Tiny KTH ACM Contest Template Library

$_{\text{tiny}}$KACTL version 1.305, 2004-02-24

Contents

C Operator Precedence and Associativity

<table>
<tr><td><code>() [] . -> x++ x--</code></td><td>→</td></tr>
<tr><td><code>++x --x + - ! ~ (cast) sizeof & *</code></td><td>←</td></tr>
<tr><td><code>* / %</code></td><td>→</td></tr>
<tr><td><code>+ -</code></td><td></td></tr>
<tr><td><code><< >></code></td><td></td></tr>
<tr><td><code>< <= > >=</code></td><td></td></tr>
<tr><td><code>== !=</code></td><td></td></tr>
<tr><td><code>&</code></td><td></td></tr>
<tr><td><code>^</code></td><td></td></tr>
<tr><td><code>|</code></td><td></td></tr>
<tr><td><code>&&</code></td><td></td></tr>
<tr><td><code>||</code></td><td></td></tr>
<tr><td><code>?:</code></td><td></td></tr>
<tr><td><code>= += -= *= /= %= &= ^= |= >>= <<=</code></td><td>←</td></tr>
<tr><td><code>,</code></td><td>→</td></tr>
</table>

Chapter 1

Contest

1.1 Facilities

Listing 1.1: Template.cc

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

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

using namespace std;
const enum {SIMPLE, FOR, WHILE} mode = NO;
bool debug = false;

void init() {
}

bool solve(int P) {
}

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

Listing 1.2: script.cc

11 lines,

```
echo 'g++ -Wall -o $1 $1.cc' > c
echo 'cat > $1.in' > i
echo 'cat > $1.ans' > o
echo './$1 < $1.in | tee $1.out' > t
echo './$1 | tee $1.out' > td
echo 'diff $1.out $1.ans' > d
echo 'a2ps --line-numbers=1 $1' > p
echo 'cp Template.cc $1.cc' > n

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

Listing 1.3: contest-keys.el.cc

25 lines,

```
(setq kactl-ext "cc")
(defun kactl-compile () (interactive)
  (shell-command (concat "g++ -ansi -lm -O2 -pedantic -Wall -o "
                   (file-name-sans-extension buffer-file-name)
                   " " buffer-file-name)))

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

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

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

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

Listing 1.4: contest-extras.el.cc

11 lines,

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

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

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

Chapter 2

Data Structures

2.1 Misc data structures

Listing 2.1: sets.cc

27 lines, <vector>

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

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

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

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

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

Listing 2.2: suffix array.cc

79 lines, <cstring>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Numerical datastructures

Listing 2.3: sign.cc

36 lines,

```
template <class T>
struct sign {
  static const T zero; // Requires declaration: const T sign¡T¿:: “
zero = T();
  T x; bool neg;
  operator sign(T _x = zero, bool _neg = false) : x(_x), neg(_ \
neg) { }
  bool operator <(const sign<T> &s) const {
    return neg==s.neg ? neg ? x>s.x : x<s.x : neg && !(x==zero&& \
s.x==zero);
  }
  bool operator ==(const sign<T> &s) const {
    return neg==s.neg ? x==s.x : x==zero&&s.x==zero;
  }
  sign<T> operator -() { return sign<T>(x, !neg); }
  sign<T> &addsub(bool add) {
    if (add) x+=s.x;
    else if (x<s.x) { T t=s.x; x = t-=x; neg=!neg; }
    else x-=s.x;
    return *this;
  }
  sign<T> &operator +=(const sign<T> &s) { return addsub(neg == \
s.neg); }
  sign<T> &operator -=(const sign<T> &s) { return addsub(neg != \
s.neg); }
```

```
  sign<T> &operator *=(const sign<T> &s) { x*=s.x, neg^=s.neg; \
return *this; }
  sign<T> &operator /=(const sign<T> &s) { x/=s.x, neg^=s.neg; \
return *this; }
};

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

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

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

Listing 2.4: rational.cc

81 lines, "gcd.cpp"

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

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

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

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

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

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

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

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

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

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

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

Listing 2.5: bigint.cc

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

