Inhaltsverzeichnis

~/.emacs

```
(custom-set-variables
 '(case-fold-search t)
 '(column-number-mode t)
 '(line-number-mode t)
 '(mouse-wheel-follow-mouse t)
 '(mouse-wheel-mode t nil (mwheel))
)

(global-unset-key "\C-z")
(global-set-key "\C-z" 'yank)
(global-unset-key "\M-g")
(global-set-key "\M-g" 'goto-line)
```

template.cc

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cctype>
#include <utility>

#define DEBUG
#ifdef DEBUG
#define DBG(x) cout << x << endl
#define DEB(x) cerr << x << endl
#else
#define DBG(x)
#define DEB(x)
#endif

using namespace std;

typedef unsigned long long ull;
typedef long long sll;

int main() {
  return 0;
}
```

~/.bashrc

```
export CXXFLAGS="-Wall -W -g"
m() { p=`basename $PWD`;
gmake $p && echo compiled && time ./$p < $p.in; }
```

Primzahlensieb

```
#include <iostream>
#include <cmath>
#include <map>
#include <iterator>
#include <list>
#include <set>

using namespace std;

int* primes;
int prime_calc;
int num_primes;

void prime_sieve (int in = 100000) {
  prime_calc = in;
  num_primes = 1;

  bool* prime_mark = new bool[prime_calc];

  int s = (int)sqrt((double)prime_calc) + 1;

  for (int i=2; i < prime_calc; i+=2 ) {
    prime_mark[i] = false;
    prime_mark[i+1] = true;
  }

  prime_mark[0] = prime_mark[1] = false;
  prime_mark[2] = true;

  for( int i=3; i<s; i+=2 ) {
    if( prime_mark[i] ) {
      num_primes++;

      for( int j=3*i; j<prime_calc; j+=(2*i) )
        prime_mark[j] = false;
    }
  }

  for( int i = s; i <prime_calc; i++ )
    if ( prime_mark[i] ) num_primes++;
  primes = new int[num_primes];
  num_primes = 0;
  for( int i=2; i<prime_calc; i++ )
    if (prime_mark[i]) primes[num_primes++] = i;
  delete[] prime_mark;
}

// Euler's totient function, phi(x):
// the number of integers smaller than and relative prime to x
void init_phi(int* phi, int n) {
  for (int i=1; i<n; ++i) phi[i]=i;
  for (int i=2; i<n; ++i)
    if (phi[i]==i) // i is prime
      for (int j=i; j<n; j+=i) phi[j]=phi[j]*(i-1)/i;
}

// simple factorization (only factors till prime_calc
template<typename T, typename BI>
void factorize (T in, BI& factors) {
  T prod = 1;
  T end = (T)sqrt ((double)in) + 1;

  for (int i = 0; primes[i] < end && prod < in; ++i) {
    if (in % primes[i] == 0) {
      T f (primes[i]);
      prod *= f;
      T curr = f * f;
      int times = 1;
      while (in % curr == 0) {
        times++;
        curr *= f;
        prod *= f;
      }
      factors.push_back (make_pair (f,times));
    }
  }
  // if this is wrong your toasted
  if (prod != in)
    factors.push_back (make_pair (in/prod, 1));

}
```

```
// begin-end of factorization pairs
template<typename IT, typename C, typename T>
  void get_divisors (IT begin, IT end, T curr, C& divisors) {
  if (begin == end) {
    divisors.insert (curr);
    return;
  }

  T prod (1);
  IT next (begin);
  ++next;
  for (T i (0); i <= begin->second; ++i) {
    get_divisors (next, end, curr * prod, divisors);
    prod *= begin->first;
  }
}

int main() {
  prime_sieve(2000000000);
  cout << num_primes << endl;

  unsigned n;
  list<pair<unsigned, unsigned> > factors;
  while (cin >> n) {
    factors.clear();
    factorize (n, factors);
    unsigned c = 1;
    for (list<pair<unsigned, unsigned> >::iterator
           i = factors.begin();
         i != factors.end(); ++i) {
      c *= 2 * i->second + 1;
    }
    cout << n << ' ' << c << endl;
  }
}
```

Binary Heap

```
// Priority queue, implementiert als binary heap
// Elements are stored by reference (internally as pointers)
template<typename T, typename Compare = less<T> >
class prio_queue {
 private:
  T** heap;
  size_t heapSize;
  Compare cmp;
  void adjust(size_t hole, T* value);

 public:
  prio_queue(size_t capacity, Compare comp=Compare())
    : heap(new T*[capacity]), heapSize(0), cmp(comp) { }
  ~prio_queue() { delete[] heap; }
  T& extract() { T* m=heap[0];
               adjust(0, heap[--heapSize]); return *m; }
  T& root() { return *heap[0]; }
  bool insert(T& v);
  bool empty() const { return !heapSize; }
  size_t size() const { return heapSize; }
  void clear() { heapSize=0; }
};

// insert if not already contained, else decrease key
template <typename T, typename Compare>
inline bool
prio_queue<T, Compare>::insert(T& v) {
  size_t vhi=v.heapIndex;
  if (vhi>=0 && vhi<heapSize && heap[vhi]==&v) {
    adjust(vhi, &v);
    return false;
  }
  else {
    adjust(heapSize++, &v);
    return true;
} }
// Remove heap[hole] and insert value instead,
// maintain heapIndex of elements
template <typename T, typename Compare>
void
prio_queue<T, Compare>::adjust(size_t hole, T* value) {
  size_t other;

  other=(hole-1)/2; /* other=parent */
  if (hole>0 && cmp(*value, *heap[other])) {
    // new element is better than parent
    // => move hole rootward before insertion
    do {
      (heap[hole]=heap[other])->heapIndex=hole;
      hole=other;
      other=(hole-1)/2;
    } while (hole>0 && cmp(*value, *heap[other]));
  }
  else {
    /* move hole leafward before insertion */
    other=2*hole+2; /* other=secondChild */
    while (other < heapSize) {
      if (cmp(*heap[other-1], *heap[other])) --other;
      /* other now is the better of the two children */
      if (!cmp(*heap[other], *value)) break;
      (heap[hole]=heap[other])->heapIndex=hole;
      hole=other;
      other=2*hole+2;
    }
    if (other-- == heapSize && cmp(*heap[other], *value)) {
      (heap[hole]=heap[other])->heapIndex=hole;
      hole=other;
    }
  }
  (heap[hole]=value)->heapIndex=hole;
}
```

Longest Increasing Subsequence

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// calculate the length of the longest increasing subsequence
// complexity: O(nlogn)
template<class T>
int lis (const vector<T> &v)
{
    vector<T> last;
    for (typename vector<T>::const_iterator i = v.begin ();
         i != v.end (); i++)
    {
        typename vector<T>::iterator pos =
            lower_bound (last.begin (), last.end (), *i);
        if (pos == last.end ())
            last.push_back (*i);
        else *pos = *i;
    }

    return last.size ();
}
```

Longest Common Subsequence

```
// length of longest common subsequence (lcs)
// tested with 10066

template<class T>
int lcs (const vector<T> &v1, const vector<T> &v2) {

  unsigned n1 = v1.size (), n2 = v2.size ();
  vector<int> b1 (n1+1), b2 (n1+1);

  for (unsigned i = 0; i < n2; ++i) {
    b2[0] = 0;
    for (unsigned j = 1; j <= n1; ++j) {
      b2[j] = b2[j-1];
      b2[j] >?= b1[j];
      if (v1[j-1] == v2[i])
        b2[j] >?= 1 + b1[j-1];
    }

    b1 = b2;
  }

  return b1.back ();
}
```

Euclid

```
template<typename T> inline T gcd(T a, T b) { // a, b unsigned
  T d;
  if (a<b) { d = a; a = b; b = d; }
  if (b==0) return a;
  do { d=a%b; a=b; b=d; } while(d!=0);
  return a;
}

template<typename T> inline T lcm(T a, T b) { // a, b unsigned
  return a*b/gcd(a,b);
}
```

```
template<typename IT> typename iterator_traits<IT>::value_type
lcmi (IT begin, IT end) {
  typedef typename iterator_traits<IT>::value_type value_t;
  value_t a=*begin;
  while (++begin != end) {
    value_t& b=*begin;
    a=a*b/gcd(a,b);
  }
  return a;
}

// post: ret = ax + by, ret>=0
template<typename T> T extended_euclid (T a, T b, T& x, T& y) {
  x = 0; y = 0;

  if (a<b) return extended_euclid(b, a, y, x);
  if (b == 0) {
    x = a>0 ? 1 : -1;
    y = 0;
    return a>0 ? a : -a;
  }
  T x1 (0), x2 (1), y1 (1), y2 (0);
  while (b != 0) {
    T q = a / b;
    T r = a - q * b;
    x = x2 - q * x1;
    y = y2 - q * y1;
    a = b; b = r;
    x2 = x1; x1 = x;
    y2 = y1; y1 = y;
  }
  if (a>0) {
    x = x2; y = y2;
    return a;
  }
  else {
    x = -x2; y = -y2;
    return -a;
  }
}
```

Stringmatching – KMP

```
/* Beim Verfahren nach Knuth-Morris-Pratt nutzt man die
Tatsache aus, dass wenn man beim Vergleich des Musters mit
dem Text an der j-ten Stelle des Musters ein Mismatch
erhält, die vorangehenden j-1 Zeichen im Muster und Text
bereits übereingestimmt haben. Diese Eigenschaft nutzt man,
um das Muster nach dem Mismatch nicht stets nur um eine
Position, wie beim naiven Verfahren, sondern so weit wie
möglich nach rechts zu verschieben. Dieses Verfahren führt
auf eine Laufzeit von O(n+m). */

#include <iostream>
#include <string>

using namespace std;

void KMP(const char *pattern, int m, const char *text,
        int n, int pos, int *next) {
    int i,j;
    for (i=0,j=0;j<m && i<n;i++,j++) {
        while((j>=0) && (text[i]!=pattern[j])) j = next[j];
        if (j==m-1) {
            // found pattern at position i-m+1+pos in text
            cout << i-m+1+pos << endl;
            j=next[j]-1;
            i--;
        }
    }
}

void initnext(const char *p,int m,int *next) {
    next[0]=-1;
    for (int i=0,j=-1;i<m;i++,j++,next[i]=j) {
        while((j>=0) && (p[i]!=p[j])) j = next[j];
    }
}

int main() {
    string pattern = "fisch";
    string text = "fischers fritz fischt frische fische";
    int m = pattern.length();
    int n = text.length();
    int pos = 0;
    int *next = new int[m+1];
    initnext(pattern.c_str(), m, next); // Preprocessing

    KMP(pattern.c_str(), m, text.c_str(), n, pos, next);
    return 0;
}
```

Stringmatching – BM

```
/*Das Verfahren von Boyer-Moore verfolgt die Grundidee, dass
man ein Muster von links nach rechts an den Text anlegt,
aber zeichenweise von rechts nach links vergleicht. Dabei
verwendet das Verfahren zwei Heuristiken, die
Vorkommens-Heuristik und die Match-Heuristik. Man kann
erwarten, dass das Verfahren für genügend kurze Muster und
hinreichend große Alphabete etwa O(n/m) Schritte durchführt,
d.h. das Verfahren inspiziert nur jedes m-te Tetxtzeichen
und das Muster kann nahezu immer um die gesamte Musterlänge
nach rechts verschoben werden."*/

#include <iostream>
#include <string>

using namespace std;

void processBCshift(const char *pattern, int m, int *bc) {
    for (int i=0; i<256; i++) bc[i] = m;
    for (int i = 0; i<m-1; i++) bc[pattern[i]] = m-i-1;
}

void suffixes(const char *pattern, int m, int *suff) {
    int f = 0;
    suff[m-1] = m;
    int g = m-1;
    for (int i = m-2; i>=0; i--) {
        if (i > g && suff[i+m-1-f] < i-g)
            suff[i] = suff[i+m-1-f];
        else {
            if (i < g) g = i;
            f = i;
            while (g >= 0 && pattern[g] == pattern[g+m-1-f])
                g--;
            suff[i] = f-g;
        }
    }
}

void processGSshift(const char *pattern, int m, int *gs) {
    int suff[m+1];
    suffixes(pattern, m, suff);
    for (int i=0; i<m; i++)
        gs[i] = m;
    int j = 0;
    for (int i = m-1; i>=-1; i--)
        if (i == -1 || suff[i] == i+1)
            for (; j<m-1-i; j++)
                if (gs[j] == m)
                    gs[j] = m-1-i;
    for (int i=0; i<=m-2; i++)
        gs[m-1-suff[i]] = m-1-i;
}


void BM(const char *pattern, int m, const char *text,
        int n, int *gs, int *bc) {
    int i, j;

    // Searching
    j = 0;
    while (j <= n-m) {
        for (i=m-1; i>=0 && pattern[i]==text[i+j]; i--);
        if (i < 0) {
            // found pattern at position j in text
            cout << j << endl;
            j += gs[0];
        }
        else
            j += gs[i] >? bc[text[i+j]]-m+1+i;
    }
}


int main() {
    string pattern = "fisch";
    string text = "fischers fritz fischt frische fische";
    int m = pattern.length();
    int n = text.length();

    int gs[m+1], bc[256];  // 256 is alphabet size

    // Preprocessing
    processGSshift(pattern.c_str(), m, gs);
    processBCshift(pattern.c_str(), m, bc);

    // Boyer Moore pattern matching
    BM(pattern.c_str(), m, text.c_str(), n, gs, bc);
    return 0;
}
```

Konvexe Hülle

```
/* Convex Hull nach Graham, wie in Robert Sedgewick: Algorithmen
   beschrieben. Liefert keine kolinearen Punkte, nur die Ecken. */

#include <stdio.h>
#include <math.h>
#include <values.h>
#include <algorithm>

#define PSEUDOANGLE // schnellere Pseudowinkelbestimmung

// Datenstruktur
struct point {
    int x, y;
    double theta;
    bool operator< (const point& b) const {
        if (theta==b.theta) return x < b.x;
        else return theta < b.theta;
    }
};

// Laufrichtungsbestimmung 3er Punkte
inline int
ccw (const point &p0, const point &p1, const point &p2) {
    int dx1 = p1.x - p0.x; int dy1 = p1.y - p0.y;
    int dx2 = p2.x - p0.x; int dy2 = p2.y - p0.y;
    if     (dx1*dy2>dy1*dx2)                return 1;
    else if (dx1*dy2<dy1*dx2)               return -1;
    else return 0; // kollinear, egal welche Reihenfolge
};

#ifdef PSEUDOANGLE
// Effiziente Pseudowinkelerstellung zw. 2 Punkten zum sortieren
inline void gettheta (const point &p1, point &p2) {
    int dx = p2.x - p1.x;
    int ax = (dx<0) ? -dx : dx;
    int dy = p2.y - p1.y;
    int ay = (dy<0) ? -dy : dy;

    if (dx==0 && dy==0) p2.theta = 0.0;
    else            p2.theta = (double)dy / (double)(ax+ay);
    if (dx<0)       p2.theta = 2 - p2.theta;
    else if (dy<0)  p2.theta = 4 + p2.theta;
};

#else
// etwas ineffizientere Variante, dafür weniger Tipparbeit
inline void gettheta (const point &p1, point &p2) {
    p2.theta = atan2(p2.y - p1.y, p2.x - p1.x);
};
#endif

int convexhull(point* p, int length, int anchor) {
    point temp;
    // Anker als p[1]
    if (anchor) { temp=p[anchor]; p[anchor]=p[1]; p[1]=temp; }
    // Punkte um Anker herum sortieren
    for (int i=2; i<=length; i++) gettheta(p[1], p[i]);
    sort(p+2, p+length+1);

    // Graham
    p[0] = p[length];
    int m = 2;
    for (int i=3; i<=length; i++) {
        while (ccw(p[m], p[m-1], p[i])>=0) m--;
        m++;
        temp = p[m]; p[m] = p[i]; p[i] = temp;
    }
    // Abschluss auf kolineare Punkte prüfen
    if (ccw(p[m], p[m-1], p[1])==0) m--;
    return m;
}

int main () {
    point p[100];
    int n;
    scanf("%d", &n); // Anzahl Punkte einlesen

    int miny = MAXINT;
    int minx = MAXINT;
    int anchor = 0;
    for (int j=1; j<=n; j++) {
        // Punkte einlesen (erster Punkt: index 1)
        scanf("%d %d", &p[j].x, &p[j].y);
        // Ankerpunkt (min y und min x) setzen
        if (p[j].y < miny) {
            miny=p[j].y;
            minx=p[j].x;
            anchor=j;
            }
        else if (p[j].y == miny && p[j].x < minx) {
            minx=p[j].x;
            anchor=j;
        }
    }

    // Konvexe Hülle erzeugen und testeshalber ausgeben
    int hullsize = convexhull(p, n, anchor);
    for (int j=1; j<=hullsize; j++)
      printf("%d %d\n", p[j].x, p[j].y);
}
```

