edges between A and the complement of A is a minimal cut.

Minimal path cover of a DAG, determines a minimum set of disjoint paths to cover all vertices. For each $i \in V$, create two new vertices x_i and y_i and draw an edge between x_i and y_j if $(i, j) \in E$. Find a maximal bipartite matching, and if (x_i, y_j) is in the matching, let the edge (i, j) be part of a path.

7.4 Matching

7.4.1 hopcroft karp

Complexity $\mathcal{O}\left(\sqrt{V}E\right)$

Listing 7.9: hopcroft karp.cc

85 lines, <queue>, <vector>, <utility>

```
template< class M >
bool hk_recurse(int b, int *lPred, vector<int> *rPreds, M m_b) {
  vector<int> L;
  L.swap(rPreds[b]);
  for(unsigned i = 0; i < L.size(); ++i) {
    int a = L[i], b2 = lPred[a];
    lPred[a] = -2;
    if(b2 == -2) continue;
    if(b2 == -1 || hk_recurse(b2, lPred, rPreds, m_b)) {
      m_b[b] = a;
      return true;
    }
  }
  return false;
}


template< class G, class M, class T >
int hopcroft_karp(G g, int n, int m, M match_b,
                  T mis_a, T mis_b) {
  typedef typename G::value_type::const_iterator E_iter;

  int lPred[n];
  vector<int> rPreds[m];
  queue<int> leftQ, rightQ, unmatchedQ;
  bool rProc[m], rNextProc[m];

  for(int i = 0; i < m; i++)
    match_b[i] = -1;

  // Greedy matching (start)
  for(int i = 0; i < n; i++)
    for(E_iter e = g[i].begin(); e != g[i].end(); ++e)
      if(match_b[*e] < 0) {
        match_b[*e] = i;
        break;
      }

  while(true) {
    for(int i = 0; i < n; ++i)
      lPred[i] = -1; // i is in the first layer
    for(int j = 0; j < m; ++j)
      if(match_b[j] >= 0)
        lPred[match_b[j]] = -2;//remove from layer alltogether
    for(int j = 0; j < m; ++j)
      rPreds[j].clear(), rProc[j] = rNextProc[j] = false;
    for(int i = 0; i < n; ++i)
      if(lPred[i] == -1) leftQ.push(i);

    while(!leftQ.empty() && unmatchedQ.empty()) {
      while(!leftQ.empty()) {
        int a = leftQ.front(); leftQ.pop();
        for(E_iter e = g[a].begin(); e != g[a].end(); ++e )
          if(!rProc[*e]) {
            rPreds[*e].push_back(a);
            if(!rNextProc[*e])
              rightQ.push(*e), rNextProc[*e] = true;
          }
      }
      while(!rightQ.empty()) {
        int b = rightQ.front(); rightQ.pop();
        rProc[b] = true;
        if(match_b[b] >= 0)
          leftQ.push(match_b[b]), lPred[match_b[b]] = b;
        else unmatchedQ.push( b );
      }
    }

    while(!leftQ.empty()) leftQ.pop();

    if(unmatchedQ.empty()) { // No more alternating paths
      int nMatch = 0;
      for(int i = 0; i < n; ++i)
        mis_a[i] = lPred[i]>=-1;
      for(int j = 0; j < m; ++j)
        mis_b[j] = !rProc[j], nMatch += match_b[j]>=0;
      return nMatch;
    }

    while(!unmatchedQ.empty()) {
      int b = unmatchedQ.front(); unmatchedQ.pop();
      hk_recurse( b, lPred, rPreds, match_b );
    }
  }
}
```

7.4.2 Matching transformations

Max weight of maximum cardinality to max weight: Change each weight to $w \leftarrow \lfloor n/2 \rfloor (w_{max} - w_{min}) + w$.

Min weight of maximum cardinality to max weight: Change weights to $w \leftarrow (\lfloor n/2 \rfloor + 1)(w_{max} - w_{min}) - w$

7.4.3 max weight bipartite matching

Complexity $\mathcal{O}\left(V(E + V^2)\right)$

Listing 7.10: mwbm.cc

126 lines, <vector>

```
template< class E, class M, class W >
inline bool augment( E &edges, int a, int n, int m,
                     vector<W> &pot, vector<bool> &free,
                     vector<int> &pred, vector<W> &dist, M &match_b,
                     bool perfect ) {
  typedef typename E::value_type L;
  typedef typename L::const_iterator L_iter;

  vector<bool> proc(m, false);
  dist[a] = 0;
  pred[a] = a; // Start of alternating path
  int best_a = a, a1 = a, v;
  W minA = pot[a], delta;

  while( true ) {
    // Relax all edges out of a1
    for(L_iter e = edges[a1].begin(); e != edges[a1].end(); ++e){
      int b = n+e->first;
      if( match_b[b-n] == a1 )
        continue;
      W db = dist[a1] + (pot[a1]+pot[b]-e->second);
      if( pred[b] < 0 || db < dist[b] )
        dist[b] = db, pred[b] = a1;
    }

    // Select a node b with minimal distance db
    int b1 = -1;
    W db=0; // unused but makes compiler happy
    for( int b=n; b<n+m; b++ )
      if( !proc[b-n] && pred[b]>=0 && (b1<0 || dist[b]<db) )
        b1 = b, db = dist[b];

    if(b1 >= 0) proc[b1-n] = true;


    // End conditions
    if( !perfect && (b1<0 || db >= minA) ) {
      // Augment by path to best node in A
      delta = minA;
      free[a] = false; free[best_a] = true;// NB! Order important
      v = best_a;
      break;
    } else if( b1<0 ) {
      return false;
    } else if( free[b1] ) {
      // Augment by path to b
      delta = db;
      free[a] = free[b1] = false;
      v = b1;
      break;
    }

    // Continue shortest-path computation
    a1 = match_b[ b1-n ];
    pred[a1] = b1;
    dist[a1] = db;
    if( db+pot[a1] < minA )
      best_a = a1, minA = db+pot[a1];
  }

  // Augment path
  while( true ) {
    int vn = pred[v];
    if(v == vn) break;
    if(v >= n) match_b[v-n] = vn;
    v = vn;
  }

  for( int a=0; a<n; a++ ) {
    if( pred[a]>=0 ) {
      W dpot = delta - dist[a];
      pred[a] = -1;
      if(dpot > 0) pot[a] -= dpot;
    }
  }
  for( int b=n; b<n+m; b++ ) {
    if( pred[b]>=0 ) {
      W dpot = delta - dist[b];
      pred[b] = -1;
      if( dpot > 0 ) pot[b] += dpot;
    }
  }
  return true;
}

template< class E, class M, class W >
bool max_weight_bipartite_matching(E &edges, int n, int m,
                                   M &match_b, W &max_weight,
                                   bool perfect) {
  typedef typename E::value_type L;
  typedef typename L::const_iterator L_iter;

  vector<W> pot( n+m, 0 );
  vector<bool> free(n+m, true );
  vector<int> pred( n+m, -1 );
  vector<W> dist( n+m, 0 );

  for(int b=0; b<m; b++)
    match_b[b] = -1;

  // Initialize pot and matching with simple heuristics
  for( int a=0; a<n; a++ ) {
    int b = -1;
    W Cmax = 0;

    for(L_iter e = edges[a].begin(); e != edges[a].end(); ++e)
      if( b<0 || e->second > Cmax ||
          e->second==Cmax && free[n+e->first])
        b = n+e->first, Cmax = e->second;
    pot[a] = Cmax;
    if(b>=0 && free[b])
      match_b[b-n] = a, free[a] = free[b] = false;
  }

  // Augment matching
  for(int a=0; a<n; ++a)
    if(free[a])
      if(!augment(edges, a, n, m, pot, free, pred, dist,
              match_b, perfect))
        return false;

  max_weight = 0;
  for(int i=0; i < n+m; ++i) max_weight += pot[i];

  return true;
}
```