```
/* if long longs are disallowed:
 * #define LSIZE 10000
 * #define LIMBDIGS 4
 * typedef int limb;    */
typedef long long limb;
typedef vector<limb> bigint;
typedef bigint::const_iterator bcit;
typedef bigint::reverse_iterator brit;
typedef bigint::const_reverse_iterator bcrit;
typedef bigint::iterator bit;

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

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

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

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

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

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

bool sub(bigint& a, const bigint& b) { /* Ret sign changed */
```

```
  if (a.size() < b.size()) a.resize(b.size());
  limb cy = 0; bit i = a.begin();
  for (bcit j = b.begin(); i != a.end() &&
       (cy || j < b.end()); ++j, ++i) {
    *i -= cy + (j < b.end() ? *j : 0);
    if ((cy = *i < 0)) *i += LSIZE;
  }
  if (cy) /* Only if sign may change. */
    while (i-- > a.begin()) *i = LSIZE - *i;
  return cy;
}

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

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

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

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

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

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

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

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

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

Chapter 3

Numerical

3.1 Number theory

Listing 3.1: euclid.cc

5 lines,

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

Listing 3.2: chinese.cc

5 lines, "euclid.cpp", "solves x mod m = a, x mod n = b, 0 <= x < mn, (m,n) = 1"

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

3.1.1 Primes

The 1000th prime is 7919. The first every 10000th primes are:

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

Listing 3.3: prime sieve.cc

41 lines, <algorithm>, <cmath>

```
using namespace std;

struct prime_sieve {
  static const int pregen = 3*5*7*11*13;
  typedef unsigned char uchar;
  typedef unsigned int uint;
  uint n, sqrtn;
  uchar *isprime;
  int *prime, primes;

  prime_sieve(int _n): n(_n), sqrtn((int)ceil(sqrt(1.0*n))) {
    int n0 = n >> 4;
    prime = new int[max(2775,(int)(1.12*n/log(n)))];
    prime[0] = 2; prime[1] = 3; prime[2] = 5;
    prime[3] = 7; prime[4] = 11; prime[5] = 13;
    primes = 6;
    isprime = new uchar[n0];
    memset(isprime, 255, n0);
    for (int j = 1, p = prime[j]; j < 6; p = prime[++j])
      for (int i=(p*p-3)>>4, s=(p*p-3)/2 & 7; i<=pregen; i+=(s+= \
p)>>3, s&=7)
        isprime[i] &= ~(1 << s);

    for (int d = pregen, b = pregen+1; b < n0; b += d, d <<= 1)
      memcpy(isprime + b, isprime + 1, (n0 < b + d) ? n0-b : d);
    for (uint p = 17, i = 0, s = 7; p < n; p += 2, i += ++s >> 3, \
s &= 7)
      if (isprime[i] & (1 << s)) {
        prime[primes++] = p;
        if (p < sqrtn) {
          int ii = i, ss = s + (p-1)*p/2;
          for (uint pp = p*p; pp < n; pp += p<<1, ss += p) {
            ii += (ss >> 3);
            ss &= 7;
            isprime[ii] &= ~(1 << ss);
          }
        } // end if
      } // end if
  } // end constructor
};
```

Listing 3.4: miller-rabin.cc

20 lines, "expmod.h", "mulmod.h"

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

Listing 3.5: pollard-rho.cc

34 lines, "gcd.h", "mulmod.h"

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

/* Returns a non-trivial factor of N, if any. (Note that if N is
 * prime, pollard_rho never returns, so this should be checked \
first.) */
template <class T>
inline T pollard_rho(T N, int maxiter = 500) {
  T x = rand() % N, y = x; /* replace rand by random number \
generator
```