Dancing Links

```
struct column;

struct node {
  node *u, *d, *l, *r;
  column *c;
} nodes[4096];
unsigned num_nodes;

struct column : public node {
  int s;
} root, cols[64];

node* add_node(column *c, node* prev = 0) {
  node* n = nodes + num_nodes++;
  if (prev == 0) n->r = prev = n;
  ++ c->s;
  n->c = c;
  n->d = c;
  n->u = c->u;
  n->l = prev;
  n->r = prev->r;
  n->l->r = n->r->l = n->u->d = n->d->u = n;
  return n;
}

void cover_column(column* c) {
  c->l->r = c->r;
  c->r->l = c->l;
  for (node* i = c->d; i != c; i = i->d) {
    for (node* j = i->r; j != i; j = j->r) {
      j->u->d = j->d;
      j->d->u = j->u;
      -- j->c->s;
    }
  }
}

void uncover_column(column* c) {
  for (node* i = c->u; i != c; i = i->u) {
    for (node* j = i->l; j != i; j = j->l) {
      ++ j->c->s;
      j->u->d = j->d->u = j;
    }
  }
  c->r->l = c->l->r = c;
}

inline column* choose_column() {
  column* best_col = 0;
  int best_size = 0x7fffffff;
  for (column* c = static_cast<column*>(root.r); c != &root;
       c = static_cast<column*>(c->r)) {
    if (c->s >= best_size) continue;
    best_size = c->s;
    best_col = c;
  }
  return best_col;
}

bool dance() {
  if (root.r == &root) return true; // complete cover
  column* c = choose_column();
  if (c->d == c) return false; // uncoverable column
  cover_column(c);
  for (node* r = c->d; r != c; r = r->d) {
    for (node* i = r->r; i != r; i = i->r)
      cover_column(i->c);
    if (dance()) return true;
    for (node* i = r->l; i != r; i = i->l)
      uncover_column(i->c);
  }
  uncover_column(c);
  return false;
}
```

Zweizusammenhang

```
// Finding biconnected components of a connected graph.
// Based on the algorithm from EA1/2, but using union find
// for the second part (which is worse in theory, but saves
// some space, as we only have to manage one additional pointer
// per edge).
// Tested with 10765: Doves and Bombs

#include <cstdio>

using namespace std;

const int MAXV = 16*1024; // maximal number of vertices
const int MAXDEG = 20; // maximum number of edges per vertex

struct edge;
struct vertex {
  int deg;
  edge *adj[MAXDEG];
  int dfsid;
  edge *parent_link;
  void init () { deg = 0; dfsid = -1; }
};

struct edge {
  vertex *a, *b;
  bool treelink; // otherwise backward link
  edge *uf_parent; // pointer to union find parent

  vertex *other (vertex *v) {
    return (v == a) ? b : a;
  }

  // union operation (with included find)
  inline void uf_union (edge *e) {
    uf_find ()->uf_parent = e->uf_find ();
  }

  // find operation
  edge *uf_find () {
    if (uf_parent == this)
      return this;
    edge *rep = uf_parent->uf_find ();
    uf_parent = rep;
    return rep;
  }
};

int n; // number of vertices
vertex V[MAXV];

int m; // number of edges
edge E[MAXV * MAXDEG];

void initV (int howMany) {
  n = howMany;
  m = 0;
  for (int i = 0; i < n; ++i) V[i].init ();
}

void addEdge (vertex *v, vertex *u) {
  edge *e = E+m++;
  u->adj[u->deg++] = e;
  v->adj[v->deg++] = e;
  e->a = v;
  e->b = u;
}

int gid; // global id for dfs numbering
vertex *Vdfs[MAXV]; // vertices in dfs order
// initial dfs to sort edges in TREE and BACKW
void dfs_init (vertex *v) {
  Vdfs[gid] = v;
  v->dfsid = gid++;

  for (int i = 0; i < v->deg; ++i) {
    edge *e = v->adj[i];
    if (e == v->parent_link) continue; // never return ...

    e->uf_parent = e;
    vertex *u = e->other (v);
    if (u->dfsid >= 0) // already visited
      e->treelink = false;
    else {
      e->treelink = true;
      u->parent_link = e;
      dfs_init (u);
    }
  }
}

// calculate the biconnected components of the graph above.
// after execution the uf_find method of edge can be used to get
// a representative of the component the edge is in.
// further processing (counting components) is up to the caller.
// the graph has to be connected!
void bicc () {
  gid = 0;
  V->parent_link = 0;
  dfs_init (V);

  for (int i = 0; i < n; ++i) {
    vertex *v = Vdfs[i];
    for (int j = 0; j < v->deg; ++j) {
      edge *e = v->adj[j];
      if (e->treelink) continue; // only check backward links
      vertex *u = e->other (v);
      if (u->dfsid < v->dfsid) continue; // only edges pointing to v
      while (u != v) {
        edge *f = u->parent_link;
        if (f->uf_parent == f) // isolated
          u = f->other (u); // u = parent (u)
        else
          u = v; // break
        f->uf_union (e);
      }
    }
  }
}

// =========== actual program starts here ===============
#include <algorithm>
#include <set>

struct bomb {
  int idx;
  int num;
  inline bool operator< (const bomb &b) const {
    return (num > b.num) || ((num == b.num) && (idx < b.idx));
  }
};

bomb bombs[MAXV];

int main () {
  int N, M, i, j;
  while (scanf ("%d %d", &N, &M) && N > 0) {
    initV (N);
    while (scanf ("%d %d", &i, &j) && i >= 0)
      addEdge (V+i, V+j);

    bicc ();

    for (i = 0; i < n; ++i) {
      bombs[i].idx = i;
      vertex *v = V+i;
      set<edge *> s;
      for (j = 0; j < v->deg; ++j)
        s.insert (v->adj[j]->uf_find ());
      bombs[i].num = s.size ();
    }
    sort (bombs, bombs+n);
    for (i = 0; i < M; ++i)
      printf ("%d %d\n", bombs[i].idx, bombs[i].num);
    printf ("\n");
  }
  return 0;
}
```

Modulo-Exponentiation

```
int modexp(int x, int n, int m) { /* x^n % m */
  int r=1;
  x%=m;
  while (n) {
    if (n&1) r=(r*x)%m;
    x=(x*x)%m;
    n>>=1;
  }
}
```

Ungewichtetes Matching

```
//
// Maximum cardinality matching for bipartite graphs.
// Tested with 3126: Taxi Cab Scheme (acmicpc-live-archive).
// Might still contain minor bugs.
//


#include <iostream>
#include <cstdio>

using namespace std;

// solve max cardinality matching for graph G = (V+U, E)
// Complexity O(min(|U|,|V|) * |E|)

const int MAXV = 1024; // MAXV >= |U| + |V|

// ----- Algorithm input -----

// graph description, V = {0, ..., nv-1}, U = {nv, ..., nv+nu-1}
int nv, nu; // |V| and |U|
int edeg[MAXV]; // degree of a vertex v \in U+V
int E[MAXV][MAXV]; // nodes adjacent to v (only 0..edeg[v] valid)

inline void clearE () {
  for (int i = 0; i < MAXV; ++i) edeg[i] = 0;
}

inline void add_edge (int i, int j) {
  E[i][edeg[i]++] = j;
  E[j][edeg[j]++] = i;
}

// ----- Algorithm output -----
int mate[MAXV]; // the mate of a vertex in a matching (or -1)

// make i and j mates
inline void marry (int i, int j) {
  mate[i] = j;
  mate[j] = i;
}

// ----- Managed by the algorithm -----
int prev[MAXV]; // previous node (for traversing aug. path)

int iq, nq, Q[MAXV]; // helper array used queue like

// enqueue and adjust prev
inline void enq (int v, int pred) {
  Q[nq++] = v;
  prev[v] = pred;
}

// perform maximum cardinality matching on E and store result
// in mate. returns the cardinality of the matching.
int matching () {
  int n = nu + nv, naug = 0;
  for (int i = 0; i < n; ++i) mate[i] = -1;

  // greedy initialization for speedup (can be omitted):
  // marry neighbour with minimal degree
  for (int i = 0; i < nv; ++i) {
    int best = -1;
    for (int j = 0; j < edeg[i]; ++j)
      if (mate[E[i][j]] < 0 &&
          (best < 0 || edeg[E[i][j]] < edeg[best]))
        best = E[i][j];
    if (best >= 0) {
      ++naug;
      marry (i, best);
    }
  }

  // starting search for augmenting paths
  bool found = true;
  while (found) {
    found = false;
    for (int i = 0; i < n; ++i) prev[i] = -1;

    // find exposed vertices in V
    nq = iq = 0;
    for (int i = 0; i < nv; ++i)
      if (mate[i] < 0) enq (i, i);

    // start search
    while (iq < nq) {
      int v = Q[iq++];

      if (v >= nv && mate[v] < 0) {
        // found exposed target in U,
        // so augment and restart search

        marry (v, prev[v]);
        v = prev[v];
        while (prev[v] != v) {
          v = prev[v];
          marry (v, prev[v]);
          v = prev[v];
        }

        ++naug;
        found = true;
        break; // escape from BFS
      }
      else { // continue BFS style

        if (v >= nv) {
          // we are in U, and we already know, we have a mate
          if (prev[mate[v]] < 0)
            enq (mate[v], v);
        } else { // follow unvisited and unmatched!
          for (int j = 0; j < edeg[v]; ++j)
            if (prev[E[v][j]] < 0 && E[v][j] != mate[v])
              enq (E[v][j], v);
        }
      }
    }
  }

  return naug;
}


// ================== actual program ================

inline int mdist (int a, int b) {
  return a > b ? a-b : b-a;
}

int T, xs[MAXV], ys[MAXV], xe[MAXV], ye[MAXV];
int t, ts[MAXV], te[MAXV];

int main () {
  scanf ("%d", &T);
  while (T--) {
    scanf ("%d", &nv);
    nu = nv;
    for (int i = 0; i < nv; ++i) {
      // cin>>t>>c>>ts[i]>>xs[i]>>ys[i]>>xe[i]>>ye[i];
      scanf ("%d:%d %d %d %d %d", &t,ts+i,xs+i,ys+i,xe+i,ye+i);
      ts[i] += 60*t;
      te[i] = ts[i] + mdist(xs[i],xe[i]) + mdist (ys[i],ye[i]);
    }

    clearE ();
    for (int i = 0; i < nv; ++i) {
      for (int j = i+1; j < nv; ++j)
        if (te[i]+mdist(xe[i],xs[j])+mdist(ye[i],ys[j]) < ts[j])
          add_edge (i, j+nv);
    }

    printf ("%d\n", nv - matching ());
  }
  return 0;
}
```

Edmonds-Karp B

```
/*
 * Edmonds-Karp MaxFlow: worst-case O(m*|F|) and O(n*m^2)
 * n = nodes, m = edges, |F| = max flow
 * Tested with 10330
 */

#include <iostream>
#include <cmath>
#include <cstdio>

using namespace std;

const int maxnodes = 1000;
const int maxedges = 60000;
// each edge is needed twice (both directions)!
const int infty = 0x7fffffff;

int num_edges = 0, num_nodes = 0;
int start_node = 0, target_node = 0;

struct node {
  int edge; // index of first edge (or -1)
  int seen; // used for DFS
} nodes[maxnodes];

struct edge {
  int u, v; // start and end node
  int cap; // capacity
  int next; // next edge in adj list of u
  int rev; // index of reversed edge
  void init (int U, int V, int C, int N, int R) {
    u = U; v = V; cap = C; next = N; rev = R;
  }
} edges[maxedges];

// initialize for n nodes
void init (int n) {
  num_nodes = n;
  num_edges = 0;
  for (int i = 0; i < n; ++i)
    nodes[i].edge = -1;
}

// add edges between u and v, where cap1 is capacity for (u,v) and
// cap2 is for (v,u). cap? >= 0
void add_edge (int u, int v, int cap1, int cap2) {
  int e1 = num_edges++, e2 = num_edges++;
  edges[e1].init (u, v, cap1, nodes[u].edge, e2);
  edges[e2].init (v, u, cap2, nodes[v].edge, e1);
  nodes[u].edge = e1;
  nodes[v].edge = e2;
}

// helper routine for FF: augment
// returns: 0 in case of failure, capacity increase otherwise
#if 0
// DFS version: simple, but slower in theory
int augment (int v = start_node, int max_cap = infty) {
  if (v == target_node)
    return max_cap;

  // iterate on edges
  for (int e = nodes[v].edge; e >= 0; e = edges[e].next) {
    edge *ee = edges + e;
    if (ee->cap <= 0 || nodes[ee->v].seen > 0) continue;
    nodes[ee->v].seen = 1;
    int cap = augment (ee->v, max_cap <? ee->cap);
    if (cap > 0) { // let it flow . . .
      ee->cap -= cap;
      edges[ee->rev].cap += cap;
      return cap;
    }
  }
  return 0;
}
#else

int Q[maxnodes], qs, qe;

// BFS version: has Edmonds-Karp upper bound
int augment () {
  qs = qe = 0;
  Q[qe++] = start_node;
  while (qs < qe) {
    int v = Q[qs++];
    if (v == target_node) { // perform augmentation
      int cap = infty;
      for (int u = target_node; u != start_node; ) {
        edge *ee = edges + nodes[u].seen;
        cap <?= ee->cap;
        u = ee->u;
      }
      for (int u = target_node; u != start_node; ) {
        edge *ee = edges + nodes[u].seen;
        ee->cap -= cap;
        edges[ee->rev].cap += cap;
        u = ee->u;
      }
      return cap;
    }

    for (int e = nodes[v].edge; e >= 0; e = edges[e].next) {
      edge *ee = edges + e;
      if (ee->cap <= 0 || nodes[ee->v].seen >= 0) continue;
      nodes[ee->v].seen = e;
      Q[qe++] = ee->v;
    }
  }
  return 0;
}
#endif

// calculate max flow from start_node to target_node (see above)
// on graph (graph is destroyed) and returns value
// Algorithm: Ford-Fulkerson O(num_edges * value) worst case
int max_flow (int s) {
  int value = 0;
  int step = 1;
  while (step > 0) {
    for (int i = 0; i < num_nodes; ++i)
      nodes[i].seen = -1;
    nodes[s].seen = 1;
    step = augment ();
    value += step;
  }

  return value;
}


// =========== problem specific code ============

int main () {

  int n, m, a, b, c, d;

  while (scanf("%d", &n) == 1) {
    init (2*n+2);
#define HEAD(i) ((i)<<1)
#define TAIL(i) (((i)<<1)+1)

    for (int i = 0; i < n; ++i) {
      scanf ("%d", &a);
      add_edge (TAIL(i), HEAD(i), a, 0);
    }

    scanf ("%d", &m);
    while (m--) {
      scanf ("%d %d %d", &a, &b, &c);
      --a; --b;
      add_edge (HEAD(a), TAIL(b), c, 0);
    }

    start_node = num_nodes - 1;
    target_node = num_nodes - 2;
    scanf ("%d %d", &b, &d);
    while (b--) {
      scanf ("%d", &a);
      --a;
      add_edge (start_node, TAIL(a), infty, 0);
    }
    while (d--) {
      scanf ("%d", &a);
      --a;
      add_edge (HEAD(a), target_node, infty, 0);
    }

    printf ("%d\n", max_flow (num_nodes-1));
  }

  return 0;
}
```