7.4.4 General weighted matching

Complexity $\mathcal{O}\left(V^3\right)$

Listing 7.11: weighted matching.cc

309 lines, <algorithm>

```
typedef int Weight;

/*
  todo:
  clean up unpair_all
  clean away secondmate
  try to understand the code better
*/

template <int maxV=50, int maxE=500, int asize = maxV+2*maxE+3>
struct matcher {
  int V, dummyVtx, dummyEdge;
Weight weight[asize], y[maxV+2], nextd[maxV+2], lastd, delta;
int a[asize], end[asize], nextpair[asize];
int mate[maxV+2], link[maxV+2], base[maxV+2];
int nextvtx[maxV+2], lastvtx[maxV+2], nextedge[maxV+2];
int lastedge[2], secondmate;

template <class AdjList> // NB!! 1-indexed vertices!!
int max_match(AdjList *graph, int _V) {
  typedef typename AdjList::iterator It;
  int E = (V = _V) | 1, v;

  memset(a, 0, sizeof(a));
  memset(lastedge, 0, sizeof(lastedge));
  memset(nextpair, 0, sizeof(nextpair));
  memset(nextvtx, 0, sizeof(nextvtx));
  lastd = secondmate = 0;

  for (int i = 1; i <= V; ++i) // init
    for (It it = graph[i].begin(); it != graph[i].end(); ++it)
      if (it->first > i) {
        E += 2;
        lastd >?= weight[E] = 4*it->second;
        end[E-1] = i;
        end[E] = it->first;

        // Isn't "a" just a linked list of the edges leading out of
        // a vertex, ordered by destination vertex??? Can it be
        // replaced by the actual adjacency list (possibly if we
        // sort the edges by vertex numbers first??)
        for (v = i; a[v] && end[a[v]] <= it->first; v = a[v]);
        a[E] = a[v];
        a[v] = E;
        for (v = it->first; a[v]; v = a[v]);
        a[v] = E-1;
      }
  dummyVtx = V+1;
  dummyEdge = E+1;

  end[dummyEdge] = dummyVtx;
  lastd /= 2;

  for (int i = 1; i <= V+1; ++i) { // more init...
    mate[i] = nextedge[i] = dummyEdge;
    link[i] = -dummyEdge;
    base[i] = lastvtx[i] = i;
    y[i] = nextd[i] = lastd;
  }

  for (int e, loop=1;; ++loop) { // solve!
    delta = 0;
    for (v = 1; v <= V; ++v)
      if (mate[v] == dummyEdge)
        pointer(dummyVtx, v, dummyEdge);

    for (e = 0; !e; ) {
      v = min_element(nextd+1, nextd+V+1) - nextd;
      if ((delta = nextd[v]) == lastd) goto done;

      int ne = nextedge[v];
      v = base[v];

      if (link[v] < 0) {
        int w = bmate(v);
        if (link[w] < 0) pointer(v, w, oppedge(ne));
        else unpair(v, w);
      } else e = pair(v);
    }
    lastd -= delta;
    set_bounds();
    rematch(bend(e), oppedge(e));
    rematch(bend(oppedge(e)), e);
  }

done:  // postprocess
  set_bounds();
  unpair_all();
  int ans = 0;
  for (int i = 1; i <= V; ++i)
    if (end[mate[i]] == dummyVtx) mate[i] = 0;
    else ans += weight[mate[i]|1], mate[i] = end[mate[i]];
  return ans / 8;
}

// Assign a pointer link to a vertex. Edge e joins a vertex in
// blossom u to a linked vertex.
void pointer(int u, int v, int e) {
  int i;
  Weight del;
  link[u] = -dummyEdge;
  nextbase(u) = nextbase(v) = dummyVtx;

  del = (lastvtx[u] == u) ? lastd : -slack(mate[nextvtx[u]])/2;
  for (int i = u; i != dummyVtx; i = nextvtx[i])
    y[i] += del, nextd[i] += del;
  if (link[v] < 0) {
    link[v] = e;
    nextpair[dummyEdge] = dummyEdge;
    scan(v, delta);
  }
  link[v] = e;
}

// Scan each vertex in the blossom whose base is x
int scan(int x, Weight del) {
  int newbase = base[x], stopscan = nextbase(x);
  for ( ; x != stopscan; x = nextvtx[x]) {
    int pairpt = dummyEdge, u;
    y[x] += del;
    nextd[x] = lastd;
    for (int e = a[x]; e; e = a[e])
      if (link[u = bend(e)] < 0) {
        if ((link[bmate(u)] < 0 || lastvtx[u] != u) &&
            nextd[end[e]] > slack(e))
          nextd[end[e]] = slack(e), nextedge[end[e]] = e;
      } else if (u != newbase) insert_pair(newbase, e, pairpt);
  }
  nextedge[newbase] = nextpair[dummyEdge];
  return newbase;
}

// Process an edge linking two linked vertices,
// v is the base of one end of the linking edge.
// Returns edge on success
int pair(int v) {
  int e = nextedge[v];
  while (slack(e) != 2*delta) e = nextpair[e];
  int w = bend(e), u = bmate(v);

  for (link[bmate(w)] = -e; link[u] != -e;
      v = base[end[link[v]]], u = bmate(v)) {
    link[u] = -e;
    if (mate[w] != dummyEdge) swap(v, w);
  }
  if (u == dummyVtx && v != w) return e;
  int newlast = v, oldfirst = nextvtx[v], i;
  link_path(v, e, newlast);
  link_path(v, oppedge(e), newlast);
  nextvtx[newlast] = oldfirst;
  if (lastvtx[v] == v) lastvtx[v] = newlast;
  nextpair[dummyEdge] = dummyEdge;
  merge_pairs(v, v);
  i = nextvtx[v];
  do {
    merge_pairs(v, i);
    i = nextbase(i);
    v = scan(i, 2*delta - slack(mate[i]));
    i = nextbase(i);
  } while (i != oldfirst);
  return 0;
}

//Merge a subblossom's pair list into a new blossom's pair list
// v is the base of the previously unlinked subblossom
// b is the base of the new blossom
// nextpair[dummyEdge] is the first edge on b's pair list
void merge_pairs(int b, int v) {
  int pairpt = dummyEdge, e = nextedge[v], ne = nextpair[e];
  nextd[v] = lastd;
  for ( ; e != dummyEdge; e = ne, ne = nextpair[e])
    if (bend(e) != b) insert_pair(b, e, pairpt);
}

// links the unlinked vertices in the path P(end[v], b)
// b is the base vertex of the new blossom
// newlast is the last vertex in b's current blossom
void link_path(int b, int e, int &newlast) {
  for (int v = bend(e); v != b; e = link[v], v = bend(e)) {
    int u = bmate(v);
    link[u] = oppedge(e);
    nextvtx[newlast] = v;
    nextbase(v) = u;
    newlast = lastvtx[u];
    for (int i=v; i != dummyVtx; base[i] = b, i = nextvtx[i]);
  }
}

// update a blossom's pair list.
// e is the edge to be inserted.
// pairpoint is the next pair on the pair list
void insert_pair(int b, int e, int &pairpoint) {
  int nextpoint = nextpair[pairpoint];
  while (end[nextpoint] < end[e])
    nextpoint = nextpair[pairpoint = nextpoint];
  if (end[nextpoint] == end[e]) {
    if (slack(e) >= slack(nextpoint)) return;
    nextpoint = nextpair[nextpoint];
  }
  nextpair[pairpoint] = e;
  nextpair[pairpoint = e] = nextpoint;
  nextd[b] <?= slack(e)/2;
}

// expands a blossom. fixes up link and mate
void unpair(int oldbase, int oldmate) {
  int e, newbase, u;
  unlink(oldbase);
  newbase = bmate(oldmate);
  if (newbase != oldbase) {
    link[oldbase] = -dummyEdge;
    if (rematch(newbase, mate[oldbase]) == lastedge[0])
         link[secondmate] = -lastedge[1]; // NB! rematch affects
```

```
      else link[secondmate] = -lastedge[0]; // secondmate!
    }
    u = bend(oppedge(e = link[oldmate]));
    if (u != newbase) {
      link[bmate(u)] = -e;
      for ( ; u != newbase; u = bend(e))
        e = -link[u], pointer(u, bmate(u), -link[bmate(u)]);
      e = oppedge(e);
    }
    pointer(newbase, oldmate, e);
  }

  // changes the matching along an alternating path
  // firstmate is the first base vertex on the path
  // edge e is the new matched edge for firstmate
  int rematch(int firstmate, int e) {
    mate[firstmate] = e;
    int eopp = 0;
    for (int ne = -link[firstmate]; ne != dummyEdge ; ) {
      firstmate = bend(e = ne);
      secondmate = bend(eopp = oppedge(e));
      ne = -link[firstmate];
      link[firstmate] = -mate[secondmate];
      link[secondmate] = -mate[firstmate];
      mate[firstmate] = eopp;
      mate[secondmate] = e;
    }
    return eopp;
  }

  // unlinks subblossoms in a blossom. oldbase is the base of the
  // blossom to be unlinked
  int unlink(int oldbase) {
    int i = nextvtx[oldbase], newbase = i;
    int e = link[nextbase(i)];
    for (int j = 0; j < 2; ++j) {
      do {
        lastedge[j] = oppedge(link[newbase]);
        for (int k = 0; k < 2; ++k) {
          link[newbase] = -link[newbase];
          do base[i] = newbase, i = nextvtx[i];
          while (i != nextbase(newbase));
          newbase = nextbase(newbase);
        }
      } while (link[nextbase(newbase)] == lastedge[j]);

      if (!j && link[nextbase(newbase)] != oppedge(e)) {
        lastedge[1] = lastedge[0];
        break;
      }
    }

    if (base[lastvtx[oldbase]] == oldbase)
         nextvtx[oldbase] = newbase;
    else nextvtx[lastvtx[oldbase] = oldbase] = dummyVtx;
    return e;
  }

  // updates numerical bounds for linking paths
  // called with lastd set to the bound on delta for the next \
search
  void set_bounds() {
    for (int v = 1; v <= V; nextd[v] = lastd, ++v) {
      if (link[v] < 0 || base[v] != v) continue;
      link[v] = -link[v];
      for (int i=v; i != dummyVtx; y[i] -= delta, i=nextvtx[i]);
      int e = mate[v];
```

```
      Weight del = slack(e);
      if (e != dummyEdge)
        for(int i=bend(e); i!=dummyVtx; y[i]-=del, i=nextvtx[i]);
    }
  }

  // undoes all blossoms to get the final matching
  void unpair_all() {
    for (int v = 1; v <= V; ++v) {
      if (base[v] != v || lastvtx[v] == v) continue;
      int nextu = v;
      nextbase(nextu) = dummyVtx;
      do {
        int u = nextu;
        nextu = nextvtx[u];
        int e = unlink(u);
        if (lastvtx[u] != u)
          nextbase(bend((lastedge[1] == oppedge(e)) ?
                 lastedge[0] : lastedge[1])) = u;
        int newbase = bmate(bmate(u));
        if (newbase != dummyVtx && newbase != u) {
          link[u] = -dummyEdge;
          rematch(newbase, mate[u]);
        }
        while (nextu == lastvtx[nextu] && nextu != dummyVtx)
          nextu = nextvtx[nextu];
      } while (lastvtx[nextu] != nextu);
    }
  }

  // utility functions
  int& nextbase(int v){ return nextvtx[lastvtx[v]]; }//NB!ret ref
  int bend(int e) { return base[end[e]]; }
  int bmate(int v) { return base[end[mate[v]]]; }
  int oppedge(int e) { return e ^ 1; }
  Weight slack(int e){ return y[end[e]]+y[end[e^1]]-weight[e|1]; \
}
};
```