```
                              of choice. */
  T d;
  int iter = 0;

  if (!(N & 1)) return 2; /* Check factor 2 */

  /* Check for small factor. While this _shouldn't_ be necessary,
     for some weird reason there's
   * trouble factoring the number 25 otherwise. Also, this gives \
a
   _considerable_ speed increase. */
  for (d = 3; d <= 9997; d += 2)
    if (!(N % d))
      return d;

  d = 1;
  while (d == 1) {
    /* Reseed if too many iterations passed. This shouldn't be
       necessary either, but seemed
     * to increase stability for Valladolid 10290 ("sum{i++} = N" \
)    */
    if (iter++ == maxiter) {
      x = y = rand() % N;
      iter = 0;
    }
    x = pollard_step(x, N);
    y = pollard_step(pollard_step(y, N), N);
    d = gcd(y - x, N);
    if (d == N) d = 1;
  }
  return d;
}
```

3.1.2 Perfect numbers

n is perfect iff $n = \frac{p(p+1)}{2}$, where $p = 2^k - 1$ is prime. First Mersenne primes are obtained for k = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497.

3.1.3 Josephus

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

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

Listing 3.6: josephus.cc

6 lines,

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

3.2 Linear Equations

Listing 3.7: solve linear.cc

54 lines,

```
const double NAN = 0.0/0.0;
const double EPS = 1e-12;

// Solves A*x = b. Returns rank.
int solve_linear(int n, double **A, double *b, double *x) {
  int row[n], col[n], undef[n], invrow[n], invcol[n];

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

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

  for (int i = rank; i-- ; ) {
    b[row[i]] /= A[row[i]][col[i]];
    A[row[i]][col[i]] = 1;
    for (int j = rank; j < n; ++j)
      if (abs(A[row[i]][col[j]]) > EPS)
        undef[i] = true;
    for (int j = i - 1; j >= 0; --j) {
      if (undef[i] && abs(A[row[j]][col[i]]) > EPS)
        undef[j] = true;
      else {
        b[row[j]] -= A[row[j]][col[i]] * b[row[i]];
        A[row[j]][col[i]] = 0;
      }
    }
  }

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

Listing 3.8: matrix inverse.cc

58 lines,

```
void matrix_inverse() {
  bool singular = false;
  double A[n][n]; // input
  double inv[n][n];
  int row[n], col[n];
  memset(inv, 0, sizeof(inv));

  for (int i = 0; i < n; ++i) {
    inv[i][i] = 1;
    row[i] = i;
    col[i] = i;
  }

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

  // backward pass:
  for (int i = n-1; i > 0; --i)
    for (int j = i-1; j >= 0; --j) {
      double v = A[row[j]][col[i]];
      // don't care about A at this point, just eliminate inv \
backward
      for (int k = 0; k < n; ++k)
```

```
      inv[row[j]][col[k]] -= v*inv[row[i]][col[k]];
    }

  int decol[n];
  for (int i = 0; i < n; ++i)
    decol[col[i]] = i;

  // inv[row[decol[i]]][j] is element (i,j) of solution (unless \
singular)
}
```

3.2.1 Calculating determinant

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

Listing 3.9: determinant.cc

24 lines,

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

Listing 3.10: int determinant.cc

30 lines, "euclid.cpp"

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

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

3.3 Optimization

3.3.1 Simplex method

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

Listing 3.11: simplex.cc

55 lines,

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

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

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

3.4 Polynomials

Listing 3.12: polynomial.cc

23 lines, <vector>

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

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

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

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

Listing 3.13: poly roots.cc

28 lines, "polynomial.cpp"

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

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

3.5 Bit manipulation hacks

Listing 3.14: bitmanip.cc

14 lines,

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

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

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

// next higher number with the same number of bits set
unsigned nexthi_same_count_ones(unsigned a) {
  unsigned c = (a & -a), r = a+c;
  return (((r ^ a) >> 2) / c) | r;
}
```

Chapter 4

Combinatorial

4.1 Misc

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

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

4.1.2 Knapsack

Usage `R res = knapsack<R>(n, C, costs, values [, bound = 500000]);`

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

Listing 4.1: knapsack.cc

27 lines, <vector>

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

  int C_max = (int) (scale * C) + 1;
  R max = R();
  vector<R> val_max(C_max, R(-1));
  val_max[0] = R();

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

  return max;
}
```

4.2 Permutations

4.2.1 Permutations to/from integers

Usage `int perm[n], x;`
`perm_to_int(x, perm, perm + n);`
`int_to_perm(x, perm, perm + n);`