Edmonds-Karp W

```
#include <iostream>
#include <vector>
#include <deque>
#include <utility>
#include <cstring>

using namespace std;

const int MAX_NODES = 100;

struct EDGE {
  int u, v, c, f;
  EDGE( int u, int v, int cap, int flow )
    : u(u), v(v), c(cap), f(flow) {}
  // bi-directional part
  EDGE *e;
  inline void flow(int change) { f += change; e->c -= change; }
};

int N;
vector< EDGE* > edge[MAX_NODES];
vector< EDGE* > alledges;
EDGE* pred[MAX_NODES];

int maxflowbfs(int s, int t)
{
  memset(pred, 0, sizeof(pred));
  pred[s] = (EDGE*)1;
  deque< pair< int, int > > queue;
  queue.push_back(make_pair(s, 0x7fffffff));
  while( !queue.empty() )
  {
    pair< int, int > node = queue.front();
    queue.pop_front();
    for( vector< EDGE* >::iterator it = edge[node.first].begin(),
         itend = edge[node.first].end(); it != itend; ++it )
    {
      bool used = false;
      EDGE *e = *it;
      if( e->u == node.first && pred[e->v] == NULL && e->f < e->c)
      {
        pred[e->v] = e;
        queue.push_back(make_pair(e->v,
                          node.second <? (e->c - e->f)));
        used = true;
      }
      if( e->v == node.first && pred[e->u] == NULL && e->f > 0 )
      {
        pred[e->u] = e;
        queue.push_back(make_pair(e->u, node.second <? (e->f)));
        used = true;
      }
      if( used && (e->u == t || e->v == t) )
      {
        if( queue.size() > 0 )
          node = queue.back();
        int n = t;
        EDGE *p;
        while( n != s )
        {
          p = pred[n];
          if( p->v == n )
          {
            p->flow(node.second);
            n = p->u;
          }
          else
          {
            p->flow(-node.second);
            n = p->v;
          }
        }
        return node.second;
      }
    }
  }
  return 0;
}


int main()
{
  int cn = 0;
  int s, t, E;
  while( cin >> N && N )
  {
    for( vector< EDGE* >::iterator it = alledges.begin(),
         itend = alledges.end(); it != itend; ++it )
      delete (*it);
    alledges.clear();
    for( int i = 0; i < N; ++i )
      edge[i].clear();

    cin >> s >> t >> E;
    --s; --t;
    for( int i = 0; i < E; ++i )
    {
      int u, v, c;
      cin >> u >> v >> c;
      --u; --v;
      EDGE *e1 = new EDGE(u, v, c, 0);
      EDGE *e2 = new EDGE(v, u, c, 0);
      e1->e = e2;
      e2->e = e1;
      edge[u].push_back(e1);
      edge[v].push_back(e1);
      alledges.push_back(e1);
      edge[u].push_back(e2);
      edge[v].push_back(e2);
      alledges.push_back(e2);
    }

    int flow = 0, add = 1;
    while( add > 0 )
    {
      add = maxflowbfs(s, t);
      flow += add;
    }

    cout << "Network " << (++cn) << endl;
    cout << "The bandwidth is " << flow << "." << endl << endl;
  }
  return 0;
}
```

Push-Relabel M

```
/* (c) 2006 Martin von Gagern
 * Maximum flow / minimum cut with preflow push relabel algo,
 * selecting maximal excess node for complete discharge
 * and using gap heuristic. Capacity matrix may be asymmetric.
 * App has to set up capacity matrix and neighbors relation.
 * This may be done using clear_graph and add_edge.
 */

#include <vector>
#include <cstring>

using namespace std;

const int MAX_NODES = 512;

int capacity[MAX_NODES][MAX_NODES];
int flow[MAX_NODES][MAX_NODES];
int gap[2*MAX_NODES];

struct node {
  vector<int> neighbors;
  vector<int>::iterator current;
  int height;
  int excess;
} nodes[MAX_NODES];

int max_flow(int n, int source, int target) {
  bool gap_heuristic = true;
  for (int i = 0; i != n; ++i) {
    nodes[i].height = 0;
    nodes[i].excess = 0;
    for (int j = 0; j < n; ++j) {
      flow[i][j] = 0;
    }
    nodes[i].current = nodes[i].neighbors.begin();
  }
  node& ns = nodes[source];
  ns.height = n;
  for (vector<int>::iterator i = ns.neighbors.begin(),
       e = ns.neighbors.end(); i != e; ++i) {
    int c = capacity[source][*i];
    flow[source][*i] = c;
    flow[*i][source] = -c;
    nodes[*i].excess = c;
    ns.excess -= c;
  }
  if (gap_heuristic) {
    for (int i = 0; i != 2*n; ++i) gap[i] = 0;
    gap[0] = n-1;
    gap[n] = 1;
  }
  for (;;) {
    int u, e = 0;
```

```
    for (int i = 0; i != n; ++i) {
      if (i == source || i == target || e >= nodes[i].excess)
        continue;
      e = nodes[i].excess;
      u = i;
    }
    if (!e) break;
    node& nu = nodes[u];
    int old_height = nu.height;

    // discharge u:
    while (nu.excess) {
      if (nu.current == nu.neighbors.end()) {
        // relabel u:
        int minh = 0x7fffffff;
        for (vector<int>::iterator i = nu.neighbors.begin(),
               e = nu.neighbors.end();
             i != e; ++i)
          if (capacity[u][*i] > flow[u][*i])
            minh <?= nodes[*i].height;
        nu.height = 1+minh;
        nu.current = nu.neighbors.begin();
      }
      int v = *nu.current;
      if (capacity[u][v] > flow[u][v] &&
          nu.height > nodes[v].height) {
        // push from u to v:
        int send = nu.excess <? (capacity[u][v] - flow[u][v]);
        flow[u][v] += send;
        flow[v][u] -= send;
        nodes[u].excess -= send;
        nodes[v].excess += send;
      }
      else
        ++nu.current;
    }

    if (nu.height == old_height || !gap_heuristic) continue;
    ++gap[nu.height];
    if (--gap[old_height]) continue;
    for (int j = 0; j != n; ++j)
      if (j != source && nodes[j].height > old_height)
        nodes[j].height >?= n+1;
    gap_heuristic = false;
  }
  return nodes[target].excess;
}

void clear_graph() {
  memset(capacity, 0, sizeof(capacity));
  for (int i = 0; i != MAX_NODES; ++i) nodes[i].neighbors.clear();
}

void add_edge(int src, int dst, int cap) {
  if (!capacity[src][dst] && !capacity[dst][src]) {
    nodes[dst].neighbors.push_back(src);
    nodes[src].neighbors.push_back(dst);
  }
  capacity[src][dst] += cap;
}
```

Unsigned Big Integer

```
#include <iostream>
#include <vector>
#include <cassert>

using namespace std;

// unsigned big integer
struct ubi {

  // typedefs
  typedef unsigned long ul;
  typedef unsigned short us;
  typedef vector<us> dt;
  typedef dt::iterator di;
  typedef dt::reverse_iterator dri;
  typedef dt::const_iterator ci;
  typedef dt::const_reverse_iterator cri;

  // constants
  static const ul radix = 10000;
  static const ubi zero, one, ten;

  // data fields
  dt d; // digits

  // constructors
  ubi() : d() { }
  ubi(ul i) { push(i); }

  // short inlined helpers
  ul len() const { return d.size(); }
  ul digit(ul i) const { return i < d.size() ? d[i] : 0; }
  void swap(ubi& b) { d.swap(b.d); }
  void push(ul i) { while (i) { d.push_back(i % radix);
                             i /= radix; } }
  void trim() { while (!d.empty() && !d.back()) d.pop_back(); }

  // longer helper methods
  int toInt() const;
  double toDouble() const;
  int cmp(const ubi& b) const;
  ul divmod(ul f);
  void divmod(const ubi& b, ubi* q, ubi* m) const;

  // basic implemented operators
  ubi& operator+= (const ubi& b);
  ubi& operator-= (const ubi& b);
  ubi& operator*= (ul f);
  ubi operator* (const ubi& b) const;

  // derived inlined operators
  ubi& operator= (ul i) { d.clear(); push(i); return *this;}
  ubi operator+ (const ubi& b) const { return ubi(*this) += b; }
  ubi operator- (const ubi& b) const { return ubi(*this) -= b; }
  bool operator< (const ubi& b) const { return cmp(b) < 0; }
  bool operator> (const ubi& b) const { return cmp(b) > 0; }
  bool operator<= (const ubi& b) const { return cmp(b) <= 0; }
  bool operator>= (const ubi& b) const { return cmp(b) >= 0; }
  bool operator== (const ubi& b) const { return cmp(b) == 0; }
  bool operator!= (const ubi& b) const { return cmp(b) != 0; }
  ubi operator* (ul f) const { return ubi(*this) *= f; }
  ubi& operator/= (ul f) { divmod(f); return *this; }
  ubi operator/ (ul f) const { return ubi(*this) /= f; }
  ul operator% (ul f) const { return ubi(*this).divmod(f); }
  ubi& operator*= (const ubi& b)
                { ubi r = *this * b; swap(r); return *this; }
  ubi operator/ (const ubi& b) const
                { ubi r; divmod(b, &r, 0); return r; }
  ubi& operator/= (const ubi& b)
                { ubi r = *this / b; swap(r); return *this; }
  ubi operator% (const ubi& b) const
                { ubi r; divmod(b, 0, &r); return r; }
  ubi& operator%= (const ubi& b)
                { ubi r = *this % b; swap(r); return *this; }
  ubi& operator++ () { return *this += ubi(1); }
  ubi& operator-- () { return *this -= ubi(1); }
  ubi operator++(int)
                { ubi r(*this); *this += ubi(1); return r; }
  ubi operator--(int)
                { ubi r(*this); *this -= ubi(1); return r; }
  ubi& operator<<=(int i)
                { while(i--) *this *= 2; return *this; }
  ubi& operator>>=(int i)
                { while(i--) *this /= 2; return *this; }
  ubi operator<<(int i) { return ubi(*this) <<= i; }
  ubi operator>>(int i) { return ubi(*this) >>= i; }

};

const ubi ubi::zero(0), ubi::one(1), ubi::ten(10);

ostream& operator<< (ostream& o, const ubi& b) {
  if (b.d.empty()) return o << '0';
  ubi::cri i = b.d.rbegin(), e = b.d.rend();
  o << *i;
  for (++i; i != e; ++i) {
    for (ubi::ul j = (*i >? 1) * 10; j < ubi::radix; j *= 10)
      o << '0';
    o << *i;
  }
  return o;
}

istream& operator>> (istream& i, ubi& b) {
  if (i.flags() & ios_base::skipws) i >> ws;
  bool bad = true;
  char c;
  b.d.clear();
  while (i.get(c) && c >= '0' && c <= '9') {
    bad = false;
    (b *= 10) += ubi(c - '0');
  }
  i.putback(c);
  if (bad) i.setstate(ios_base::badbit);
  return i;
}
```

```
int ubi::cmp(const ubi& b) const {
  if (len() != b.len()) return len() - b.len();
  for (cri ai = d.rbegin(), ae = d.rend(), bi = b.d.rbegin();
      ai != ae; ++ai, ++bi)
    if (*ai != *bi) return *ai - *bi;
  return 0;
}

int ubi::toInt() const {
  assert(*this <= ubi(0xffffffffUL));
  int res = 0;
  for (cri i = d.rbegin(), e = d.rend(); i != e; ++i)
    res = res * radix + *i;
  return res;
}

double ubi::toDouble() const {
  double res = 0.;
  for (cri i = d.rbegin(), e = d.rend(); i != e; ++i)
    res = res * radix + *i;
  return res;
}

ubi& ubi::operator+= (const ubi& b) {
  ul carry = 0;
  if (len() < b.len()) d.resize(b.len());
  for (ul i = 0; (i < b.len() || carry) && i < len(); ++i) {
    carry += d[i] + b.digit(i);
    d[i] = carry % radix;
    carry /= radix;
  }
  if (carry) d.push_back(carry);
  return *this;
}

ubi& ubi::operator-= (const ubi& b) {
  assert(*this >= b);
  ul carry = 0, i;
  for (i = 0; i < b.len() || carry; ++i) {
    if (i < b.len()) carry += b.d[i];
    assert(i < len());
    if (d[i] >= carry) {
      d[i] -= carry;
      carry = 0;
    }
    else {
      d[i] = radix + d[i] - carry;
      carry = 1;
    }
  }
  trim();
  return *this;
}

ubi& ubi::operator*= (ul f) {
  assert(f < radix);
  ul carry = 0;
  for (di i = d.begin(), e = d.end(); i != e; ++i) {
    carry += *i * f;
    *i = carry % radix;
    carry /= radix;
  }
  if (carry) d.push_back(carry);
  return *this;
}

ubi::ul ubi::divmod (ul f) {
  assert (f < radix);
  ul rem = 0;
  for (dri i = d.rbegin(), e = d.rend(); i != e; ++i) {
    rem = radix * rem + *i;
    *i = rem / f;
    rem %= f;
  }
  trim();
  return rem;
}

ubi ubi::operator* (const ubi& b) const {
  ubi res;
  res.d.resize(len() + b.len());
  for (ul i = 0; i < b.len(); ++i) {
    ul bdi = b.d[i], carry = 0;
    for (ul j = 0; j < len(); ++j) {
      carry += res.d[i+j] + bdi * d[j];
      res.d[i+j] = carry % radix;
      carry /= radix;
    }
    if (carry) res.d[i + len()] = carry;
  }
  res.trim();
  return res;
}

void ubi::divmod (const ubi& b, ubi* q, ubi* m) const {
  if (q != 0) q->d.clear();   // initialize quotioent to zero
  if (len() < b.len()) {
    if (m != 0) m->d = d;      // remainder = this
    return;
  }
  ubi r;
  for (cri i = d.rbegin(); i != d.rend(); ++i) {
    r.d.insert(r.d.begin(), *i); // add new digit to remainder
    ul head = radix * r.digit(b.len()) + r.digit(b.len()-1);
    ul minq = head / (b.d.back() + 1UL);
    ul maxq = head / b.d.back() + 1UL;
    if (maxq > radix) maxq = radix;
    while (minq + 1U != maxq) { // binary search for quotient digit
      ul medq = (minq + maxq) / 2;
      ubi p = b * medq;
      if (p > r) maxq = medq;
      else minq = medq;
    }
    if (q != 0 && (minq || !q->d.empty()))
      q->d.insert(q->d.begin(), minq);
    if (minq != 0) r -= b * minq;
  }
  if (m != 0) m->swap(r);
}

namespace std {
  template<> void swap<ubi>(ubi& a, ubi& b) {
    a.swap(b);
  }
}
```