Chapter 8: Geometry

8.1 Geometric primitives

Listing 8.1: point.cc

34 lines,

```
template <class T>
struct point {
  typedef T coord_type;
  typedef point S;
  typedef const S &R;
  T x, y;
  point(T _x=T(), T _y=T()) : x(_x), y(_y) { }
  bool operator< (R p) const {
    return x < p.x || x <= p.x && y < p.y;
  }
  S operator-(R p) const { return S(x - p.x, y - p.y); }
  S operator+(R p) const { return S(x + p.x, y + p.y); }
  S operator*(T d) const { return S(x * d, y * d); }
  S operator/(T d) const { return S(x / d, y / d); }
  T dot(R p) const { return x*p.x + y*p.y; }
  T cross(R p) const { return x*p.y - y*p.x; }
  T dist2() const { return dot(*this); }

  T dx(R p) const { return p.x - x; }
  T dy(R p) const { return p.y - y; }

  double dist() const { return sqrt(dist2()); }
  double angle() const { return atan2(y, x); }

  S unit() const { return *this / dist(); }
  S perp() const { return S(-y, x); }
  S normal() const { return perp().unit(); }

  double theta() {
    if (x==0 && y==0) return 0;
    double t = y / (x<0 ^ y<0 ? x-y : x+y);
    return x<0 ? y<0 ? t-2 : t+2 : t;
  }
};
```

Listing 8.2: point3.cc

21 lines,

```
template <class T>
struct point3 {
  typedef T coord_type;
  typedef point3 S;
  typedef const S &R;
  T x, y, z;
  point3(T _x=T(), T _y=T(), T _z=T()) : x(_x), y(_y), z(_z) { }
  bool operator< (R p) const {
    return x < p.x || x <= p.x && (y < p.y || y <= p.y && z<p.z);
  }
  S operator-(R p) const { return S(x - p.x, y - p.y, z - p.z); }
  S operator+(R p) const { return S(x + p.x, y + p.y, z + p.z); }
  S operator/(T d) const { return S(x / d, y / d, z / d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  S cross(R p) const { return S(y*p.z - z*p.y,
                              z*p.x - x*p.z,
                              x*p.y - y*p.x); }
};

// unit normal to a plane from two vectors
template <class P> P normal(P p, P q) { return unit(p.cross(q));}
```