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

Listing 4.2: intperm.cc

26 lines, <algorithm>

```
/* May well be replaced by a factorial lookup table, with the
 * appropriate changes in perm_to_int and int_to_perm. */
template <class Z>
Z factorial(int n) {
  Z r = Z(1);
  for (int i = n; i >= 2; --i) r *= i;
  return r;
}

/* Z is the number class, typically int or long long
 * It does not have to be RandomAccess!!
 * Complexity: O(n^2), where n is the number of elements in the \
permutation. */
template <class Z, class It>
void perm_to_int(Z& val, It begin, It end) {
  int x = 0, n = 0;
  for (It i = begin; i != end; ++i, ++n)
    if (*i < *begin) ++x;
  if (n > 2) perm_to_int<Z>(val, ++begin, end);
  else val = 0;
  val += factorial<Z>(n-1)*x;
}

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

4.3 Counting

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

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

Listing 4.3: choose.cc

11 lines, <algorithm>

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

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

  return c;
}
```

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

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

Listing 4.4: multinomial.cc

10 lines,

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

  return c;
}
```

4.3.3 Stirling numbers of the first kind

The Stirling numbers of the first kind $s(n, k)$ is defined as $(-1)^{n-k} c(n, k)$, where $c(n, k)$ is the number of permutations on n items with k cycles. It is given by

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

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

4.3.4 Stirling numbers of the second kind

The stirling number $S(n, k)$, i.e. in how many ways can n different items be put in k boxes with at least one item in every box, or mathematically speaking – the number of partitions of n elements into k partitions. It is given by

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

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

4.3.5 Bell numbers

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

The Bell numbers count the ways n elements can be partitioned.

4.3.6 Eulerian numbers

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

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

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

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

4.3.7 Second-order Eulerian numbers

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

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

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

4.3.8 Catalan numbers

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

4.3.9 Derangements

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

4.3.10 Involutions

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

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

Chapter 5

Graph

5.1 Misc basics

5.1.1 Bellman-Ford

Complexity $\mathcal{O}(VE)$

Listing 5.1: bellman ford.cc

30 lines,

```
template <class E, class M, class P, class D>
bool bellman_ford_2(E &edges, M &min, P &path, int start, int n, \
int m) {
  typedef typename M::value_type T;
  T inf(1<<29);

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

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

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

5.1.2 Shortest Tour

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

5.1.3 Kruskal

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

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

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

Listing – sets.cc, p. 3

Listing 5.2: kruskal.cc

37 lines, <algorithm>, <vector>, "../../datastructures/sets.cpp"

```
template<class V>
void kruskal( const V &graph, V &tree, int n ) {
  typedef typename V::value_type      E;
  typedef typename E::const_iterator  E_iter;
  typedef typename E::value_type::second_type D;

  sets sets(n);
  vector< pair< D,pair<int,int> > > edges;

  // Convert all edges into a single edge-list
  for( int i=0; i<n; i++ ) {
    for( E_iter iter=graph[i].begin(); iter!=graph[i].end(); \
iter++ ) {
      if( i < (*iter).first ) // Undirected: only use half of \
the edges
        edges.push_back( make_pair((*iter).second,
                                   make_pair(i,(*iter).first)) );
    }
  }

  // Clear tree
  for( int i=0; i<n; i++ )
    tree[i].clear();

  sort( edges.begin(), edges.end() );

  // Add edges in order of non-decreasing weight
  int numEdges = edges.size();
  for( int i=0; i<numEdges; i++ ) {
    pair<int,int> &edge = edges[i].second;

    // Add edge if the edge-endpoints aren't in the same set
    if( !sets.equal(edge.first, edge.second) ) {
      sets.link( edge.first, edge.second );
      tree[edge.first].push_back( make_pair(edge.second, edges[i] \
.first) );
      tree[edge.second].push_back( make_pair(edge.first, edges[i] \
.first) );
    }
  }
}
```

Listing 5.3: topo sort.cc

25 lines, <vector>, <queue>

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

5.2 Euler walk

5.2.1 Euler walk

Listing – euler walk.cc, p. 14

Complexity $\mathcal{O}(E)$

Usage `euler_walk( V &edges, int start, list<int> &path, bool cyclic )`

Find an eulerian walk in a directed graph, i.e. a walk traversing all edges exactly once.

The algorithm *assumes* that there exists an eulerian walk. If it does not exists, it will return any maximal path, not neccessarily the longest.