suffix tree

```
#include <iostream>
#include <map>
#include <list>
#include <string>

using namespace std;

struct node;
struct edge;

const char fieldBreak = ' ';
string text;
int lines;
node* root;
node* base;
int begin, end;
int suffix; // starting position of current state
list<int> lineBreaks;
int nextLineBreak;

struct node {
  int lineNumber;
  node* suffixLink;
  map<char, edge*> edges;
  node(int line = -1) : lineNumber(line), suffixLink(0) { }
  ~node();
  edge* getEdge(char c) { return edges[c]; }
  void addEdge(edge* e);
  void dump(string indent);
};

struct edge {
  int begin, end;
  node* target;
  edge(int b, int e, node* t) : begin(b), end(e), target(t) {}
  ~edge() { delete target; }
  char charAt(int i) { return text[begin+i]; }
  int length() { return end - begin; }
  node* split(int initialLength) {
    edge* e = new edge(begin + initialLength, end, target);
    node* n = new node();
    end = begin + initialLength;
    target = n;
    n->addEdge(e);
    return n;
  }
  void dump(string indent);
};

inline node::~node() {
  for (map<char, edge*>::iterator i = edges.begin();
      i != edges.end(); ++i) delete i->second;
}
```

```
inline void node::addEdge(edge* e) {
  edges[e->charAt(0)] = e;
}

void node::dump(string indent) {
  if (lineNumber>=0) cout << "(" << lineNumber << ")";
  cout << endl;
  for (map<char, edge*>::iterator i = edges.begin();
       i != edges.end(); ++i) i->second->dump(indent);
}

void edge::dump(string indent) {
  cout << indent << "- \""
       << text.substr(begin, length()) << "\" ";
  target->dump(indent+" |");
}

// follow a suffix link, taking care of line numbers.
void followSuffix() {
  base = base->suffixLink;
  if (suffix == nextLineBreak) {
    if (!lineBreaks.empty()) {
      nextLineBreak = lineBreaks.front();
      lineBreaks.pop_front();
    }
    else nextLineBreak = -1;
    ++lines;
  }
  ++suffix;
}

// make canonical reference by folowing complete edges
void canonize() {
  if (base == 0 && begin != end) {
    base = root;
    ++begin;
  }
  if (begin == end) return;
  edge* e = base->getEdge(text[begin]);
  while (e->length() <= end-begin) {
    begin += e->length();
    base = e->target;
    if (begin == end) return;
    e = base->getEdge(text[begin]);
  }
}

// Test if matching transition exists,
// otherwise make state explicit.
node* testAndSplit() {
  char t = text[end];
  if (begin != end) { // implicit state
    edge* e = base->getEdge(text[begin]);
    if (e->charAt(end-begin) == t) // endpoint
      return 0;
    else // new explicit node
      return e->split(end-begin);
  }
  else { // explicit state
    if (base != 0 && base->getEdge(t) == 0)
      return base; // node is already explicit
    else
      return 0; // endpoint
  }
}

// update tree by adding character at position end.
void update() {
  node* oldr = root;
  node* r = testAndSplit();
  while (r != 0) {
    node* newLeaf = new node(lines);
    edge* newEdge = new edge(end, text.length(), newLeaf);
    r->addEdge(newEdge);
    if (oldr != root) oldr->suffixLink = r;
    oldr = r;
    followSuffix();
    canonize();
    r = testAndSplit();
  }
  if (oldr != root) oldr->suffixLink = base;
}

// Make all nodes on the boundary explicit.
// Also annotate them with the line number.
void makeExplicit() {
  node* oldr = root;
  while (begin!=end) {
    node* r = base->getEdge(text[begin])->split(end-begin);
    r->lineNumber = lines;
    if (oldr != root) oldr->suffixLink = r;
    oldr = r;
    followSuffix();
    canonize();
  }
  if (oldr != root) oldr->suffixLink = base;
  while (base != 0) {
    base->lineNumber = lines;
    followSuffix();
  }
}

void buildTree() {
  lines = 0;
  root = new node();
  base = root;
  begin = 0;
  suffix = 0;
  lineBreaks.clear();
  nextLineBreak = -1;
  for (end = 0; end < int(text.size()); ++end) {
    if (text[end] == fieldBreak) {
      if (nextLineBreak == -1) nextLineBreak = end;
      else lineBreaks.push_back(end);
    }
    canonize();
    update();
  }
  canonize();
  makeExplicit();
}

int main(int argc, char** argv) {
  for (int i = 1; i != argc; ++i)
    text += argv[i], text += fieldBreak;
  buildTree();
  root->dump("|");
  return 0;
}
```

suffix array

```
#include <iostream>
#include <cstring>
#include <string>
#include <cstdio>
#include <cmath>

using namespace std;

class Suffix
{
  private:
    char *str;
    int len;
    int *pos;

  public:
    static const char sentinel = -128;

    Suffix(char *s, int l = 0) : str(s)
    {
      len = (l == 0 ? strlen(str) : l);
      str[len++] = sentinel;
      str[len] = 0;
      int n = len >? 256;
      int *prm = new int[n], *count = new int[n];
      int *bh = new int[n+1];
      pos = new int[n];

      radix(prm, bh);
      bucket(prm, bh, count);

      delete bh;
      delete count;
      delete prm;
    }

    ~Suffix()
    {
      delete pos;
    }

    // get the position in the string of the i'th suffix
    int getpos(int i)
    {
      return pos[i];
    }

    // last = NULL:
    //  return the position of the lexicographically smallest
```

```
// prefix (lsp) of pat
// last != NULL:
//  return the index to the pos array of the
//  lexicographically smallest prefix of pat, in last the
//  index of the lexicographically largest prefix of pat
// -1 if not found
int find(char *pat, int *last = NULL)
{
  int patlen = strlen(pat);
  int L, R;
  int l = 0, r = len-1, m, c;
  c = strncmp(str+pos[0], pat, len-pos[0] <? patlen);




  if( c == 0 )
  {
    L = 0;
    r = 1;
  }
  else
  {
    while( r-l > 1 )
    {
      m = (l+r) / 2;
      c = strncmp(str+pos[m], pat, len-pos[m] <? patlen);
      if( c < 0 )
        l = m;
      else
        r = m;
    }
    L = r;
  }

  l = r-1; r = len-1;
  while( r-l > 1 )
  {
    m = (l+r + ((l+r) & 1)) / 2;
    c = strncmp(str+pos[m], pat, len-pos[m] <? patlen);
    if( c <= 0 )
      l = m;
    else
      r = m;
  }
  R = l;

  if( L > R )
    return -1;
  if( last == NULL )
    return pos[L];
  *last = R;
  return L;
}

void radix(int *prm, int *bh)
{
  for( int i = 0; i < 256; ++i )
    pos[i] = -1;
  for( int i = 0; i < len; ++i )
  {
    prm[i] = pos[(unsigned char)str[i]];
    pos[(unsigned char)str[i]] = i;
  }
  int c = 0;
  for( int i = 0; i < 256; ++i )
  {
    int p = pos[i];
    while( p != -1 )
    {
      int j = prm[p];
      prm[p] = c;
      bh[c] = (p == pos[i] ? 1 : 0);
      p = j;
      ++c;
    }
  }
  bh[len] = 1;
  for( int i = 0; i < len; ++i )
    pos[prm[i]] = i;
}


void bucket(int *prm, int *bh, int *count)
{
  int iend = (int)(log((double)len) / log(2.));
  for( int i = 0, h = 1; i < iend; ++i, h *= 2 )
  {
    for( int l = 0, r = 0; r < len; ++r )
    {
      if( bh[r] & 1 )
      {
        l = r;
        count[l] = 0;
      }
      prm[pos[r]] = l;
    }
    for( int l = 0, r = 0; r < len; ++r )
    {
      if( bh[r] & 1 ) l = r;
      int d = pos[r] - h;
      if( d < 0 || d >= len ) continue;
      prm[d] += count[prm[d]]++;
      if( count[prm[d]] != 1 ) bh[prm[d]] |= 2;
      if( (bh[prm[d]+1] & 1) == 0 ) count[prm[d]+1] = l;
    }
    for( int l = 0; l < len; ++l )
    {
      pos[prm[l]] = l;
      if(bh[l] & 2)
        bh[l] = 1;
    }
  }
 }
};


int main()
{
  char str[128*1024+1];
  char pat[1024];
  int k;
  scanf("%d\n", &k);

  while(k-- > 0)
  {
    fgets(str, sizeof(str), stdin);
    int len = strlen(str);
    str[len-1] = 0;
    Suffix suf(str, len-1);

    int q;
    scanf("%d\n", &q);

    while(q-- > 0)
    {
      fgets(pat, sizeof(pat), stdin);
      pat[strlen(pat)-1] = 0;
      int last;
      int first = suf.find(pat, &last);
      if(first == -1)
        cout << "n" << endl;
      else
        cout << "y" << endl;
    }

  }
  return 0;
}
```

sudoku.cc

```
//
// Sudoku solver
// This program solves 3285, but the code has been tested
// with two other sudoku problems
//

#include <iostream>

using namespace std;

// ==========================================
// Some speedups for using bit masks
// ==========================================

int bits[1<<9]; // number of bits in a number
int ibit[1<<9]; // index of lowest bit in a number

void init_bits () {
  for (int i = 0; i < (1<<9); ++i) {
    bits[i] = 0;
    ibit[i] = 0;

    for (int j = 8; j >= 0; --j) {
      if (i & (1<<j)) {
        ibit[i] = j;
        ++bits[i];
      }
    }
  }
```

```
}


// ===========================================
// Class for the state of a sudoku board
// ===========================================

// Some notes:
// - we use the number 0 to 8 (instead 1 to 9)
// - a field with a number in it is called fixed, else unfixed
struct state {

  int allowed[9][16]; // bitmasks containing numbers still alowed
  int used[9][16]; // The number used (0..8) or -1 (unfixed)
  int unfixed;    // number of unfixed fields (without a number)

  // constructor
  state () { init (); }

  // initialize fields
  void init () {
    for (int i = 0; i < 9; ++i)
      for (int j = 0; j < 9; ++j) {
        allowed[i][j] = (1 << 9)-1; // all numbers allowed
        used[i][j] = -1;         // TODO ??
      }
    unfixed = 9*9;
  }

  // fix the given field to the value v
  // return, whether fixing resulted in valid state
  bool fix (int i, int j, int v) {
    if (used[i][j] >= 0)
      return used[i][j] == v;

    if (!is_allowed(i, j, v))
      return false;

    used[i][j] = v;

    // simplify, i.e. remove this number from all unfixed
    // fields in the same row, col, or square

    int mask = ~(1 << v);

    // handle i-th row ([i][k])
    for (int k = 0; k < 9; ++k) {
      if (used[i][k] == v) {
        if (k != j) return false;
      }
      else if (used[i][k] < 0) {
        allowed[i][k] &= mask;
        if (allowed[i][k] == 0) return false;
        if (bits[allowed[i][k]] == 1)
          if (!fix (i, k, ibit[allowed[i][k]]))
            return false;
      }
    }

    // handle j-th col (the same as above, but [k][j])
    for (int k = 0; k < 9; ++k) {
      if (used[k][j] == v) {
        if (k != i) return false;
      }
      else if (used[k][j] < 0) {
        allowed[k][j] &= mask;
        if (allowed[k][j] == 0) return false;
        if (bits[allowed[k][j]] == 1)
          if (!fix (k, j, ibit[allowed[k][j]]))
            return false;
      }
    }

    // handle squares
    int ii = (i/3)*3, jj = (j/3)*3;
    for (int x = ii; x < ii+3; ++x)
      for (int y = jj; y < jj+3; ++y) {
        if (used[x][y] == v) {
          if (x != i || y != j) return false;
        }
        else if (used[x][y] < 0) {
          allowed[x][y] &= mask;
          if (allowed[x][y] == 0) return false;
          if (bits[allowed[x][y]] == 1)
            if (!fix (x, y, ibit[allowed[x][y]]))
              return false;
        }
      }

    // we just fixed one
    --unfixed;

    // recheck allowance
    return is_allowed (i, j, v);
  }

  // returns whether we could use the value v for field i, j
  bool is_allowed (int i, int j, int v) {
    return allowed[i][j] & (1 << v);
  }


  // NOT REQUIRED
  // print the current field, showing unfixed places as '.' and
  // using numbers 1..9 (for debugging only)
  void deb_print () {
    for (int i = 0; i < 9; ++i) {
      for (int j = 0; j < 9; ++j) {
        if (used[i][j] < 0)
          cout << '.';
        else
          cout << ((char)(used[i][j] + '1'));
      }
      cout << endl;
    }
  }

  // NOT REQUIRED
  // this is rather brute-force code for checking validity
  // of a field. return, whether complete and valid
  bool is_ok () {
    for (int i = 0; i < 9; ++i)
      for (int j = 0; j < 9; ++j)
        if (used[i][j] < 0 || used[i][j] >= 9)
          return false;

    for (int i = 0; i < 9; ++i)
      for (int j = 0; j < 9; ++j)
        for (int k = j+1; k < 9; ++k)
          if (used[i][j] == used[i][k] ||
              used[j][i] == used[k][i])
            return false;

    for (int x = 0; x < 9; x += 3)
      for (int y = 0; y < 9; y += 3)
        for (int i1 = 0; i1 < 3; ++i1)
          for (int j1 = 0; j1 < 3; ++j1)
            for (int i2 = 0; i2 < 3; ++i2)
              for (int j2 = 0; j2 < 3; ++j2) {
                if (i1 != i2 || j1 != j2)
                  if (used[x+i1][y+j1] == used[x+i2][y+j2])
                    return false;
              }

    return true;
  }

};

// ================================
// some globals
// ================================

int grid[9][9]; // the input grid (use 1..9, and 0 for unfixed)
int nodes = 0; // the number of nodes explored (profiling/testing)
int g_sol = 0; // the number of solution found

// ================================
// the actual solver
// ================================

void solve (state s) {
  ++nodes; // one more state explored

  // is this already a solution?
  if (s.unfixed == 0) {
    g_sol++;
    return;
  }

  // find the best field to work on next, i.e. the one with the
  // least number of possible numbers. This could be skipped
  // in favour of using the first or a random field
  // (but it would cost some performance)
  int besti = -1, bestj = -1;
  for (int i = 0; i < 9; ++i)
    for (int j = 0; j < 9; ++j) {
      if (s.used[i][j] < 0) {
        if (besti < 0 ||
           bits[s.allowed[i][j]] < bits[s.allowed[besti][bestj]]) {
```

```
        besti = i;
        bestj = j;
      }
    }
  }

  // explore all number allowed for this field
  for (int k = 0; k < 9; ++k) {
    if (s.is_allowed (besti, bestj, k)) {
      state t (s);
      if (t.fix (besti, bestj, k))
        solve (t);
    }
  }
}

// ========================================================
// code for checking the input grid
// this is only needed, if illegal grid can be in the input
//
// Note the the input grid should use numbers 1..9
// and 0 for unfixed!
// ========================================================

bool check_row (int i) {
  int used[10] = {0};
  for (int j = 0; j < 9; ++j) {
    int x = grid[i][j];
    if (x == 0) continue;
    if (used[x]) return false;
    used[x] = 1;
  }
  return true;
}

bool check_col (int j) {
  int used[10] = {0};
  for (int i = 0; i < 9; ++i) {
    int x = grid[i][j];
    if (x == 0) continue;
    if (used[x]) return false;
    used[x] = 1;
  }
  return true;
}

// a, b \in {0,1,2}
bool check_quad (int a, int b) {
  int used[10] = {0};

  for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 3; ++j) {
      int x = grid[a+i][b+j];
      if (x == 0) continue;
      if (used[x]) return false;
      used[x] = 1;
    }
  return true;
}

bool check_grid () {

  for (int i = 0; i < 9; ++i)
    if (!check_row (i)) return false;

  for (int i = 0; i < 9; ++i)
    if (!check_col (i)) return false;

  for (int a = 0; a < 3; ++a)
    for (int b = 0; b < 3; ++b)
      if (!check_quad (3*a, 3*b)) return false;

  return true;
}

// ================================
// Problem specific code
// ================================

// handle one problem instance,
// return false if end of input was reached
bool do_case () {

  // handling of label
  // (specific to this problem and probably not needed)
  int label;
  if (!(cin >> label && label > 0)) return false;
  cout << label << " ";

  // read state and store data in grid at the same time
  state s;
  char c;
  bool ok = true;
  for (int i = 0; i < 9; ++i)
    for (int j = 0; j < 9; ++j) {
      if (!(cin >> c)) return false;

      grid[i][j] = c - '0'; // use 1..9 here
      if (c != '0')
        ok = ok && s.fix (i, j, c - '1'); // but 0..8 here
    }

  g_sol = 0;
  if (ok) solve (s);
  cout << g_sol << endl;

  return true;
}

int main (int argc, char **argv) {

  init_bits ();

  while (do_case ());

  return 0;
}
```