Listing 8.3: point line relations.cc

30 lines, "point.cpp"

```
// Get a measure of the distance of a point from a line (0 on
// the line and positive/negative on the different sides).
template <class P> inline
double linedist(const P& p0, const P& p1, const P& q) {
  return (double) (p1-p0).cross(q-p0) / (p1-p0).dist();
}

// Get a measure of the distance of a point from a line segment.
template <class P> inline
double segdist(const P& p0, const P& p1, const P& q) {
  typename P::coord_type t=(p3-p1).dot(p2-p1), d=(p2-p1).dist2();
  if (t < 0) t = 0;
  if (t > d) t = d;
  return ((p1-p3)*d + (p2-p1)*t).dist() / d;// NB!! Overflow!!
}

// Determine if a point is on a line segment (incl end points).
template<class P> inline
bool on_segment(const P& p0, const P& p1, const P& q) {
  return (p0.dx(q)*p1.dx(q) <= 0 && p0.dy(q)*p1.dy(q) <= 0 &&
          (p1-p0).cross(q-p0) == 0);
}

// Determine on which side of a line a point is. +1/-1 is
// left/right of vector $p_1-p_0$ and 0 is on the line.
// [Note that this is sgn(linedist(...))]
template <class P> inline
int sideof(const P& p0, const P& p1, const P& q) {
  typename P::coord_type d = (p1-p0).cross(q-p0);
  return d > 0 ? 1 : d < 0 ? -1 : 0;
}

// Reflect point q around line passing through origin and p
template <class P>
P reflection(const P& p, const P &q) {
  double a = p.x*p.x - p.y*p.y, b = 2*p.x*p.y, det = p.dist2();
  return P(a*q.x + b*q.y, b*q.x - a*q.y) / det;
}
```

Listing 8.4: linear transformation.cc

7 lines,

```
// Apply the linear transformation which takes line p0-p1 to line
// q0-q1 to point r (without reflection).
template <class P>
P lintrans(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));
  typename P::coord_type den = dp.dist2();
  return q0 + P((r-p0).cross(num), (r-p0).dot(num)) / den;
}
```

8.1.1 Line intersection

Intersection point between two lines. Different cases depending on whether the lines are infinite or segments.

Listing 8.5: line isect.cc

34 lines,

```
const double NO_ISECT = -1.0/0.0;

template <class P> inline
double line_isect(const P& A0, const P& A1,
                  const P& B0, const P& B1) {
  typedef typename P::coord_type T;
  P dP1 = A1-A0, dP2 = B1-B0, dL = B0-A0;
  T det = dP1.cross(dP2), s = dL.cross(dP1), t = dL.cross(dP2);
  /* intersection between infinitely extending lines: */
  if (det == 0) return NO_ISECT;

  /* intersection between finite line segments: */
  if (det == 0) {
    T s1 = dP1.dot(dL), s2 = dP1.dot(B1)-dP1.dot(A0);
    if (t != 0 || min(s1, s2) > dP1.dist2() || max(s1, s2) < 0)
      return NO_ISECT;
    return sqrt((double) max(0, min(s1, s2)));
  }
  if (det < 0) det = -det, t = -t, s = -s;
  if (!(t >= 0 && t <= det && s >= 0 && s <= det))
    return NO_ISECT;

  /* both: */
  return (double)t / det;
}

template <class P> inline
bool line_isect(const P& A0, const P& A1,
                const P& B0, const P& B1, P &R) {
  double t = line_isect(A0, A1, B0, B1);
  if (t != NO_ISECT) R = A0*(1-t) + A1*t;
  return t != NO_ISECT;
}
```

8.1.2 Interval union

The union of several intervals given as pair¡first,last¿ in a container. The result is a disjoint list of intervals in ascending order.

Listing 8.6: ival union.cc

12 lines, <algorithm>

```
template <class It, class OIt>
It ival_union( It begin, It end, OIt dest ) {
  sort( begin, end );
  while( begin != end ) {
    *dest = *begin++;
    while( begin != end && begin->first <= dest->second )
      dest->second >?= begin++->second;
    ++dest;
  }
  return dest;
}
```

8.1.3 Circle tangents

The tangent points from a point to a circle. The algorithm returns if the point lies on the circles perimeter (in which case the two tangent points are equal).

Listing 8.7: circle tangents.cc

10 lines,

```
template <class P, class T>
bool circle_tangents(const P &p, const P &c, T r, P &t1, P &t2) {
  P a = (c-p), ap = perp(a);
  double a2 = dist2(a), r2 = r*r;
  P x = p+a*(1-r2/a2), y = ap*(sqrt(a2-r2)*r/a2);

  t1 = x + y;
  t2 = x - y;
  return fabs(a2-r2) < 1e-10; // or other eps.
}
```

8.2 Triangles

8.2.1 Heron triangle area

$A = \sqrt{p(p-a)(p-b)(p-c)}$, $p = \frac{a+b+c}{2}$.

8.2.2 Enclosing circle

`incircle` returns a determinant, whose sign determines whether a point lies inside the circle enclosing three other points.

Usage `bool enclosing_centre(a, b, c, &p[, eps]);`

Fills in the enclosing circle centre of points a, b, c in point p. Returns false if the points are colinear within the eps limit.

Usage `bool enclosing_radius(a, b, c, &r[, eps]);`

Fills in the enclosing circle radius in r, using $r = \frac{abc}{4K}$, where K is the triangle area (as in Heron). Returns false if the points are colinear within the eps limit.

Listing 8.8: incircle.cc

29 lines,

```
template <class P>
double incircle(P A, P B, P C, P D) {
  typedef typename P::coord_type T;
  P a = A - D; T a2 = a.dist2();
  P b = B - D; T b2 = b.dist2();
  P c = C - D; T c2 = c.dist2();
  return (a2*b.cross(c) + b2*c.cross(a) + c2*a.cross(b));
}

template <class P, class R>
bool enclosing_centre(P A, P B, P C, R &p, double eps = 1e-13) {
  typedef typename R::coord_type T;
  P a = A - C, b = B - C;
  T det2 = a.cross(b) * 2;
  if (-eps < det2 && det2 < eps) return false;
  T a2 = a.dist2(), b2 = b.dist2();
  p.x = (b.y * a2 - a.y * b2) / det2 + C.x;
  p.y = (a.x * b2 - b.x * a2) / det2 + C.y;
  return true;
}

template <class P, class T>
bool enclosing_radius(P A, P B, P C, T &r, T eps = 1e-13) {
  T a = (B-C).dist(), b = (C-A).dist(), c = (A-B).dist();
  T K4 = heron(a, b, c) * 4;
  if (K4 < eps) return false;
  r = a * b * c / K4;
  return true;
}
```

8.3 Polygons

8.3.1 Inside polygon

Complexity $\mathcal{O}(n)$

Determine whether a point is inside a polygon.

Listing 8.9: inside.cc

13 lines, "point_line_relations.cc"

```
template <class It, class P>
bool poly_inside(It begin, It end, const P &p,
                 bool strict = true) {
  bool inside = false;
  for (It i = begin, j = end - 1; i != end; j = i++) {
    if (on_segment(*j, *i, p)) return !strict;
    if (min(j->x, i->x) < p.x && max(j->x, i->x) >= p.x &&
        abs(j->x - i->x)*( p.y - i->y) >
        abs( p.x - i->x)*(j->y - i->y))
      inside ^= 1;
  }
  return inside;
}
```

8.3.2 Winding number

Complexity $\mathcal{O}(n)$

Theta meta-angle winding number of a point with a polygon. The polygon should be given in ccw-order. Also function `inside_wn` which does the same as `inside` but uses the winding number. The value of `inside_wn` is +1/-1 for inside/outside and 0 for on the edge.