If the graph is not cyclic, the start node must be a node with $\deg_{out} - \deg_{in} = 1$.

`euler_walk` can be used to test if a graph has an eulerian walk by first finding a start-node (or any node if it is cyclic) and then checking if `path.size() == nrOfEdges+1`. But obviously this is slower than checking that all out degrees are equal to the in degrees (or exactly one vertex has an extraneous entering edge and another vertex an extraneous leaving edge) and that the graph is connected.

Set `cyclic=true` if the path found must be cyclic, this is mostly of internal use.

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

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

Lexicographic Path If the edges are sorted in lexicographic order for each vertex, the resulting path will be lexicographically ordered. This is accomplished by the algorithm, adding extra loops from the end first.

5.2.2 Chinese postman

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

Undirected chinese postman can be solved by computing a minimum weighted matching on the odd nodes of the graph (described in e.g. Edmonds and Johnson, "Matching, Euler Tours and the Chinese Postman.", Mathematical Programming 5: 88-124, 1973).

Directed chinese postman can be solved by using network flow techniques (also described in the previous reference).

5.3 De Bruijn Sequences

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

The shortest de Bruijn sequence of all words on 3 letters in the alphabet $\{0, 1\}$ which is lexicographically smallest is
00011101 (cyclic)
0001110100 (non-cyclic)

5.3.1 de Bruijn

Listing – deBruijn.cc, p. 14

Listing – deBruijn fast.cc, p. 15

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

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

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

The output is given as `cout`-statements.

5.4 Network Flow

Flow graphs are directed graphs with flow capacities on their edges.

To get quick access to the "back edge" of all egdes, a special flow edge struct is used in the network flow algorithms.

5.4.1 flow graph

Listing – flow graph.cc, p. 15

Usage `flow_add_edge( edges, source, dest, cap [, back_cap] );`

Flow graphs are constructed and updated by a couple of utility functions.

A flow graph should be an STL-container of `vector`s with `flow_edge`s (`map`s are not allowed).

Edges should be added using `flow_add_edge`.

Note that an edge *m*ust be added only once for each pair, simultaneously giving both forward and back capacity.

5.4.2 lift to front

Listing – lift to front.cc, p. 15

Note! This is a much more effective algorithm than Ford Fulkerson, even on bi-partite graphs, and suitable for any flow graph.

Note! Ford Fulkerson *is* faster if $En_{aug\ paths} < V^3$.

Usage `flow = lift_to_front(edges, source, sink);`

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

5.4.3 ford fulkerson

Listing – ford fulkerson.cc, p. 15

This is a DFS or BFS Ford Fulkerson which maximize the flow in the augmenting paths. The BFS is more robust but may be slower.

Usage The maximum flow is calculated by repetitive calls to `flow_increase1`: `while( ap = flow_increase1(edges, source, sink) ) flow+=ap;`

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

5.4.4 Flow constructions

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

Minimal path cover of a graph, determines a minimum set of paths to cover it.

5.5 Matching

5.5.1 hopcroft karp

Listing – hopcroft karp.cc, p. 16

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

5.5.2 max weight bipartite matching

Listing – mwbm.cc, p. 16

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

5.5.3 max weight bipartite matching of maximum cardinality

Listing – mwbm of max card.cc, p. 17

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

Graph Misc

Listing 5.4: euler walk.cc

43 lines, <list>

```
template<class V>
void euler_walk( V &edges, int start, list< int > &path, bool \
cyclic=false ) {
  int node = start, next_node;

  // Find a maximal path
  while( true ) {
    typename V::value_type &s = edges[node];

    path.push_back( node );

    if( s.empty() )
      break;

    // Follow the first edge and remove it
    next_node = *s.begin();
    s.erase( s.begin() );

    node = next_node;
  }

  // If no cyclic path was found, return an “empty” path, i.e. \
only the start node
  if( cyclic && node != start ) {
    path.clear();
    path.push_back( node );
    return;
  }

  // Extend path with cycles
  //for( list<int>::iterator iter = path.begin(); iter != path. \
end(); iter++ )

  for( list<int>::iterator iter = --path.end(); iter != path. \
begin(); ) {
    list<int>::iterator iter2 = iter; iter2--;
    node = *iter;

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

Listing 5.5: deBruijn.cc

49 lines, <iostream>, <vector>, "euler_walk.cpp"