Ungarische Methode (weighted matching)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
#include <cstdlib>

// Hungarian Algorithm aus Aufgabe 10149 (Yahtzee=Kniffel).
// Ermittelt Matching maximalen Gewichts in bipartitem Graphen.
// Es gilt immer v1[i1].cost+v2[i2].cost >= weight[i1][i2].
// Gesamtkosten werden minimiert (duales Problem).

using namespace std;

struct vertex {
  int matched; // Index des Partners im Matching
  int prev;  // Vorgaenger im alternierenden Baum
  int start; // Wurzel im alternierenden Pfad
  int cost;  // Knotenkosten fuer ungarischen Algo
  bool used; // Benutzt: bei Matchingsuche oder im Vertex Cover
  bool leaf; // Aktuell in der "queue" bei Matchingsuche
};

vertex v1[13], v2[13]; // die beiden Partitionen des Graphen
int e[13][13]; // die Excess-Matrix. 0 ist Kante im Eq. Subgraph

// Excess-Matrix berechn.: Summe der Knotenkosten - Kantengewicht
void excess(int weight[][13], int n) {
  for (int i1=0; i1<n; ++i1) for (int i2=0; i2<n; ++i2) {
    if ((e[i1][i2]=v1[i1].cost+v2[i2].cost-weight[i1][i2]) < 0) {
      cerr << "Negative excess!" << endl;
      exit(1);
    }
  }
}

// Im Equality Subgraph ein Matching max. Kardinalitaet finden
int matching(int n) {
  for (int i=0; i<n; ++i) {
    v1[i].matched=v2[i].matched=-1;
  }
  bool haspath, empty;
  int matchsize=0;
  for(;;) {
    for (int i=0; i<n; ++i) {
      v1[i].used=v1[i].leaf=v2[i].used=v2[i].leaf=false;
      if (v1[i].matched!=-1) continue;
      v1[i].start=i;
      v1[i].used=v1[i].leaf=true;
      v1[i].prev=-1;
    }

    haspath=false;
    empty=false;

    while (!empty) {

      // follow edges not in matching
      for (int i1=0; i1<n; ++i1) {
        if (!v1[i1].leaf) continue;
        v1[i1].leaf=false;
```

```
      for (int i2=0; i2<n; ++i2) {
        if (v2[i2].used || e[i1][i2]!=0 || v1[i1].matched==i2)
          continue;
        v2[i2].prev=i1;
        v2[i2].start=v1[i1].start;
        v2[i2].used=v2[i2].leaf=true;
        if (v2[i2].matched==-1) {
          v1[v2[i2].start].prev=i2;
          haspath=true;
          break;
        }
      } // for i2
    } // for i1

    if (haspath) break;
    empty=true;

    // follow edge in matching
    for (int i2=0; i2<n; ++i2) {
      if (!v2[i2].leaf) continue;
      v2[i2].leaf=false;
      int i1=v2[i2].matched;
      if (v1[i1].used) continue;
      v1[i1].prev=i2;
      v1[i1].start=v2[i2].start;
      v1[i1].used=v1[i1].leaf=true;
      empty=false;
    } // for i2

  } // while !empty

  if (!haspath) return matchsize;

  // now augment every path found
  for (int start=0; start<n; ++start) {
    if (v1[start].matched!=-1 || v1[start].prev==-1) continue;
    int i2=v1[start].prev, i1;
    do {
      i1=v2[i2].prev;
      v2[i2].matched=i1;
      v1[i1].matched=i2;
      i2=v1[i1].prev;
    } while (i1!=start);
    ++matchsize;
  }
  } // for(;;)
}

// v1[i1] definitiv nicht im cover
// => alle Kanten muessen anderen Endpunkt im Cover haben
int notincover1(int n, int i1) {
  int size=0;
  for (int i2=0; i2<n; ++i2) {
    if (e[i1][i2]!=0 || v2[i2].used) continue;
    v2[i2].used=true;
    ++size;
    size+=notincover1(n, v2[i2].matched);
  }
  return size;
}

// symmetrische variante
int notincover2(int n, int i2) {
  int size=0;
  for (int i1=0; i1<n; ++i1) {
    if (e[i1][i2]!=0 || v1[i1].used) continue;
    v1[i1].used=true;
    ++size;
    size+=notincover2(n, v1[i1].matched);
  }
  return size;
}

// Vertex Cover auf Equality Subgraph finden
void cover(int n) {
  int coversize=0, matchsize=0;
  for (int i=0; i<n; ++i) {
    v1[i].used=v1[i].leaf=v2[i].used=v2[i].leaf=false;
  }
  for (int i1=0; i1<n; ++i1) {
    if (v1[i1].matched==-1) coversize+=notincover1(n, i1);
  }
  for (int i2=0; i2<n; ++i2) {
    if (v2[i2].matched==-1) coversize+=notincover2(n, i2);
  }
  for (int i1=0; i1<n; ++i1) {
    if (v1[i1].matched==-1) continue;
    ++matchsize;
    if (v1[i1].used || v2[v1[i1].matched].used) continue;
    v1[i1].used=true;
    ++coversize;
  }
  if (matchsize!=coversize) {
    cerr << "matchsize " << matchsize
         << " != coversize " << coversize << endl;
    exit(1);
  }
}

// Kosten anpassen
void costchg(int n) {
  // Teil 1: Kosten sind minimaler Wert in
  //         nicht vom Cover abgedeckten Teil
  int eps=0x7fffffff;
  for (int i1=0; i1<n; ++i1) {
    if (v1[i1].used) continue;
    for (int i2=0; i2<n; ++i2) {
      if (v2[i2].used) continue;
      if (eps>e[i1][i2]) eps=e[i1][i2];
    }
  }

  if (eps==0) { // Gaebe Endlosschleife
    cerr << "eps==0!" << endl;
    exit(1);
  }

  // Teil 2: Gesamtkosten durch Umschichten reduzieren
  for (int i=0; i<n; ++i) {
    if (!v1[i].used) v1[i].cost-=eps;
    if (v2[i].used) v2[i].cost+=eps;
  }
}

// Steuermethode, ruft den ganzen Krempel da oben auf
int hungarian(int w[][13], int n, int* res) {
  for (int i1=0; i1<n; ++i1) {
    v1[i1].cost=v2[i1].cost=0;
    for (int i2=0; i2<n; ++i2) {
      if (v1[i1].cost<w[i1][i2]) v1[i1].cost=w[i1][i2];
    }
  }

#ifdef DEBUG
  cout << endl << "weights:\n";
  for (int i2=0; i2<n; ++i2) cout << '\t' << i2;
  for (int i1=0; i1<n; ++i1) {
    cout << endl << i1;
    for (int i2=0; i2<n; ++i2) cout << '\t' << w[i1][i2];
  }
  cout << endl;
#endif

  for(;;) {
    excess(w, n);
    if (matching(n)==n) break; // found maximum weight matching
    cover(n);
    costchg(n);
  }

  // Ergebnis liegt vor; in sinnvolle Form bringen
  int sum=0;
  for (int i1=0; i1<n; ++i1) {
    int i2=v1[i1].matched;
    sum+=(res[i2]=w[i1][i2]);
  }
  return sum;
}
```

seg_isect.cc

```
struct pos {
  int x, y;

  pos () {}
  pos (int x, int y): x(x), y(y) {}

  bool operator!= (const pos &p) const {
    return x != p.x || y != p.y;
  }
};

struct seg {
  pos a, b;
  int x, y, z;
  seg (const pos& aa, const pos& bb) : a(aa), b(bb) {
    x = aa.y - bb.y;
    y = bb.x - aa.x;
    z = aa.x * x + aa.y * y;
  }
```

```
  int sgn(const pos& c) const {
    int i = c.x * x + c.y * y - z;
    if (i > 0) return 1;
    if (i < 0) return -1;
    return 0;
  }
  int sgn(const seg& s) const {
    return sgn(s.a) * sgn(s.b);
  }
  bool intersects(const seg& s) const {
    return sgn(s) < 0 && s.sgn(*this) < 0;
  }
  bool onsegment(const pos& c) const {
    if (sgn(c)) return false;
    return
      (a.x <? b.x) <= c.x &&
      (a.x >? b.x) >= c.x &&
      (a.y <? b.y) <= c.y &&
      (a.y >? b.y) >= c.y;
  }
};
```

geom_vec2.h

```
#ifndef GEOM_VEC2_H
#define GEOM_VEC2_H

#include <cmath>

template<class T, class D = double>
struct vec2
{
  typedef T value_type;
  typedef D distance_type;

  T x,y;

  vec2 () {}
  vec2 (T f): x(f), y(f) {}
  vec2 (T x, T y): x(x), y(y) {}
  vec2 (const vec2 &v): x(v.x), y(v.y) {}
  vec2 &operator= (const vec2 &v)
    { x = v.x; y = v.y; return *this; }

  T &operator[] (int i) { return (i == 0) ? x : y; }
  const T &operator[] (int i) const { return (i == 0) ? x : y; }
  bool operator== (const vec2 &v) const
    { return (x == v.x) && (y == v.y); }
  bool operator!= (const vec2 &v) const
    { return (x != v.x) || (y != v.y); }
  bool operator< (const vec2 &v) const
    { return (x < v.x) && (y < v.y); }
  bool operator> (const vec2 &v) const
    { return (x > v.x) && (y > v.y); }
  bool operator<= (const vec2 &v) const
    { return (x <= v.x) && (y <= v.y); }
  bool operator>= (const vec2 &v) const
    { return (x >= v.x) && (y >= v.y); }
  bool is_similar (const vec2 &v, T epsilon)
    { return dist2 (v) < (epsilon*epsilon); }
  vec2 operator- () const
    { return vec2 (-x, -y); }
  vec2 &operator+= (const vec2 &v)
    { x += v.x; y += v.y; return *this; }
  vec2 &operator-= (const vec2 &v)
    { x -= v.x; y -= v.y; return *this; }
  vec2 &operator*= (T f) { x *= f; y *= f; return *this; }
  vec2 &operator/= (T f) { x /= f; y /= f; return *this; }
  D mag () const { return sqrt ((D) mag2 ()); }
  T mag2 () const { return x*x + y*y; }
  D dist (const vec2 &v) const { return sqrt ((D) dist2 (v)); }
  T dist2 (const vec2 &v) const
    { T dx = x - v.x, dy = y - v.y; return dx*dx + dy*dy; }
  vec2 &norm () { return operator/= (mag()); }
  vec2 &rot (D ang) { // counterclockwise, angle in radians
    D c = cos (ang), s = sin (ang);
    T tx = c*x - s*y;
    y = c*y + s*x;
    x = tx;
    return *this;
  }
  vec2 to_rot (D ang) const {
    vec2 v (*this);
    return v.rot (ang);
  }
};
template<class T, class D> inline vec2<T,D>
operator+ (const vec2<T,D> &v1, const vec2<T,D> &v2) {
  vec2<T,D> v (v1); return v += v2;
}
template<class T, class D> inline vec2<T,D>
operator- (const vec2<T,D> &v1, const vec2<T,D> &v2) {
  vec2<T,D> v (v1); return v -= v2;
}
template<class T, class D> inline vec2<T,D>
operator* (const vec2<T,D> &v1, T f) {
  vec2<T,D> v (v1); return v *= f;
}
template<class T, class D> inline vec2<T,D>
operator* (T f, const vec2<T,D> &v1) {
  vec2<T,D> v (v1); return v *= f;
}
template<class T, class D> inline vec2<T,D>
operator/ (const vec2<T,D> &v1, T f) {
  vec2<T,D> v (v1); return v /= f;
}
template<class T, class D> inline T
operator* (const vec2<T,D> &v1, const vec2<T,D> &v2) {
  return v1.x*v2.x + v1.y*v2.y;
}

#endif // GEOM_VEC2_H
```

geom_vec3.h

```
#ifndef GEOM_VEC3_H
#define GEOM_VEC3_H

#include <cmath>

template<class T, class D = double>
struct vec3
{
public:
  T x, y, z;

  vec3 () {}
  vec3 (T f): x(f), y(f), z(f) {}
  vec3 (T x, T y, T z): x(x), y(y), z(z) {}
  vec3 (const vec3 &v): x(v.x), y(v.y), z(v.z) {}

  inline vec3 &operator= (const vec3 &v)
    { x = v.x; y = v.y; z = v.z; return *this; }
  inline T &operator[] (int i)
    { return (i == 0) ? x : ((i == 1) ? y : z); }
  inline const T &operator[] (int i) const
    { return (i == 0) ? x : ((i == 1) ? y : z); }
  inline bool operator== (const vec3 &v) const
    { return (x == v.x) && (y == v.y) && (z == v.z); }
  inline bool operator!= (const vec3 &v) const
    { return (x != v.x) || (y != v.y) || (z != v.z); }
  inline bool operator< (const vec3 &v) const
    { return (x < v.x) && (y < v.y) && (z < v.z); }
  inline bool operator> (const vec3 &v) const
    { return (x > v.x) && (y > v.y) && (z > v.z); }
  inline bool operator<= (const vec3 &v) const
    { return (x <= v.x) && (y <= v.y) && (z <= v.z); }
  inline bool operator>= (const vec3 &v) const
    { return (x >= v.x) && (y >= v.y) && (z >= v.z); }
  inline bool is_similar (const vec3 &v, T epsilon)
    { return dist2 (v) < (epsilon*epsilon); }
  inline vec3 operator- () const
    { return vec3 (-x, -y, -z); }
  inline vec3 &operator+= (const vec3 &v)
    { x += v.x; y += v.y; z += v.z; return *this; }
  inline vec3 &operator-= (const vec3 &v)
    { x -= v.x; y -= v.y; z -= v.z; return *this; }
  inline vec3 &operator*= (T f)
    { x *= f; y *= f; z *= f; return *this; }
  inline vec3 &operator/= (T f)
    { x /= f; y /= f; z /= f; return *this; }
  inline D mag () const { return sqrt ((D) mag2 ()); }
  inline T mag2 () const { return x*x + y*y + z*z; }
  inline D dist (const vec3 &v) const
    { return sqrt ((D) dist2 (v)); }
  inline T dist2 (const vec3 &v) const {
    T dx = x - v.x, dy = y - v.y, dz = z - v.z;
    return dx*dx + dy*dy + dz*dz; }
  inline vec3 &norm () { return operator/= (mag()); }
};

template<class T, class D> inline vec3<T,D>
operator+ (const vec3<T,D> &v1, const vec3<T,D> &v2) {
  vec3<T,D> v (v1); return v += v2;
}
template<class T, class D> inline vec3<T,D>
operator- (const vec3<T,D> &v1, const vec3<T,D> &v2) {
  vec3<T,D> v (v1); return v -= v2;
}
template<class T, class D> inline vec3<T,D>
operator* (const vec3<T,D> &v1, T f) {
  vec3<T,D> v (v1); return v *= f;
```