Listing 8.10: winding number.cc

26 lines, "point_line_relations.cpp"

```
template <class It, class P>
double winding_nr(It begin, It end, const P &t, bool &onEdge) {
  double wind = 0;
  onEdge = false;
  for (It i = begin, j = end−1; i != end; j = i++) {
    if(onsegment(*i, *j, t)) {
      onEdge = true;
      continue;
    }
    double t1 = (t−*i).theta(), t2 = (t−*j).theta();
    double dt = t1−t2;
    if (dt > 2) dt −= 4;
    if (dt < −2) dt += 4;
    wind += dt;
  }
  return wind;
}

template<class It, class P> // V is vector/array of point<T>s
int inside_wn(It begin, It end, P t) {
  bool edge;
  double wind = winding_nr(begin,end,t, edge);
  if(edge) return wind>4 ? 1:0;
  // Not on edge, i.e. wind is (approx) 4*nr of turns.
  return wind > 2 ? 1:−1;
}
```

8.3.3 Polygon area

Twice the signed polygon area.

Listing 8.11: poly area.cc

7 lines, "point.cc"

```
template <class T, class It>
T poly_area2(It begin, It end) {
  T a = T();
  for (It i = begin, j = end − 1; i != end; j = i++)
    a += j−>cross(*i);
  return a;
}
```

8.3.4 Polyhedron volume

Signed polyhedron volume.

Listing 8.12: poly volume.cc

7 lines,

```
template <class V, class L>
double poly_volume(const V &p, const L &trilist) {
  typename L::value_type::coord_type v = 0;
  for (typename L::const_iterator i = trilist.begin(); i != \
trilist.end(); ++i)
    v += dot(cross(p[i−>a], p[i−>b]), p[i−>c]);
  return (double) v / 6;
}
```

8.3.5 Polygon cut

Usage `iterator r_end = poly_cut(v.begin(), v.end(), p0, p1, r.begin())`

Cuts a polygon with (a half plane specified by) a line. `r` is filled in with the cut polygon, and the end of the filled in interval is returned. The polygon is kept connected by (overlapping) line segments along the cutting line if the cut splits the polygon in parts.

Listing 8.13: poly cut.cc

17 lines, "line_isect.cpp"

```
template <class CI, class OI, class P>
OI poly_cut(CI first, CI last, P p0, P p1, OI result) {
  if (first == last) return result;
  P p = p1−p0;
  CI j = last; −−j;
  bool pside = p.cross(*j−p0) > 0;
  for (CI i = first; i != last; ++i) {
    bool side = p.cross(*i−p0) > 0;
    if (pside ^ side)
      line_isect(p0, p1, *i, *j, *result++);
    if (side)
      *result++ = *i;
    j = i; pside = side;
  }
  return result;
}
```

8.3.6 Center of mass

Polygon and triangular center of mass.

Listing 8.14: center of mass.cc

41 lines, <iterator>, "geometry.h"

```
template <class V>
inline double tri_area(V p) { // cross-product / 2
  return ((double)dx(p[0],p[1])*dy(p[0],p[2])−
        (double)dy(p[0],p[1])*dx(p[0],p[2]))/2;
}

template <class V>
void centerofmass( V p, int n, point<double> &com ) {
  com.x = com.y = 0.0;

  if( n<=3 ) {
    // Simple case
    for( int i=0; i<n; i++ ) {
      com.x += p[i].x;
      com.y += p[i].y;
    }
    com.x /= n;
    com.y /= n;
  } else {
    // More difficult case (NB! poly must be in ccw order!)
    typedef typename iterator_traits<V>::value_type::coord_type \
T;
    point<T> tri[3];

    tri[0] = p[0];

    double totarea=0.0, area;
    point<double> tri_com;
    for( int i=2; i<n; i++ ) {
      tri[1] = p[i−1];
      tri[2] = p[i];
      area = tri_area( tri ); // (with orientation)

      centerofmass( tri, 3, tri_com );
      com.x += area*tri_com.x;
      com.y += area*tri_com.y;
      totarea += area<0 ? −area:area;
    }
    com.x /= totarea;
    com.y /= totarea;
  }
}
```

8.4 Convex Hull

8.4.1 Graham scan

Usage `it hull_end = convex_hull(p.begin(), p.end())`

Swaps the points in `p` so the hull points are in order at the beginning. **NB!** Does not handle coinciding points.

Listing 8.15: convex hull.cc

38 lines,

```
template <class P>
struct cross_dist_comparator {
  P o; cross_dist_comparator(P _o) : o(_o) { }
  bool operator ()(const P &p, const P &q) const {
    typename P::coord_type c = (p−o).cross(q−o);
    return c != 0 ? c > 0 : (p−o).dist2() > (q−o).dist2();
  }
};

template <class It>
It convex_hull(It begin, It end) {
  typedef typename iterator_traits<It>::value_type P;
  // zero, one or two points always form a hull
  if (end − begin < 3) return end;
  // find a guaranteed hull point, sort in scan order around it
  swap(*begin, *min_element(begin, end));
  cross_dist_comparator<P> comp(*begin);
  sort(begin + 1, end, comp);
  // colinear points on first line of the hull must be reversed
  It i = begin + 1;
  for (It j = i++; i != end; j = i++)
    if ((*i−*begin).cross(*j−*begin) != 0)
      break;
  reverse(begin + 1, i);
  // place hull points first by doing a Graham scan
  It r = begin + 1;
  for (It i = begin + 2; i != end; ++i) {
    // change < 0 to <= 0 if colinear points are not desired
    while (r > begin && (*r - *(r-1)).cross(*i-*(r-1)) < 0)
      --r;
    swap(*++r, *i);
  }
  // removing colinear points at the end of the hull
  while (r-1 > begin && (*begin - *(r-1)).cross(*r - *(r-1))==0)
    --r;
  // return the iterator past the last hull point
  return ++r;
}
```

8.4.2 Three dimensional hull

Complexity $\mathcal{O}\left(n^2\right)$

Usage `convex_hull_space(points p, int n, list<ABC> &trilist)`

`trilist` is a list of ABC-tripples of indices of vertices in the 3D point vector `p`.

Note! Requires the hull to have positive volume. Arbitrarily triangulates the surface of the hull.

Listing 8.16: convex hull space.cc

48 lines, <set>

```
struct ABC {
  int a, b, c; ABC(int _a, int _b, int _c):a(_a), b(_b), c(_c) {}
  bool operator<(const ABC &o) const {
    return a!=o.a ? a<o.a : b!=o.b ? b<o.b : c<o.c;
  }
};

template <class V, class L>
bool convex_hull_space(V p, int n, L &trilist) {
  typedef typename V::value_type P3;
  typedef typename P3::coord_type T;
  typedef typename L::value_type I3;
  typedef set<pair<int, int> > SPI;
  int a, b, c; // Find a proper tetrahedron
  for (a = 1; a < n; ++a) if ((p[a]-p[0]).dist2() != T()) break;
  for (b = a + 1; b < n; ++b) if ((p[a]-p[0]).cross(p[b]-p[0]).
                                  dist2()) break;
  for (c = b + 1; c < n; ++c) if ((p[a]-p[0]).cross(p[b]-p[0]).
                                  dot(p[c]-p[0]) != T()) break;
  if (c >= n) return false;
  if ((p[a]-p[0]).cross(p[b]-p[0]).dot(p[c]-p[0]) > T())
    swap(a, b);
  trilist.push_back(I3(0, a, b)); // Use it as initial hull
  trilist.push_back(I3(0, b, c));
  trilist.push_back(I3(0, c, a));
  trilist.push_back(I3(a, c, b));
  for (int i = 1; i < n; ++i) {
    typename L::iterator it = trilist.begin();
    SPI edges;
    P3 &P = p[i];
    while (it != trilist.end()) {
      int a = it->a, b = it->b, c = it->c;
      P3 &A = p[a], &B = p[b], &C = p[c];
      P3 normal = (B-A).cross(C-A);
      T d = normal.dot(P-A);
      if (d > T()) {
        edges.insert(make_pair(a, b));
        edges.insert(make_pair(b, c));
        edges.insert(make_pair(c, a));
        trilist.erase(it++); // ugly!!
      } else ++it;
    }
    for (SPI::iterator j = edges.begin(); j != edges.end(); ++j)
      if (edges.count(make_pair(j->second, j->first)) == 0)
        trilist.push_back(I3(i, j->first, j->second));
  }
  return true;
}
```

8.4.3 Point inside hull

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

Usage `inside_hull(hull p, int n, point t)`

Determine whether a point `t` lies inside the hull given by the point vector `p`. The hull should not contain colinear points. A hull with 2 points are ok. The result is given as: 1 inside, 0 onedge, -1 outside.