```
using namespace std;

void deBruijn( int numSymbols, int L, char symbols[]) {
  int             numNodes;
  vector< vector<int> > edges;
  list<int>       path;

  // Number of nodes is numSymbols^(L-1)
  numNodes = 1;
  for( int i=0; i<L-1; i++ )
    numNodes *= numSymbols;

  // Create edges
  edges.resize( numNodes );

  for( int i=0; i<numNodes; i++ ) {
    edges[i].resize( numSymbols );

    for( int j=0; j<numSymbols; j++ )
      edges[i][j] = (i*numSymbols)%numNodes + j;
  }

  // Find euler walk
  path.clear();
  euler_walk( edges, 0, path );

  // Non-cyclic deBruijn sequences
  cout << "Non-cyclic:" << endl;

  string answer;
  for( list<int>::iterator iter = path.begin(); iter != path.end( \
); iter++ ) {
    int node = *iter;

    if( iter == path.begin() ) {
```

```
    int d = numNodes;

    for( int j=0; j<L-1; j++ ) {
      d/= numSymbols;
      answer += symbols[ node % numSymbols ];
    }
  } else
    answer += symbols[ node % numSymbols ];
}
cout << answer << endl;

// Cyclic deBruijn sequences
cout << "Cyclic:" << endl;
cout << answer.substr(0, answer.length()-(L-1)) << endl << \
endl;
}
```

Listing 5.6: deBruijn fast.cc

80 lines,

```
template<class V>
void euler_walk_dB( V &edges, int start, list< int > &path, int \
nSymb,
                    int nNodes )
{
  int node = start;

  while( true ) {
    int &s = edges[node];

    path.push_back( node );

    if( s == 0 )
      break;

    for( int i=0; i<nSymb; i++ ) {
      if( s & (1<<i) ) {
        node = (node*nSymb)%nNodes + i;
        s ^= (1<<i);
        break;
      }
    }
  }

  //for( list<int>::iterator iter = path.begin(); iter != path. \
end(); iter++ )

  for( list<int>::iterator iter = --path.end(); iter != path. \
begin(); ) {
    list<int>::iterator iter2 = iter; iter2--;
    node = *iter;

    int &s = edges[node];
    while( s != 0 ) {
      list<int> extra_list;
      euler_walk_dB( edges, node, extra_list, nSymb, nNodes );
      path.splice( iter, extra_list, extra_list.begin(), --extra_ \
list.end() );
    }
    iter = iter2;
  }
}

void deBruijn_fast( int nSymb, int L, char symbols[] ) {
  int             nNodes;
  vector< int >   edges;
  list<int>       path;

  nNodes = 1;
  for( int i=0; i<L-1; i++ )
    nNodes *= nSymb;

  edges.reserve( nNodes );
  for( int i=0; i<nNodes; i++ )
    edges.push_back( (1<<nSymb)-1 );

  euler_walk_dB( edges, 0, path, nSymb, nNodes );


  // Non-cyclic deBruijn sequences
  cout << "Non-cyclic:" << endl;

  string answer;
  for( list<int>::iterator iter = path.begin(); iter != path.end( \
); iter++ ) {
    int node = *iter;

    if( iter==path.begin() ) {
      int d = nNodes;

      for( int j=0; j<L-1; j++ ) {
        d/= nSymb;
        answer += symbols[ node % nSymb ];
      }
    } else
      answer += symbols[ node % nSymb ];
  }
  cout << answer << endl;

  // Cyclic deBruijn sequences
  cout << "Cyclic:" << endl;
  cout << answer.substr(0, answer.length()-(L-1)) << endl << \
endl;
}
```

Network Flow

Listing 5.7: flow graph.cc

20 lines, <vector>

```
typedef int Flow;

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

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

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

Listing 5.8: lift to front.cc

41 lines, "flow_graph.cpp"

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

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

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

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

Listing 5.9: ford fulkerson.cc

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

```
int mark[MAXNODES];
```