```
}
template<class T, class D> inline vec3<T,D>
operator* (T f, const vec3<T,D> &v1) {
  vec3<T,D> v (v1); return v *= f;
}
template<class T, class D> inline vec3<T,D>
operator/ (const vec3<T,D> &v1, T f) {
  vec3<T,D> v (v1); return v /= f;
}
template<class T, class D> inline T
operator* (const vec3<T,D> &v1, const vec3<T,D> &v2) {
  return v1.x*v2.x + v1.y*v2.y + v1.z*v2.z;
}
template<class T, class D> inline vec3<T,D>
cross_product (const vec3<T,D> &v1, const vec3<T,D> &v2) {
  return vec3<T,D> (v1.y * v2.z - v1.z * v2.y,
                     v1.z * v2.x - v1.x * v2.z,
                     v1.x * v2.y - v1.y * v2.x );
}

#endif // GEOM_VEC3_H
```

geom_line2.h

```
#ifndef GEOM_LINE2_H
#define GEOM_LINE2_H

#include "geom_vec2.h"

template<class T, class D = double>
struct line2
{
  typedef vec2<T,D> V;
  typedef const V cV;
  V a, b;

  line2 () {}
  line2 (cV &a, cV &b): a(a), b(b) {}
  // same sign = same side of line; 0 = on line
  inline T side_of_line (const vec2<T,D> &v) const {
    vec2<T,D> d = b - a;
    return (v.x - a.x) * d.y - (v.y - a.y) * d.x;
  }
};

#endif // GEOM_LINE2_H
```

geom_lseg2.h

```
#ifndef GEOM_LSEG2_H
#define GEOM_LSEG2_H

#include "geom_vec2.h"

template<class T, class D = double>
struct lseg2
{
  typedef vec2<T,D> V;
  typedef const V cV;

  V a, b;

  lseg2 () {}
  lseg2 (cV &a, cV &b): a(a), b(b) {}

  // same sign = same side of line; 0 = on line (not seg!)
  inline T side_of_line (const vec2<T,D> &v) const {
    vec2<T,D> d = b - a;
    return (v.x - a.x) * d.y - (v.y - a.y) * d.x;
  }

  inline D length () const {
    return a.dist (b);
  }
};

#endif // GEOM_LSEG2_H
```

geom_circle.h

```
#ifndef GEOM_CIRCLE_H
#define GEOM_CIRCLE_H

#include "geom_vec2.h"

#include "geom_line2.h"

#include <algorithm>
using std::swap;

template<class T, class R = T>
struct circle
{
  typedef vec2<T, R> V;
  typedef const V cV;

  R r;
  V center;

  circle () {}
  circle (R r, cV &c = V(0)): r(r), center(c) {}
};

template<class T, class D, class Cont> unsigned
tang (const circle<T,D> &c, const vec2<T,D> &v,
      Cont &container) {
  D d = c.center.dist (v);
  if (d < c.r) return 0;
  vec2<T,D> dir = v - c.center;
  if (d == c.r) {
    swap (dir.x, dir.y);
    dir.x = -dir.x;
    container.push_back (line2<T,D> (v, v+dir));
    return 1;
  }
  D ang = acos (c.r/d);
  dir.norm ();
  dir *= c.r;
  container.push_back (line2<T,D>
                    (v, c.center + dir.to_rot (ang)));
  container.push_back (line2<T,D>
                    (v, c.center + dir.to_rot (-ang)));
  return 2;
}

template<class T, class D, class Cont> unsigned
tang (const vec2<T,D> &v, const circle<T,D> &c,
      Cont &container) {
  return tang (c, v, container);
}

const unsigned long TANG_INFTY = 0xffffffff;

template<class T, class D, class Cont> unsigned
tang (const circle<T,D> &c1, const circle<T,D> &c2,
      Cont &container) {
  if (c1.r < c2.r) return tang (c2, c1, container);
  D d = c1.center.dist (c2.center);
  if (c1.r > d + c2.r) return 0; // nested circles
  if (d == 0 && c1.r == c2.r) return TANG_INFTY; // identically
  vec2<T,D> dir = c2.center - c1.center;
  dir.norm ();
  if (c1.r == d + c2.r) { // touching nested circles
    vec2<T,D> x = c1.center + c1.r * dir;
    swap (dir.x, dir.y);
    dir.x = -dir.x;
    container.push_back (line2<T,D> (x, x + dir));
    return 1;
  }
  // outer tangents
  D ang1 = acos ((c1.r - c2.r) / d);
  D ang2 = ang1 - M_PI;
  vec2<T,D> p1 = c1.r * dir, p2 = -c2.r * dir;
  container.push_back (line2<T,D>(c1.center + p1.to_rot (ang1),
                                  c2.center + p2.to_rot (ang2)));
  container.push_back (line2<T,D>(c1.center + p1.to_rot (-ang1),
                                  c2.center + p2.to_rot (-ang2)));
  if (d < c1.r + c2.r) return 2;
  if (d == c1.r + c2.r) { // touching circles
    vec2<T,D> x = c1.center + c1.r*dir;
    swap (dir.x, dir.y);
    dir.x = -dir.x;
    container.push_back (line2<T,D> (x, x + dir));
    return 3;
  }
  // inner tangents
  ang1 = acos ((c1.r + c2.r) / d);
  container.push_back (line2<T,D>(c1.center + p1.to_rot (ang1),
                                  c2.center + p2.to_rot (ang1)));
  container.push_back (line2<T,D>(c1.center + p1.to_rot (-ang1),
                                  c2.center + p2.to_rot (-ang1)));
  return 4;
}

#endif // GEOM_CIRCLE_H
```

geom_poly.h

```
#ifndef GEOM_QUICKHULL_H
#define GEOM_QUICKHULL_H

#include "geom_vec2.h"
```

```
#include "euclid.h" // for gcd
#include <algorithm>
#include <iterator>

using namespace std;

template<typename RAI> int
grid_points_on_outline (RAI begin, RAI end) {
  typedef typename iterator_traits<RAI>::value_type vector_t;
  int count = 0;
  for (RAI it (begin); it != end; ++it) {
    vector_t next = (it + 1 == end) ? *begin : *(it + 1);
    if (*it == next)
      continue;
    else if (it->x == next.x)
      count += max (it->y - next.y, next.y - it->y);
    else if (it->y == next.y)
      count += max (it->x - next.x, next.x - it->x);
    else
      count += gcd (max (it->x - next.x, next.x - it->x),
                    max (it->y - next.y, next.y - it->y));
  }
  return count;
}

template<typename T, typename D> T
triangle_area_t2 (vec2<T,D>& v1, vec2<T,D>& v2, vec2<T,D>& v3) {
  return (v2.x - v1.x)*(v3.y - v1.y)-(v3.x - v1.x)*(v2.y - v1.y);
}

template<typename RAI>
typename iterator_traits<RAI>::value_type::value_type
poly_area_t2 (RAI begin, RAI end) {
  typename iterator_traits<RAI>::value_type::value_type
    ret_area = 0;
  for (RAI it (begin); it != end - 1; ++it)
    ret_area += triangle_area_t2 (*begin, *it, *(it+1));
  return max (ret_area, -ret_area);
}

<typename RAI, typename vector_t, typename T>
  RAI quick_hull_impl (RAI begin, RAI end,
                       vector_t a, vector_t b) {
  vector_t normal (a.y - b.y, b.x - a.x);
  T maxdist = 0;
  RAI mid (end);
  for (RAI it (begin); it != end; ++it){
    if (normal * (*it - a) > maxdist) {
       maxdist = normal * (*it - a);
       mid = it;
    }
  }
  if (mid == end) return begin;
  if (distance (begin, end) < 2) return end;
  vector_t left_normal (a.y - mid->y, mid->x - a.x);
  vector_t right_normal (mid->y - b.y, b.x - mid->x);
  swap (*mid, *(end-1)); // keep mid at the end while dividing
  RAI end_left (begin);
  for (RAI it (begin); it != end - 1; ++it)
    if (left_normal * (*it - a) > 0) {
      swap (*end_left, *it);
      ++end_left;
    }
  mid = end_left;
  swap (*mid, *(end-1)); // move mid point in the middle again
  RAI end_right (end_left+1);
  for (RAI it (end_left + 1); it != end; ++it)
    if (right_normal * (*it - b) > 0) {
      swap (*end_right, *it);
      ++end_right;
    }
  RAI nleft_end = quick_hull_impl<RAI, vector_t, T>
    (begin, end_left, a, *mid);
  RAI nright_end = quick_hull_impl<RAI, vector_t, T>
    (end_left + 1, end_right, *mid, b);
  RAI new_end (nleft_end);
  for (RAI it (mid); it != nright_end; ++it)
    swap (*new_end++, *it);
  return new_end;
}

template<typename RAI> RAI quick_hull (RAI begin, RAI end) {
  typedef typename iterator_traits<RAI>::value_type vector_t;
  typedef typename vector_t::value_type value_t;

  RAI min_pos (begin);
  RAI max_pos (begin);

  for (RAI it (begin+1); it != end; ++it) {
    if (it->x < min_pos->x) min_pos = it;
    else if (it->x > max_pos->x) max_pos = it;
    if (it->x == min_pos->x && it->y < min_pos->y)
      min_pos = it;
    else if (it->x == min_pos->x && it->y > max_pos->y)
      min_pos = it;
  }
  if (*min_pos == *max_pos) return ++begin;
  vector_t normal
    (min_pos->y - max_pos->y, max_pos->x - min_pos->x);

  // keep starting points save at begining and end...
  swap (*begin, *min_pos);
  if (max_pos == begin) swap (*min_pos, *(end - 1));
  else swap (*(end - 1), *max_pos);

  RAI a1 (begin + 1);
  RAI a2 (end - 2);

  while(a1 <= a2) {
    if ((*a1 - *begin) * normal < 0) {
      swap (*a1, *a2);
      --a2;
    } else
      ++a1;
  }

  // move end of starting line into position again
  swap (*a1, *(end - 1));

  RAI end_left = quick_hull_impl<RAI, vector_t, value_t>
    (begin + 1, a1, *begin, *a1);
  RAI end_right = quick_hull_impl<RAI, vector_t, value_t>
    (a1 + 1, end, *a1, *begin);

  // move second part of hull directly behind the first part...
  // and filter collinear points
  RAI last (begin+1);
  swap (*end_left, *a1);
  for (RAI it (begin+2); it != end_left+1; ++it) {
    vector_t normal (last->y-(last-1)->y, (last-1)->x-last->x);
    if (normal * (*it - *last) == 0)
      swap (*last, *it);
    else
      swap (*++last, *it);
  }

  for (RAI it (a1+1); it != end_right; ++it) {
    vector_t normal (last->y-(last-1)->y, (last-1)->x-last->x);
    if (normal * (*it - *last) == 0)
      swap (*last, *it);
    else
      swap (*++last, *it);
  }

  return last+1;
}

#endif
```

geom_dist.h

```
#ifndef GEOM_DIST_H
#define GEOM_DIST_H

#include "geom_vec2.h"
#include "geom_vec3.h"
#include "geom_line2.h"
#include "geom_lseg2.h"
#include "geom_circle.h"

template<class T, class D>
inline D dist (const vec2<T,D> &v1, const vec2<T,D> &v2)
  { return v1.dist (v2); }

template<class T, class D>
inline D dist (const vec3<T,D> &v1, const vec3<T,D> &v2)
  { return v1.dist (v2); }

template<class T, class D>
inline D dist (const vec2<T,D> &v, const line2<T,D> &l) {
  vec2<T,D> dir = l.b - l.a;
  D lambda = (dir * (v - l.a)) / (dir*dir);
  return dist (v, l.a + lambda*dir);
}
template<class T, class D>
inline D dist (const line2<T,D> &l, const vec2<T,D> &v)
  { return dist (v,l); }

template<class T, class D>
inline D dist (const vec2<T,D> &v, const lseg2<T,D> &l) {
  vec2<T,D> dir = l.b - l.a;
```

```
  D lambda = (dir * (v - l.a)) / (dir*dir);
  if (lambda <= 0) return dist (v, l.a);
  else if (lambda >= 1) return dist (v, l.b);
  else return dist (v, l.a + lambda*dir);
}
template<class T, class D>
inline D dist (const lseg2<T,D> &l, const vec2<T,D> &v)
  { return dist (v,l); }

template<class T, class D>
inline D dist (const vec2<T,D> &v, const circle<T,D> &c) {
  D d = v.dist (c.center);
  return (d <= c.r) ? 0 : (d - c.r);
}
template<class T, class D>
inline D dist (const circle<T,D> &c, const vec2<T,D> &v)
  { return dist (v, c); }

template<class T, class D>
inline D dist (const circle<T,D> &c1, const circle<T,D> &c2) {
  D d1 = dist (c1.center, c2.center);
  D d2 = c1.r + c2.r;
  return (d1 <= d2) ? 0 : (d1 - d2);
}
template<class T, class D>
inline D dist (const circle<T,D> &c, const line2<T,D> &l) {
  D d = dist (l, c.center);
  return (d <= c.r) ? 0 : (d - c.r);
}
template<class T, class D>
inline D dist (const line2<T,D> &l, const circle<T,D> &c) {
  return dist (c,l);
}
template<class T, class D>
inline D dist (const circle<T,D> &c, const lseg2<T,D> &l) {
  D d = dist (l, c.center);
  return (d <= c.r) ? 0 : (d - c.r);
}
template<class T, class D>
inline D dist (const lseg2<T,D> &l, const circle<T,D> &c) {
  return dist (c,l);
}
template<class T, class D>
inline D dist (const line2<T,D> &l1, const line2<T,D> &l2) {
  vec2<T,D> d1 = l1.a - l1.b, d2 = l2.a - l2.b;
  if ((d1.x*d2.y - d1.y*d2.x) != 0) return 0; // not parallel
  return dist (l1, l2.a);
}
template<class T, class D>
inline D dist (const lseg2<T,D> &l1, const lseg2<T,D> &l2) {
  if ((l1.side_of_line (l2.a) * l1.side_of_line (l2.b) <= 0) &&
      (l2.side_of_line (l1.a) * l2.side_of_line (l1.b) <= 0))
    return 0; // intersection
  D d1 = dist (l1, l2.a), d2 = dist (l1, l2.b);
  D d3 = dist (l2, l1.a), d4 = dist (l2, l1.b);
  if (d1 > d2) d1 = d2;
  if (d3 > d4) d3 = d4;
  return (d1 > d3) ? d3 : d1;
}
template<class T, class D>
inline D dist (const lseg2<T,D> &ls, const line2<T,D> &li) {
  if (li.side_of_line (ls.a) * li.side_of_line (ls.b) <= 0)
    return 0; // different sides or on line
  D d1 = dist (li, ls.a), d2 = dist (li, ls.b);
  return (d1 < d2) ? d1 : d2;
}
template<class T, class D>
inline D dist (const line2<T,D> &li, const lseg2<T,D> &ls) {
  return dist (ls, li);
}