Listing 8.17: inside hull.cc

24 lines, "point_line_relations.cpp"

```
template <class It, class P>
int inside_hull_sub(It begin, It end, It i1, It i2, const P&t) {
  if (i2 - i1 <= 2) {
    int s0 = sideof(*begin, *i1, t);
    int s1 = sideof(*i1, *i2, t);
    int s2 = sideof(*i2, *begin, t);
    if (s0 < 0 || s1 < 0 || s2 < 0) return -1;
    if (i1 == begin+1 && s0 == 0 || s1 == 0 ||
        i2 == end - 1 && s2 == 0)
      return 0;
    return 1;
  }
  It i = i1 + distance(i1, i2)/2;
  int side = sideof(*begin, *i, t);
  if (side > 0) return inside_hull_sub(begin, end, i, i2, t);
  else return inside_hull_sub(begin, end, i1, i, t);
}

template <class It, class P>
int inside_hull(It begin, It end, const P &t) {
  if (end - begin < 3)
    return onsegment(*begin, end[-1], t) ? 0 : -1;
  else return inside_hull_sub(begin, end, begin+1, end-1, t);
}
```

8.4.4 Hull diameter

Complexity $\mathcal{O}(n)$

Usage `hull_diameter2(hull p, int n, &i1, &i2)`

Determine the points that are farthest apart in a hull. `i1, i2` will be the indices to those points after the call. The squared distance is returned.

Listing 8.18: hull diameter.cc

20 lines, "point.cpp"

```
template <class It>
double hull_diameter2(It begin, It end, It &i1, It &i2) {
  typedef iterator_traits<It>::value_type::coord_type T;
  int n = end - begin;
  if (n < 2) { i1 = i2 = 0; return 0; }
  T m = 0;
  int i, j = 1, k = 0;
  It i, j = begin+1, k = begin;
  for (i = begin; i <= k; i++) { // wander around
    T d2 = (*j-*i).dist2(); // find opposite
    while (++j != end) {
      T t = (*j-*i).dist2();
      if (t > d2) d2 = t; else break;
    }
    --j;
    if (i == begin) k = j; // remember first opposite index
    if (d2 > m) m = d2, i1 = i, i2 = j;
  }
  return m;
}
```

8.4.5 Minimum enclosing circle

Complexity $\mathcal{O}(n)$

Fills in c with the centre point of the minimum circle, enclosing the n point vector p. The first version fills in indices to the points determining the circle, and returns whether the third index is used. The second version returns the enclosing circle radius as a double. Colinearity of a third point is determined by the eps limit.

Listing 8.19: mec.cc

23 lines, "hull_diameter.cpp", "incircle.cpp"

```
template <class It, class P>
bool mec(It begin, It end, P &c, It &i1, It &i2, It &i3,
         double eps = 1e-13) {
  typedef typename P::coord_type T;
  hull_diameter2(begin, end, i1, i2);
  c = (*i1 + *i2) / 2;
  T r2 = (c-*i1).dist2();
  bool f = false;
  for (int i = 0; i < n; ++i)
  if ((c-*i).dist2() > r2) {
    i3 = i, f = true;
    enclosing_centre(*i1, *i2, *i3, c, eps);
    r2 = (c-*i).dist2();
  }
 return f;
}

template <class It, class P>
double mec(It begin, It end, P &c, double eps = 1e-13) {
 It i1, i2, i3;
 mec(begin, end, c, i1, i2, i3, eps);
 return dist(c, *i1);
}
```

8.4.6 Line-hull intersect

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

Usage `line_hull_intersect(hull p, int n, point p1, point p2, &s1, &s2)`

Determine the intersection points of a hull with a line. `p1, p2, s1, s2` will be the intersection points and indices to the hull line segments that intersect after the call. Returns whether there is an intersection.

Listing 8.20: line hull intersect.cc

45 lines, "point.cpp", "point_line_relations.cpp (for linedist)"

```
template <class V, class T>
struct line_hull_isct {
 typedef point<T> P;
 const V &p;
 int n;
 const P &p1, &p2;
 int &s1, &s2;
 line_hull_isct(const V &_p, int _n, const P &_p1, const P &_p2,
               int &_s1, int &_s2)
   : p(_p), n(_n), p1(_p1), p2(_p2), s1(_s1), s2(_s2) { }

 // assumes 0 <= md <= i1d, i2d
 bool isct(int i1, int m, int i2, double md) {
  if (md <= 0) {
    s1 = findisct(i1, m) % n;
    s2 = findisct(i2, m) % n;
    return true;
  }
  if( i2-i1 <= 2 ) return false;
  int l = (i1 + m) / 2, r = (m + i2) / 2;
  double ld = linedist(p1, p2, p[l % n]);
  double rd = linedist(p1, p2, p[r % n]);
  if (ld <= md && ld <= rd) return isct(i1, l, m, ld);
  if (rd <= md && rd <= ld) return isct(m, r, i2, rd);
  else return isct(l, m, r, md);
 }
 int findisct(int pos, int neg) {
  int m = (pos + neg) / 2;
  if (m == pos) return pos;
  if (m == neg) return neg;
  double d = linedist(p1, p2, p[m % n]);
  if (d <= 0) return findisct(pos, m);
  else return findisct(m, neg);
 }
};

template <class V, class T>
bool line_hull_intersect(const V &p, int n,
                         const point<T> &p1, const point<T> &p2,
                         int &s1, int &s2) {
 double d = linedist(p1, p2, p[0]);
 return line_hull_isct<V, T>(p, n, d >= 0 ? p1 : p2,
                             d >= 0 ? p2 : p1, s1, s2).
   isct(0, n, 2*n, fabs(d));
}
```

8.5 Voronoi diagrams

8.5.1 Simple Delaunay triangulation

Complexity $\mathcal{O}(n^4)$

Usage `delaunay(points p, int n, trifun)`

Uses a `trifun(int, int, int)` to return all possible delaunay triangles as tripple indices to the point vector.

Note! Triangles may overlap if points are cocircular.

Listing 8.21: delaunay simple.cc

26 lines, "point.cpp"

```
template <class V, class F>
void delaunay(V p, int n, F trifun) {
 typedef typename V::value_type P;
 typedef typename P::coord_type T;
 for (int i = 0; i < n; ++i) {
  for (int j = i + 1; j < n; ++j) {
   P J = p[j] - p[i]; T jd = J.dist2();
   for (int k = i + 1; (j != k || ++k) && k < n; ++k) {
    P K = p[k] - p[i]; T kd = K.dist2();
    T qd = J.cross(K);
    if (qd > T()) {
     P q = P(J.y*kd - K.y*jd, jd*K.x - kd*J.x);
     bool flag = true;
     for (int l = 0; l < n; ++l) {
      P L = p[l] - p[i]; T dl = L.dist2();
      if (L.dot(q) + dl * qd < T()) {
       flag = false;
       break;
      }
     }
     if (flag) trifun(i, j, k);
    }
   }
  }
 }
}
```

8.5.2 Convex hull Delaunay triangulation

Complexity $\mathcal{O}$(3d convex hull)

Usage `delaunay(points p, int n, trifun)`

Returns an arbitrary triangulation if points are cocircular.