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

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

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

Matching

Listing 5.10: hopcroft karp.cc

104 lines, <queue>, <vector>, <utility>

```
template< class M >
bool hk_recurse( int b, int *lPred, vector<int> *rPreds, M match_ \
b ) {
  vector< int > L;

  L.swap( rPreds[b] );

  for( unsigned int i=0; i<L.size(); ++i ) {
    int a = L[i];
    int b2 = lPred[a];

    lPred[a] = -2;
    if( b2 == -2 )
      continue;
    if( b2 == -1 || hk_recurse(b2, lPred, rPreds, match_b) ) {
      match_b[b] = a;
      return true;
    }
  }
  return false;
}


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

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

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

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

  while( true ) {
    for( int i=0; i<n; i++ )
      lPred[ i ] = -1; // i is in the first layer
    for( int j=0; j<m; j++ )
      if( match_b[j]>=0 )
        lPred[match_b[j]] = -2; // remove from layer alltogether

    for( int j=0; j<m; j++ ) {
      rPreds[j].clear();
      rProc[j] = rNextProc[j] = false;
    }

    for( int i=0; i<n; i++ )
      if( lPred[i]==-1 )
        leftQ.push( i );

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

        rProc[b] = true;
        if( match_b[b] >= 0 ) {
          leftQ.push( match_b[b] );
          lPred[ match_b[b] ] = b;
        } else
          unmatchedQ.push( b );
      }
    }

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

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

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

Maximum Weight Bipartite Matching

Listing 5.11: mwbm.cc

143 lines, <vector>

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

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

  while( true ) {
    // Relax all edges out of a1
    for( L_iter e = edges[a1].begin(); e != edges[a1].end(); ++e \
) {
      int b = n+e->first;
      if( match_b[b-n] == a1 )
        continue;

      W db = dist[a1] + (pot[a1]+pot[b]-e->second);

      if( pred[b] < 0 || db < dist[b] ) {
        dist[b] = db; pred[b] = a1;
      }
    }

    // Select a node b with minimal distance db
```

```
    int b1 = -1;
    W db=0; // unused but makes compiler happy
    for( int b=n; b<n+m; b++ ) {
      if( !proc[b-n] && pred[b]>=0 && (b1<0 || dist[b]<db) ) {
        b1 = b;
        db = dist[b];
      }
    }

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


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

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

  // Augment path
  while( true ) {
    int vn = pred[v];

    if( v==vn )
      break;

    if( v>=n ) match_b[v-n] = vn;
    v = vn;
  }

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

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

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

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

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

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

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

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

  return true;
}
```

Listing 5.12: mwbm of max card.cc

27 lines, "max_weight_bipartite_matching.cpp"

```
template< class E, class M, class W >
void max_weight_b_m_of_max_card( E &edges, int n, int m, M & \
match_b,
                                 W &max_weight )
{
  typedef typename E::value_type L;
  typedef typename L::iterator L_iter;

  W Cmax = 0;
  for( int a=0; a<n; a++ )
    for( L_iter e = edges[a].begin(); e != edges[a].end(); ++e )
      Cmax = max( Cmax, max(e->second, -e->second) );
  Cmax = 1 + 2*max(n,m)*Cmax;

  for( int a=0; a<n; a++ )
    for( L_iter e = edges[a].begin(); e != edges[a].end(); ++e )
      e->second += Cmax;

  max_weight_bipartite_matching( edges, n, m, match_b, max_ \
weight, false );

  for( int b=0; b<m; b++ )
    if( match_b[b] >= 0 )
      max_weight -= Cmax;

  for( int a=0; a<n; a++ )
    for( L_iter e = edges[a].begin(); e != edges[a].end(); ++e )
      e->second -= Cmax;
}
```

Chapter 6

Geometry

6.1 Geometric primitives

Listing 6.1: point.cc

33 lines,

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

  T dx(R p) const { return p.x - x; }
  T dy(R p) const { return p.y - y; }
  double dist() const { return sqrt(dist2()); }
  double angle() const { return atan2(y, x); }

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

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

Listing 6.2: point3.cc

21 lines,

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

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

Listing 6.3: point line relations.cc

16 lines, "point.cpp"