#endif // GEOM_DIST_H
```

geom_ang.h

```
#ifndef GEOM_ANG_H
#define GEOM_ANG_H

#include "geom_vec2.h"
#include "geom_vec3.h"
#include "geom_line2.h"
#include "geom_lseg2.h"

// definitions for PI:
#ifndef M_PI
#define M_PI 3.14159265358979323846 // from math.h
// const double M_PI = 2.0 * acos (0);
#endif // ifndef M_PI

template<class T>
inline T deg2rad (T t) { return M_PI * t / 180.0; }

template<class T>
inline T rad2deg (T t) { return 180.0 * t / M_PI; }

/*****************************
 * all angles are in radians! *
 ******************************/

template<class T, class D>
inline D ang (const vec2<T,D> &v1, const vec2<T,D> &v2)
{
  T scal = v1*v2;
  if (scal == 0) return M_PI/2.0;
  return acos (scal / (v1.mag () * v2.mag ()));
}

// directed angle: returns angle for
// counterclockwise rotation from v1 onto v2
template<class T, class D>
inline D dir_ang (const vec2<T,D> &v1, const vec2<T,D> &v2) {
  T scal = v1*v2;
  D a = (scal==0) ? M_PI/2.0 : acos(scal/(v1.mag()*v2.mag()));
  T x = v1.x * v2.y - v1.y * v2.x;
  return (x >= 0) ? a : (2. * M_PI - a);
}

template<class T, class D>
inline D ang (const line2<T,D> &l1, const line2<T,D> &l2) {
  D a = ang (l1.a - l1.b, l2.a - l2.b);
  return (a <= M_PI/2.0) ? a : (M_PI - a);
}

#endif // GEOM_ANG_H
```

geom_isect.h

```
#ifndef GEOM_ISECT_H
#define GEOM_ISECT_H

#include "geom_vec2.h"
#include "geom_vec3.h"
#include "geom_line2.h"
#include "geom_lseg2.h"
#include "geom_circle.h"

#include "geom_dist.h"

#include <algorithm>
using std::swap;

const unsigned long ISECT_INFTY = 0xffffffff;

template<class T, class D, class Cont>
inline unsigned long isect
(const line2<T,D> &l1, const line2<T,D> &l2, Cont &container) {
  vec2<T,D> d1 = l1.b - l1.a, d2 = l2.b - l2.a;
  T det = d1.x*d2.y - d1.y*d2.x;
  if (det == 0) // parallel
    return (dist (l1, l2.a) == 0) ? ISECT_INFTY : 0;
  vec2<T,D> v (d2.y*l2.a.x - d2.x*l2.a.y,
               d1.x*l1.a.y - d1.y*l1.a.x);
  container.push_back (vec2<T,D>
    ((d1.x*v.x + d2.x*v.y)/det, (d1.y*v.x + d2.y*v.y)/det));
  return 1;
}

template<class T, class D, class Cont> inline unsigned long isect
(const line2<T,D> &li, const lseg2<T,D> &ls, Cont &container) {
  T a_side = li.side_of_line(ls.a), b_side = li.side_of_line(ls.b);
  if (a_side == 0) {
    if (b_side == 0) return ISECT_INFTY;
    else return container.push_back (ls.a), 1;
  }
  else if (b_side == 0)
    return container.push_back (ls.b), 1;
  if (a_side * b_side > 0) return 0;
  vec2<T,D> d1 = li.b - li.a, d2 = ls.b - ls.a;
  T det = d1.x*d2.y - d1.y*d2.x; // != 0
  vec2<T,D> v(d2.y*ls.a.x-d2.x*ls.a.y, d1.x*li.a.y-d1.y*li.a.x);
  container.push_back (vec2<T,D>
    ((d1.x*v.x + d2.x*v.y)/det, (d1.y*v.x + d2.y*v.y)/det));
  return 1;
}

template<class T, class D, class Cont> inline unsigned long isect
(const lseg2<T,D> &ls, const line2<T,D> &li, Cont &container) {
  return isect (li, ls, container);
}

template<class T, class D, class Cont> inline unsigned long isect
(const lseg2<T,D> &l1, const lseg2<T,D> &l2, Cont &container) {
```

```
  T l1_side_a = l1.side_of_line(l2.a);
  T l1_side_b = l1.side_of_line(l2.b);
  T l2_side_a = l2.side_of_line(l1.a);
  T l2_side_b = l2.side_of_line(l1.b);

  if (!((l1_side_a*l1_side_b) <= 0 && (l2_side_a*l2_side_b) <= 0))
    return 0;

  vec2<T,D> d1 = l1.b - l1.a, d2 = l2.b - l2.a;
  T det = d1.x*d2.y - d1.y*d2.x;
  if (det == 0) { // parallel
    if (l1_side_a != 0) return 0; // not on same line
#define __TRANS(_x) (d1.x * _x.x + d1.y * _x.y)
    T max1 = __TRANS(l1.a), min1 = __TRANS(l1.b);
    T max2 = __TRANS(l2.a), min2 = __TRANS(l2.b);
#undef __TRANS
    if (max1 < min1) swap (max1, min1);
    if (max2 < min2) swap (max2, min2);
    if (max1 < min2 || min1 > max2) return 0; // no intersection
    if (max1 == min2 || min1 == max2) { // touching
      if (l1.a == l2.a || l1.a == l2.b) container.push_back (l1.a);
      else if (l1.b==l2.a || l1.b==l2.b) container.push_back(l1.b);
      return 1;
    }
    return ISECT_INFTY; // full intersection
  }
  vec2<T,D> v(d2.y*l2.a.x-d2.x*l2.a.y, d1.x*l1.a.y-d1.y*l1.a.x);
  container.push_back (vec2<T,D>
    ((d1.x*v.x + d2.x*v.y)/det, (d1.y*v.x + d2.y*v.y)/det));
  return 1;
}

template<class T, class D, class Cont> inline unsigned long isect
(const line2<T,D> &l, const circle<T,D> &c, Cont &container) {
  vec2<T,D> dir = l.b - l.a;
  D lambda = (dir * (c.center - l.a)) / (dir*dir);
  vec2<T,D> p = l.a + lambda * dir;

  D d = p.dist (c.center);
  if (d > c.r) return 0;
  if (d == c.r) return container.push_back (p), 1;
  D ang = acos (d/c.r);
  if (p == c.center) {
    swap (dir.x, dir.y);
    dir.x = -dir.x;
  }
  else dir = p - c.center;

  dir.norm ();
  dir *= c.r;

  container.push_back (c.center + dir.to_rot (ang));
  container.push_back (c.center + dir.to_rot (-ang));
  return 2;
}

template<class T, class D, class Cont> inline unsigned long isect
(const circle<T,D> &c, const line2<T,D> &l, Cont &container) {
  return isect (l, c, container);
}

template<class T, class D, class Cont> inline unsigned long isect
(const lseg2<T,D> &l, const circle<T,D> &c, Cont &container) {
  const D epsilon = 0.000000001;

  vec2<T,D> dir = l.b - l.a;
  D lambda = (dir * (c.center - l.a)) / (dir*dir);
  vec2<T,D> p = l.a + lambda * dir;

  D d = p.dist (c.center);
  if (d > c.r) return 0;
  if (d == c.r) {
    if (dist (p, l) < epsilon) {
      container.push_back (p);
      return 1;
    }
    else return 0;
  }
  D ang = acos (d/c.r);
  if (p == c.center) {
    swap (dir.x, dir.y);
    dir.x = -dir.x;
  }
  else dir = p - c.center;

  dir.norm ();
  dir *= c.r;

  unsigned numi = 0;
  p = c.center + dir.to_rot (ang);
  if (dist (p, l) < epsilon) {
    container.push_back (p);
    numi++;
  }
  p  = c.center + dir.to_rot (-ang);
  if (dist (p, l) < epsilon) {
    container.push_back (p);
    numi++;
  }
  return numi;
}

template<class T, class D, class Cont> inline unsigned long isect
(const circle<T,D> &c, const lseg2<T,D> &l, Cont &container) {
  return isect (l, c, container);
}

template<class T, class D, class Cont> inline unsigned long isect
(const circle<T,D> &c1, const circle<T,D> &c2, Cont &container) {
  if (c1.r < c2.r) return isect (c2, c1, container);

  D d = c1.center.dist (c2.center);
  if (d > c1.r + c2.r || c1.r > d + c2.r) return 0;
  if (d == 0 && c1.r == c2.r) return ISECT_INFTY;
  if (d == c1.r + c2.r || c1.r == d + c2.r) {
    container.push_back
      (c1.center + (c2.center-c1.center).norm()*c1.r);
    return 1;
  }

  D ang = acos ((c1.r*c1.r + d*d - c2.r*c2.r) / (2.0 * c1.r * d));
  vec2<T,D> p = c2.center - c1.center;
  p.norm ();
  p *= c1.r;

  container.push_back (c1.center + p.to_rot (ang));
  container.push_back (c1.center + p.to_rot (-ang));
  return 2;
}

#endif // GEOM_ISECT_H
```

Formeln

Kreis: Umfang $U = 2\pi r$, Fläche $S = r^2\pi$
Kugel: Oberfl. $S = 4\pi R^2$, Volumen $V = \frac{4}{3}\pi R^3$
Dreieck: Ecken A, B, C, Seiten a, b, c gegenüber, Winkel α, β, γ an der Ecke.

- halber Umfang $s = (a + b + c)/2$
- Fläche $S = \frac{1}{2}ab\sin\gamma = \sqrt{s(s-a)(s-b)(s-c)}$
- Umkreisdurchmesser $2R = \frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma}$
- Inkreisradius $r = \sqrt{(s-a)(s-b)(s-c)/s}$
  $r = s\tan\left(\frac{\alpha}{2}\right)\tan\left(\frac{\beta}{2}\right)\tan\left(\frac{\gamma}{2}\right)$
- $c = a\sin\gamma/\sin\alpha$
- $c^2 = a^2 + b^2 - 2ab\cos\gamma$

Kongruenzen:

- $a^{p-1} \equiv 1 \pmod p$ falls p prim
- $a^{\phi(b)} = 1 \pmod b$
- Chinesischer Restsatz:
  - $x \equiv b_1 \pmod{m_1} \wedge x \equiv b_2 \pmod{m_2} \wedge \ldots$
  - $m = m_1 \cdot m_2 \cdot \ldots \qquad a_i = m/m_i$
  - x_j so dass $a_j \cdot x_j \equiv b_j \pmod{mj}$
  - $x = a_1 \cdot x_1 + a_2 \cdot x_2 + \ldots \pmod m$

Projektive Geometrie:

- $l_\infty = (0, 0, 1)$
- $F = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad F^\Delta = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$
- $\text{join}(p, q) = p \times q$
- $\text{meet}(l, m) = l \times m$
- $\text{parallel}(p, l) = (l \times l_\infty) \times p$
- $\text{orthogonal}(p, l) = (F^\Delta l) \times p$

Entfernungen auf der Kugel:

$$\begin{aligned} \text{versine}(x) :=& 1 - \cos x \\ \text{haversine}(x) :=& \frac{1-\cos x}{2} = \sin^2\frac{x}{2} \\ a =& \text{haversine}(\beta_2 - \beta_1) \\ b =& \cos(\beta_1)\cdot\cos(\beta_2)\cdot\text{haversine}(\lambda_2 - \lambda_1) \\ c =& 2\cdot\tan^{-1}\frac{\sqrt{a+b}}{\sqrt{1-a-b}} \\ d =& R\cdot c \end{aligned}$$

β Breite (Latitude), λ Länge (Longitude)

CGA FAQ (shortened)

2D Computations: Points, Segments, Circles, Etc.

How do I rotate a 2D point?

In 2D, you make (X, Y) from (x, y) with a rotation by angle t so: $X = x\cos t - y\sin t; Y = x\sin t + y\cos t$
As a 2x2 matrix this is very simple. If you want to rotate a column vector v by t degrees using matrix M, use $M = \begin{pmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{pmatrix}$ in the product Mv.
If you have a row vector, use the transpose of M (turn rows into columns and vice versa). If you want to combine rotations, in 2D you can just add their angles, but in higher dimensions you must multiply their matrices.

How do I find the distance from a point to a line?

Let the point be C (C_x, C_y) and the line be AB (A_x, A_y) to (B_x, B_y). Let P be the point of perpendicular projection of C on AB. The parameter r, which indicates P's position along AB, is computed by the dot product of AC and AB divided by the square of the length of AB:

$$r = \frac{AC \cdot AB}{\|AB\|^2}$$

r has the following meaning:

$r = 0$: $P = A$

$r = 1$: $P = B$

$r < 0$: P is on the backward extension of AB

$r > 1$: P is on the forward extension of AB

$0 < r < 1$: P is interior to AB

The length of a line segment in d dimensions, AB is computed by:

$$L = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2 + \ldots + (B_d - A_d)^2}$$

so in 2D:

$$L = \sqrt{(Bx - Ax)^2 + (By - Ay)^2}$$

and the dot product of two vectors in d dimensions, $U \cdot V$ is computed:

$$D = (U_x \cdot V_x) + (U_y \cdot V_y) + \ldots + (U_d \cdot V_d)$$

so in 2D:

$$D = (U_x \cdot V_x) + (U_y \cdot V_y)$$

So the equation above expands to:

$$r = \frac{(C_x - A_x)(B_x - A_x) + (C_y - A_y)(B_y - A_y)}{L^2}$$

The point P can then be found:

$$P_x = A_x + r(B_x - A_x); P_y = A_y + r(B_y - A_y)$$

And the distance from A to P is $r \cdot L$.
Use another parameter s to indicate the location along PC, with the following meaning:

$s < 0$: C is left of AB

$s > 0$: C is right of AB

$s = 0$: C is on AB

Compute s as follows:

$$s = \frac{(A_y - C_y)(B_x - A_x) - (A_x - C_x)(B_y - A_y)}{L^2}$$

Then the distance from C to P is $|s| \cdot L$.

How do I find intersections of 2 2D line segments?

This problem can be extremely easy or extremely difficult; it depends on your application. If all you want is the intersection point, the following should work:
Let A, B, C, D be 2-space position vectors. Then the directed line segments AB and CD are given by:

$$AB = A + r(B - A) \quad r \in [0, 1]$$
$$CD = C + s(D - C) \quad s \in [0, 1]$$

If AB and CD intersect, then $A + r(B - A) = C + s(D - C)$, or $A_x + r(B_x - A_x) = C_x + s(D_x - C_x) \wedge A_y + r(B_y - A_y) = C_y + s(D_y - C_y)$ for some $r, s \in [0, 1]$.
Solving the above for r and s yields

$$r = \frac{(A_y - C_y)(D_x - C_x) - (A_x - C_x)(D_y - C_y)}{(B_x - A_x)(D_y - C_y) - (B_y - A_y)(D_x - C_x)} \quad (1)$$
$$s = \frac{(A_y - C_y)(B_x - A_x) - (A_x - C_x)(B_y - A_y)}{(B_x - A_x)(D_y - C_y) - (B_y - A_y)(D_x - C_x)} \quad (2)$$

Let P be the position vector of the intersection point, then
$P = A + r(B - A)$ or $P_x = A_x + r(B_x - A_x) \wedge P_y = A_y + r(B_y - A_y)$
By examining the values of r and s, you can also determine some other limiting conditions:
If $0 \le r \le 1 \wedge 0 \le s \le 1$, intersection exists, if $r < 0$ or $r > 1$ or $s < 0$ or $s > 1$ line segments do not intersect.
If the denominator in (1) is zero, AB and CD are parallel. If the numerator in (1) is also zero, AB and CD are collinear.
If they are collinear, then the segments may be projected to the x- or y-axis, and overlap of the projected intervals checked.
If the intersection point of the 2 lines are needed (lines in this context mean infinite lines) regardless whether the two line segments intersect, then
If $r > 1$, P is located on extension of AB If $r < 0$, P is located on extension of BA If $s > 1$, P is located on extension of CD If $s < 0$, P is located on extension of DC
Also note that the denominators of (1) and (2) are identical.

How do I generate a circle through three points?

Let the three given points be a, b, c. Use $_x$ and $_y$ to represent x and y coordinates. The coordinates of the center $p = (p_x, p_y)$ of the circle determined by a, b, and c are:

$$A = b_x - a_x$$
$$B = b_y - a_y$$
$$C = c_x - a_x$$
$$D = c_y - a_y$$
$$E = A(a_x + b_x) + B(a_y + b_y)$$
$$F = C(a_x + c_x) + D(a_y + c_y)$$
$$G = 2(A(c_y - b_y) - B(c_x - b_x))$$
$$p_x = (DE - BF)/G$$
$$p_y = (AF - CE)/G$$

If G is zero then the three points are collinear and no finite-radius circle through them exists.
Otherwise, the radius of the circle is: $r^2 = (a_x - p_x)^2 + (a_y - p_y)^2$

2D Polygon Computations

How do I find the area of a polygon?

The signed area can be computed in linear time by a simple sum. The key formula is this:
If the coordinates of vertex v_i are x_i and y_i, twice the signed area of a polygon is given by

$$2A(P) = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1}).$$

Here n is the number of vertices of the polygon, and index arithmetic is mod n, so that $x_n = x_0$, etc. A rearrangement of terms in this equation can save multiplications and operate on coordinate differences, and so may be both faster and more accurate:

$$2A(P) = \sum_{i=0}^{n-1} (x_i (y_{i+1} - y_{i-1}))$$

Here again modular index arithmetic is implied, with $n \equiv 0$ and $-1 \equiv n - 1$.
This can be avoided by extending the `x[]` and `y[]` arrays up to `[n+1]` with `x[n]=x[0]`, `y[n]=y[0]` and `y[n+1]=y[1]`, and using instead

$$2A(P) = \sum_{i=1}^{n} (x_i (y_{i+1} - y_{i-1}))$$

To find the area of a planar polygon not in the x-y plane, use:

$$2A(P) = \left| N \cdot \sum_{i=0}^{n-1} (v_i \times v_{i+1}) \right|$$

How can the centroid of a polygon be computed?

The centroid (a.k.a. the center of mass, or center of gravity) of a polygon can be computed as the weighted sum of the centroids of a partition of the polygon into triangles. The centroid of a triangle is simply the average of its three vertices, i.e., it has coordinates $\frac{x_1 + x_2 + x_3}{3}$ and $\frac{y_1 + y_2 + y_3}{3}$. This suggests first triangulating the polygon, then forming a sum of the centroids of each triangle, weighted by the area of each triangle, the whole sum normalized by the total polygon area. This indeed works, but there is a simpler method: the triangulation need not be a partition, but rather can use positively and negatively oriented triangles (with positive and negative areas), as is used when computing the area of a polygon. This leads to a very simple algorithm for computing the centroid, based on a sum of triangle centroids weighted with their signed area. The triangles can be taken to be those formed by any fixed point, e.g., the vertex v_0 of the polygon, and the two endpoints of consecutive edges of the polygon: $(v_1, v_2), (v_2, v_3)$, etc. The area of a triangle with vertices a, b, c is half of this expression:

$$(b_x - a_x)(x_y - a_y) - (c_x - a_x)(b_y - a_y)$$

How do I find if a point lies within a polygon?

The essence of the ray-crossing method is as follows. Think of standing inside a field with a fence representing the polygon. Then walk north. If you have to jump the fence you know you are now outside the poly. If you have to cross again you know you are now inside again; i.e., if you were inside the field to start with, the total number of fence jumps you would make will be odd, whereas if you were ouside the jumps will be even.
The code below is from Wm. Randolph Franklin `<wrf@ecse.rpi.edu>` (see URL below) with some minor modifications for speed. It returns 1 for strictly interior points, 0 for strictly exterior, and 0 or 1 for points on the boundary. The boundary behavior is complex but determined; in particular, for a partition of a region into polygons, each point is "in" exactly one polygon.