Note! Depending on convex hull implementation it may fail if *all* points are cocircular, as is currently the case.

Listing 8.22: delaunay hull.cc

15 lines, <vector>, <list>, "point3.cpp", "convex_hull_space.cpp"

```
template <class V, class F>
void delaunay(V &p, int n, F trifun) {
 typedef point3<typename V::value_type::coord_type> P3;
 typedef vector<P3> V3;
 typedef list<ABC> L;
 V3 p3(n);
 for (int i = 0; i < n; ++i)
   p3[i] = P3(p[i].x, p[i].y, p[i].dist2());
 L l;
 convex_hull_space(p3, n, l);
 for (L::iterator it = l.begin(); it != l.end(); ++it)
  if ((p3[it->b]-p3[it->a]).
      cross(p3[it->c]-p3[it->a]).dot(P3(0,0,1)) < 0)
    trifun(it->a, it->c, it->b); // triangles are turned!
}
```

8.6 Misc. Point Set Problems

8.6.1 Closest Pair Divide and Conquer

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

Usage `closestpair(points p, int n, &i1, &i2)`

`i1, i2` are the indices to the closest pair of points in the point vector `p` after the call. The distance is returned.

Listing 8.23: closest pair.cc

75 lines, <iterator>, <vector>

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

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

template<class It, class IIt> /* IIt = vector<It>::iterator */
double cp_sub(IIt ya, IIt yaend, IIt xa, It &i1, It &i2) {
 typedef typename iterator_traits<It>::value_type P;
 int n = yaend-ya, split = n/2;

 if(n <= 3) { // base case
```

```
    double a = (*xa[1] - *xa[0]).dist();
    double b = 1e50, c = 1e50;
    if(n == 3)
      b = (*xa[2] - *xa[0]).dist(), c = (*xa[2] - *xa[1]).dist();
    if(a <= b) {
      i1 = xa[1];
      if(a <= c) return i2 = xa[0], a;
      else return i2 = xa[2], c;
    } else {
      i1 = xa[2];
      if(b <= c) return i2 = xa[0], b;
      else return i2 = xa[1], c;
    }
  }

  vector<It> ly, ry, stripy;
  P splitp = *xa[split];
  double splitx = splitp.x;

  for(IIt i = ya; i != yaend; ++i) { // Divide
    if(*i != xa[split] && (**i-splitp).dist2() < 1e-12)
      return i1 = *i, i2 = xa[split], 0;// nasty special case!
    if (**i < splitp) ly.push_back(*i);
    else ry.push_back(*i);
  } // assert((signed)lefty.size() == split)

  It j1, j2; // Conquer
  double a = cp_sub(ly.begin(), ly.end(), xa, i1, i2);
  double b = cp_sub(ry.begin(), ry.end(), xa+split, j1, j2);
  if(b < a) a = b, i1 = j1, i2 = j2;
  double a2 = a*a;

  for(IIt i = ya; i != yaend; ++i) { // Create strip (y-sorted)
    double x = (*i)->x;
    if(x >= splitx-a && x <= splitx+a)
      stripy.push_back(*i);
  }

  for(IIt i = stripy.begin(); i != stripy.end(); ++i) {
    const P &p1 = **i;
    for(IIt j = i+1; j != stripy.end(); ++j) {
      const P &p2 = **j;
      if(p2.y-p1.y > a) break;
      double d2 = (p2-p1).dist2();
      if(d2 < a2)
        i1 = *i, i2 = *j, a2 = d2;
    }
  }
  return sqrt(a2);
}

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

  sort(xa.begin(), xa.end(), it_less<It>);
  sort(ya.begin(), ya.end(), y_it_less<It>);

  return cp_sub(ya.begin(), ya.end(), xa.begin(), i1, i2);
}
```

8.6.2 Closest Pair Simpler method

Complexity $\mathcal{O}\left(n^2 \text{ (average } n)\right)$

Usage `closestpair(points p, int n, &i1, &i2)`

Listing 8.24: closest pair simple.cc

9 lines,

```
template <class It>
double closest_pair(It begin, It end) {
  sort(begin, end);
  double best = 1e99;
  for (It p = end; p-- != begin; )
    for (It l = p; l-- != begin && sqr(l->x - p->x) < best; )
      best <?= p->dist2(*l);
  return best;
}
```

8.6.3 Minimal enclosing d-ball

Expected complexity $\mathcal{O}(d!n)$

Listing 8.25: enclosing ball.cc

46 lines, <algorithm>

```
/* P is D-dimensional point type.
 * P needs: operator+, operator- (componentwise)
 *          operator* (by scalar), dot(), dist2()
 */
template <typename P, int D=3>
struct enclosing_ball {
  int support;
  P ortho[D+1], center[D+1];
  double radsqr[D+1], z[D+1];

  const P& Center() { return center[support]; }
  double RadSqr() { return radsqr[support]; }

  // find the minimal enclosing ball of range [begin, end)
  template <typename It>
  void compute(It begin, It end) {
    radsqr[0] = support = 0; // reset ball
    random_shuffle(begin, end); // to ensure expected linear time
    find_sub(begin, end, 0);
  }

  bool add_support(const P& p, int b) {
    if ((p-Center()).dist2() - RadSqr() < 1e-10)
      return false; // p already inside sphere
    if (!b) center[0] = p;
    else {
      P q = p - center[0], proj = P();
      for (int i = 1; i < b; ++i)
        proj = proj + ortho[i] * (ortho[i].dot(q) / z[i]);
      ortho[b] = q - proj;
      z[b] = ortho[b].dist2();
      if (z[b] < 1e-25 * RadSqr()) return false;
      double e = ((p-center[b-1]).dist2() - radsqr[b-1]) / 2;
      center[b] = center[b-1] + ortho[b]*(e/z[b]);
      radsqr[b] = radsqr[b-1] + e*e/z[b];
    }
    support = b;
    return true;
  }

  template <typename It>
  void find_sub(It begin, It end, int b) {
    if (begin != end && b != D+1) {
      find_sub(begin, --end, b);
      if (add_support(*end, b))
        find_sub(begin, end, b+1);
        // TODO: Welzl's move-to-front heuristic
    }
  }
};
```

Chapter 9: Games, Puzzles and Martinis

9.1 Games

9.1.1 Impartial take-and-break games (NIM-like games)

A game where two players take turns removing (indistinguisable) tokens from some heaps of tokens. The player removing the last token (thus causing the next player to be unable to move) is the winner. The moves available are: removing x tokens from a heap (for some set of allowed x), and splitting a heap of n tokens into two heaps of n_1 and n_2 tokens where $n_1, n_2 < n$. Because every move reduces a heap size by at least 1, such games can never end in draw. To find optimal strategies, Grundy numbers (or nimbers) can be used. The Grundy value of a state $S = \{n\}$ is defined as $G(S) = \text{mex } S'$ where S' runs over all successor states to S and mex is the minimal excluded (nonnegative) value. The Grundy value of $S = \{n_1, n_2, \dots n_k\}$ is defined as $\bigoplus_{i=1}^{k} G(\{n_i\})$. A state S is winning iff $G(S) \neq 0$.

9.2 Puzzles

Listing 9.1: nqueens.cc

7 lines,