```
int pnpoly(int npol, float *xp, float *yp,
  float x, float y) {
  int i, j, c = 0;
  for (i = 0, j = npol-1; i < npol; j = i++) {
    if ((((yp[i]<=y) && (y<yp[j])) ||
         ((yp[j]<=y) && (y<yp[i]))) &&
        (x <   (xp[j]-xp[i]) * (y-yp[i])
            / (yp[j]-yp[i]) + xp[i]))
      c = !c;
  }
  return c;
}
```

The code may be further accelerated, at some loss in clarity, by avoiding the central computation when the inequality can be deduced, and by replacing the division by a multiplication for those processors with slow divides. For code that distinguishes strictly interior points from those on the boundary, see [O'Rourke (C)] pp. 239-245.

How do I find the intersection of two convex polygons?

Unlike intersections of general polygons, which might produce a quadratic number of pieces, intersection of convex polygons of n and m vertices always produces a polygon of at most $(n+m)$ vertices. There are a variety of algorithms whose time complexity is proportional to this size, i.e., linear.
The first, due to Shamos and Hoey, is perhaps the easiest to understand. Let the two polygons be P and Q. Draw a horizontal line through every vertex of each. This partitions each into trapezoids (with at most two triangles, one at the top and one at the bottom). Now scan down the two polygons in concert, one trapezoid at a time, and intersect the trapezoids if they overlap vertically.
A more difficult-to-describe algorithm is in [O'Rourke (C)], pp. 252-262. This walks around the boundaries of P and Q in concert, intersecting edges. An implementation is included in [O'Rourke (C)].

How do I do a hidden surface test (backface culling) with 2D points?

$c = (x_1 - x_2) \cdot (y_3 - y_2) - (y_1 - y_2) \cdot (x_3 - x_2)$
$(x_1, y_1), (x_2, y_2), (x_3, y_3)$ are the first three points of the polygon.
If c is positive the polygon is visible. If c is negative the polygon is invisible (or the other way).

How do I find a single point inside a simple polygon?

Given a simple polygon, find some point inside it. Here is a method based on the proof that there exists an internal diagonal, in [O'Rourke (C), 13-14]. The idea is that the midpoint of a diagonal is interior to the polygon.

1. Identify a convex vertex v; let its adjacent vertices be a and b.
2. For each other vertex q do:
   (a) If q is inside avb, compute distance to v (orthogonal to ab).
   (b) Save point q if distance is a new min.
3. If no point is inside, return midpoint of ab, or centroid of avb.
4. Else if some point inside, qv is internal: return its midpoint.

How do I find the orientation of a simple polygon?

Compute the signed area (see above). The orientation is counterclockwise if this area is positive.
A slightly faster method is based on the observation that it isn't necessary to compute the area. Find the lowest vertex (or, if there is more than one vertex with the same lowest coordinate, the rightmost of those vertices) and then take the cross product of the edges fore and aft of it. Both methods are $O(n)$ for n vertices, but it does seem a waste to add up the total area when a single cross product (of just the right edges) suffices.
The reason that the lowest, rightmost (or any other such extreme) point works is that the internal angle at this vertex is necessarily convex, strictly less than pi (even if there are several equally-lowest points).

How can I triangulate a simple polygon?

Triangulation of a polygon partitions its interior into triangles with disjoint interiors. Usually one restricts corners of the triangles to coincide with vertices of the polygon, in which case every polygon of n vertices can be triangulated, and all triangulations contain $n-2$ triangles, employing $n-3$ "diagonals" (chords between vertices that otherwise do not touch the boundary of the polygon).
Triangulations can be constructed by a variety of algorithms, ranging from a naive search for noncrossing diagonals, which is $O(n^4)$, to "ear" clipping, which is $O(n^2)$ and relatively straightforward to implement [O'Rourke (C), Chap. 1], to partitioning into monotone polygons, which leads to $O(nlogn)$ time complexity [O'Rourke (C), Chap. 2; Overmars, Chap. 3], to several randomized algorithms—by Clarkson et al., by Seidel, and by Devillers, which lead to $O(nlog{*}n)$ complexity—to Chazelle's linear-time algorithm, which has yet to be implemented.

How can I find the minimum area rectangle enclosing a set of points?

First take the convex hull of the points. Let the resulting convex polygon be P. It has been known for some time that the minimum area rectangle enclosing P must have one rectangle side flush with (i.e., collinear with and overlapping) one edge of P. This geometric fact was used by Godfried Toussaint to develop the "rotating calipers" algorithm in a hard-to-find 1983 paper, "Solving Geometric Problems with the Rotating Calipers" (Proc. IEEE MELECON). The algorithm rotates a surrounding rectangle from one flush edge to the next, keeping track of the minimum area for each edge. It achieves $O(n)$ time (after hull computation).

Zahlen
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997

Bit	Primzahlen			
11:	1069	1489	1103	1907
12:	2857	4049	2389	3299
13:	6473	6301	4729	6553
14:	13331	13627	8887	10487
15:	23917	26017	26489	31393
16:	43151	58573	47497	40507
17:	66923	128153	67433	81047
18:	255217	238201	169181	147799
19:	457837	268171	447611	334643
20:	907717	557671	679879	983119
21:	1684303	1807121	1448611	2086361
22:	3378013	3611623	2213963	3451321
23:	4392811	6738383	4790761	5988271
24:	13527287	12098617	8909567	12988579
25:	23888309	32513609	32384159	23409973
26:	50789399	48387959	35259733	52329583
27:	124323799	131168899	83689583	93827221
28:	219235409	262103717	162859979	174477769
29:	501586847	282850171	507900853	285388897
30:	1013969347	1060034539	843006239	1007230039
31:	1728180893	1687579349	1211005057	1932475823
32:	4236450521	2922795553	3434538467	3151623199

33:	6529546559	8374655183
34:	8924041949	14752757779
35:	33590394983	24939543229
36:	37785000361	58177891333
37:	128013996403	94073178923
38:	193464303073	250411812137
39:	533410722863	504306960293
40:	652717088099	817892093449
41:	1546192209143	1497356018387
42:	3114668279743	2826178423253
43:	5369227577369	6006767950829
44:	17539251612637	13631131353953
45:	34577799055711	22583437184039
46:	68485147273163	66057533531681
47:	125623054600273	106029859827127
48:	269156173569677	170011377491161
49:	431813999018653	435753203240053
50:	739884587828783	1004823258673733
51:	1574905980117683	1810127575764713
52:	3269057202345953	2493116182663991
53:	6269846692367737	8150289866397551
54:	15738434214030233	13413771962395729
55:	35156421844413971	26322134309369051
56:	68051792586324221	54056661083105831
57:	110224948546351213	130859601029044709
58:	262346137621016479	258588362210910349
59:	437288670590619227	384789162888754259
60:	974903453460825019	705181567915808141
61:	2135536739405590819	1665857185867705813
62:	4456733320252833701	2653139764746672913
63:	5689998691521416221	7450346134407938609
64:	16637137928157708209	12156900775559726933

Mersenne-Primzahlen:

$2^2-1=$	3=	`0x3`
$2^3-1=$	7=	`0x7`
$2^5-1=$	31=	`0x1f`
$2^7-1=$	127=	`0x7f`
$2^{13}-1=$	8.191=	`0x1fff`
$2^{17}-1=$	131.071=	`0x1ffff`
$2^{19}-1=$	524.287=	`0x7ffff`
$2^{31}-1=$	2.147.483.647=	`0x7fffffff`
$2^{61}-1=$	2.305.843.009.213.693.951=	`0x1fffffffffffffff`

Potenzen und Fakultäten

n	2^n	$n!$	$\binom{n}{\lfloor n/2 \rfloor}$
1	2	1	1
2	4	2	2
3	8	6	3
4	16	24	6
5	32	120	10
6	64	720	20
7	128	5040	35
8	256	40320	70
9	512	362880	126
10	1024	3628800	252
11	2048	3.99e+07	462
12	4096	4.79e+08	924
13	8192	6.23e+09	1716
14	16384	8.72e+10	3432
15	**32768**	1.31e+12	6435
16	**65536**	2.09e+13	12870
17	131072	3.56e+14	24310
18	262144	6.40e+15	48620
19	524288	1.22e+17	92378
20	1.05e+06	2.43e+18	184756
21	2.10e+06	5.11e+19	352716
22	4.19e+06	1.12e+21	705432
23	8.39e+06	2.59e+22	1.35e+06
24	1.68e+07	6.20e+23	2.70e+06
25	3.36e+07	1.55e+25	5.20e+06
26	6.71e+07	4.03e+26	1.04e+07
27	1.34e+08	1.09e+28	2.01e+07
28	2.68e+08	3.05e+29	4.01e+07
29	5.37e+08	8.84e+30	7.76e+07
30	1.07e+09	2.65e+32	1.55e+08
31	**2.15e+09**	8.22e+33	3.01e+08
32	**4.29e+09**	2.63e+35	6.01e+08
33	8.59e+09	8.68e+36	1.17e+09
34	1.72e+10	2.95e+38	2.33e+09
35	3.44e+10	1.03e+40	4.54e+09
36	6.87e+10	3.72e+41	9.08e+09
37	1.37e+11	1.38e+43	1.77e+10
38	2.75e+11	5.23e+44	3.53e+10
39	5.50e+11	2.04e+46	6.89e+10
40	1.10e+12	8.16e+47	1.38e+11
41	2.20e+12	3.35e+49	2.69e+11
42	4.40e+12	1.41e+51	5.38e+11
43	8.80e+12	6.04e+52	1.05e+12
44	1.76e+13	2.66e+54	2.10e+12
45	3.52e+13	1.20e+56	4.12e+12
46	7.04e+13	5.50e+57	8.23e+12
47	1.41e+14	2.59e+59	1.61e+13
48	2.81e+14	1.24e+61	3.22e+13
49	5.63e+14	6.08e+62	6.32e+13
50	1.13e+15	3.04e+64	1.26e+14
51	2.25e+15	1.55e+66	2.48e+14
52	4.50e+15	8.07e+67	4.96e+14
53	9.01e+15	4.27e+69	9.73e+14
54	1.80e+16	2.31e+71	1.95e+15
55	3.60e+16	1.27e+73	3.82e+15
56	7.21e+16	7.11e+74	7.65e+15
57	1.44e+17	4.05e+76	1.50e+16
58	2.88e+17	2.35e+78	3.01e+16
59	5.76e+17	1.39e+80	5.91e+16
60	1.15e+18	8.32e+81	1.18e+17
61	2.31e+18	5.08e+83	2.33e+17
62	4.61e+18	3.15e+85	4.65e+17
63	**9.22e+18**	1.98e+87	9.16e+17
64	**1.84e+19**	1.27e+89	1.83e+18