```
void N_queens(int N, int *cols) {
  int n = N & ~1, a1 = 1, a2 = 0;
  if (n % 6 == 2) a1 = n/2-1, a2 = n/2+2;
  for (int i = 0; i < n/2; ++i)
    cols[i] = (a1 + 2*i) % n, cols[i+n/2] = (a2 + 2*i) % n;
  if (N & 1) cols[n] = n;
}
```

Listing 9.2: magicsquare.cc

47 lines, <algorithm>

```
int sol[500][500];

void magic_odd(int n, int y0, int x0, int add) {
  int x = n/2, y = 0;
  for (int i = 0; i < n*n; ++i, ++x, --y) {
    if (x == n) x = 0;
    if (y < 0) y = n-1;
    if (sol[y0+y][x0+x] != -1) {
      if (--x < 0) x = n-1;
      if ((y += 2) >= n) y -= n;
    }
    sol[y0+y][x0+x] = i+add;
  }
}

void magic_4mul(int n, int add) {
  for (int r = 0, base = 0; r < n/2; ++r, base += n) {
    for (int i = 0; i < n; ++i) {
      bool b = (((i+1)/2) & 1) ^ (r & 1);
      sol[r][i] = ((b ? n*n-i-base-1 : i+base)) + add;
      sol[n-r-1][n-i-1] = n*n-sol[r][i]-1 + 2*add;
    }
  }
}

bool magic_square(int n, int sum) {
  if (n == 2 || (sum - n*(n*n-1)/2) % n) return false;
  int add = (sum - n*(n*n-1)/2)/n;
  memset(sol, -1, sizeof(sol));
  if (n & 1)
    magic_odd(n, 0, 0, add);
  else if (!(n & 3))
    magic_4mul(n, add);
  else {
    magic_odd(n/2, 0, 0, add);
    magic_odd(n/2, n/2, 0, add + 3*n*n/4);
    magic_odd(n/2, 0, n/2, add + n*n/2);
    magic_odd(n/2, n/2, n/2, add + n*n/4);
    for (int i = 0; i < n/2; ++i)
      for (int j = 0; j < n/4; ++j)
        swap(sol[i][j+(i==n/4)], sol[i+n/2][j+(i==n/4)]);
    for (int i = n-n/4+1; i < n; ++i)
      for (int j = 0; j < n/2; ++j)
        swap(sol[j][i], sol[j+n/2][i]);
  }
  return true;
}
```

9.2.1 Tournament scheduling

Compute a minimal day schedule where all teams meet all other teams. `sched[i][j]` will contain the opponent of team `j` on day `i`, or 0 if no game that day. Returns number of days.

Listing 9.3: tournaments.cc

8 lines,

```
template <class M> int tournaments(int n, M& sched) {
  int d = n-1 + (n & 1);
  for (int i = 0; i < d; ++i)
    for (int j = 0; j < d; ++j)
      if ((sched[i][j] = (d+i-j) % d) == j)
        sched[i][j] = (n & 1) ? -1 : (sched[i][d] = j), d;
  return d;
}
```

9.2.2 Josephus

Which person remains when repeatedly removing the k:th person from a total of n persons (cyclic)?

Complexity $\mathcal{O}\left(\log_{\frac{k}{k-1}}(n)\right)$

Listing 9.4: josephus.cc

6 lines,

```
int josephus(int n, int k) {
  int d = 1;
  while (d <= (k - 1) * n)
    d = (k * d + k - 2) / (k - 1);
  return k * n + 1 - d;
}
```

9.3 Martinis

Listing 9.5: bitmanip.cc

51 lines,

```
typedef unsigned int uint;

int lowest_bit(int x) { return x & -x; }

bool ispow2(uint x) { return (x & x - 1) == 0; }

uint nlpow2(uint x) { // power of two round up
  if (!x--) return 1;
  for (int i = 0; i < 5; ++i)
    x |= x >> (1 << i);
  return ++x;
}

// next higher number with the same number of bits set
```

```
uint nexthi_same_count_ones(uint a) {
  uint c = (a & -a), r = a+c;
  return (((r ^ a) >> 2) / c) | r;
}

const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000};

uint log2(uint v) {
  uint c = 0;
  for (int i = 4; i >= 0; i--)
    if (v & b[i])
      v >>= (1 << i), c |= (1 << i);
  return c;
}

template <class T> // bit count, use with bitop
void bitcount(T &x, int s, T m) { x = (x >> s & m) + (x & m); }
// Use __builtin_popcount or __builtin_popcountll if available.

template <class T> // bit reversal, use with bitop
void revbits(int &x, int s, int m) { x = x >> s & m | (x & m) << \
s; }

template <class F> int bitop(int x, F _fun) {
  _fun(x, 1, 0x55555555);
  _fun(x, 2, 0x33333333);
  _fun(x, 4, 0x0f0f0f0f);
  _fun(x, 8, 0x00ff00ff);
  _fun(x,16, 0x0000ffff);
  return x;
}

template <class F> long long bitop(long long x, F _fun) {
  _fun(x, 1, 0x5555555555555555ll);
  _fun(x, 2, 0x3333333333333333ll);
  _fun(x, 4, 0x0f0f0f0f0f0f0f0fll);
  _fun(x, 8, 0x00ff00ff00ff00ffll);
  _fun(x,16, 0x0000ffff0000ffffll);
  _fun(x,32, 0x00000000ffffffffll);
  return x;
}
```

9.3.1 Dry Martini

- 5 parts dry gin
- 1 part Noilly Prat (dry vermouth)

Stir with ice.
Strain into a chilled martiniglas.
Decorate with a green olive.

9.3.2 Perfect Martini

- 4 parts gin
- 1 part sweet Martini (sweet vermouth)
- 1 part Noilly Prat (dry vermouth)

Stir and strain into a cocktail glass.
Decorate with a red cherry and a small lemon peel
(See also Perfect Numbers, page 8.)

9.3.3 Longest Increasing Subsequence

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

Best served with fresh strawberries.

Listing 9.6: lis.cc

18 lines,

```
template <class It> // change this function if necessary
inline bool it_less(It i, It j) { return *i < *j; }

// NB! output cannot overlap input!
template <class It, class OutIt>
OutIt lis(It begin, It end, OutIt res) {
  int n = end - begin, i = 0;
  It idxs[n], back[n], *idxend = idxs;
  for (It it = begin; it != end; ++it, ++i) {
    // upper_bound if non-decreasing rather than increasing.
    It *b = lower_bound(idxs, idxend, *it, it_less);
    if (b == idxend) ++idxend;
    *b = it;
    back[i] = (b == idxs) ? end : *--b;
  } // length is idxend - idxs
  It it = idxend[-1]; OIt ans = res += idxend - idxs;
  while (it != end) *--ans = *it, it = back[it-begin];
  return res;
}
```

9.4 Hex and tri grids

Listing 9.7: grids.cc

28 lines,

```
%!
/size 15 def
/r 20 def
size size scale
1 size div setlinewidth
-1 -2 moveto
/e 3 def
r 3 mul {
  e 3 sqrt rmoveto
  r {
    1 3 sqrt rlineto
    2 0 rlineto
    1 3 sqrt neg rlineto
    false { %Trigrid
      -4 0 rlineto
      1 3 sqrt neg rmoveto
      2 3 sqrt 2 mul rlineto
      -2 0 rmoveto
      2 3 sqrt -2 mul rlineto
      1 3 sqrt rmoveto
    } if
    2 0 rmoveto
  } repeat
  r -6 mul 0 rmoveto
  /e e neg def
} repeat
stroke
showpage
```

Index