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

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

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

6.1.1 Line intersection

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

Listing 6.4: line isect.cc

32 lines,

```
const double NO_ISECT = −1.0/0.0;

template <class P> inline
double line_isect(const P& A0, const P& A1, const P& B0, const P& \
 B1) {
  typedef P::value_type T;
  P dP1 = A1−A0, dP2 = B1−B0, dL = B0−A0;
  T det = dP1.cross(dP2), s = dL.cross(dP1), t = dL.cross(dP2);

  /* intersection between infinitely extending lines: */
  if (det == 0) return NO_ISECT;

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

  /* both: */
  if (det < 0) det = −det, t = −t, s = −s;
  if (!(t >= 0 && t <= det && s >= 0 && s <= det))
    return NO_ISECT;
  return (double)t / det;
}

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

6.1.2 Interval union

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

Listing 6.5: ival union.cc

12 lines, <algorithm>

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

6.1.3 Circle tangents

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

Listing 6.6: circle tangents.cc

10 lines,

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

  t1 = x + y;
  t2 = x − y;
  return a2==r2;
}
```

6.2 Triangles

6.2.1 Heron triangle area

Heron's formula: $A = \sqrt{p(p-a)(p-b)(p-c)}$, $p = \frac{a+b+c}{2}$.

6.2.2 Enclosing circle

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

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

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

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

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

Listing 6.7: incircle.cc

31 lines, "heron.cpp"

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

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

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

6.3 Polygons

6.3.1 Inside polygon

Complexity $\mathcal{O}(n)$

Usage `inside(polygon,nPts,point) == true;`

Determine whether a point is inside a polygon. If it is on an edge, standard computer graphics rules determine the returned value (inside above and to the left of the polygon, but not below or to the right). This is usually *not* the desired behaviour in contest geometry problems. Use `on_edge` in `pointline.cpp` to check if a point is on the edge.

Listing 6.8: inside.cc

11 lines, "point_line_relations.cc"

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

6.3.2 Polygon area

Twice the signed polygon area.

Listing 6.9: poly area.cc

7 lines, "point.cc"

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

6.3.3 Polyhedron volume

Signed polyhedron volume.

Listing 6.10: poly volume.cc

7 lines,

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

6.3.4 Polygon cut

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

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

Listing 6.11: poly cut.cc

17 lines, "line_isect.cpp"

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

6.3.5 Center of mass

Polygon and triangular center of mass.

Listing 6.12: center of mass.cc

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

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

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

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

    tri[0] = p[0];

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

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

6.4 Convex Hull

NOTE None of the Graham scans handle multiple coinciding points, so make sure the points are unique before calling!

6.4.1 Graham scan

Listing – convex hull.cc, p. 22

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

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

Swaps the points in `p` so the hull points are in order at the beginning.

Note! Handles colinear points on the hull

6.4.2 Three dimensional hull

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

Listing – convex hull space.cc, p. 22

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

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

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

6.4.3 Point inside hull

Listing – inside hull.cc, p. 22

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

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

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

6.4.4 Point inside hull simple

Listing – inside hull simple.cc, p. 22

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

Usage `inside_hull_simple(It begin, It end, point t)`

Determine whether a point `t` lies inside the hull given by begin and end. Colinear points are ok. If duplicate points exists, it will return *onedge* when it is inside. The hull must have at least one point. The result is given as: 1 inside, 0 onedge, -1 outside.

6.4.5 Hull diameter

Listing – hull diameter.cc, p. 22

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

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

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

6.4.6 Minimum enclosing circle

Listing – mec.cc, p. 23

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

Usage `bool mec(p, n, c, &i1, &i2, &i3[, eps]);`

Usage `double mec(p, n, c[, eps]);`

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

6.4.7 Line-hull intersect

Listing – line hull intersect.cc, p. 23

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

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

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

6.5 Minimum enclosing circle

See Convex hull, Minimum enclosing circle.

6.6 Voronoi diagrams

6.6.1 Simple Delaunay triangulation

Listing – delaunay simple.cc, p. 23

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

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

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

Note! Triangles may overlap if points are cocircular.

6.6.2 Convex hull Delaunay triangulation

Listing – delaunay hull.cc, p. 23

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

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

Returns an arbitrary triangulation if points are cocircular.

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

6.7 Nearest Neighbour

6.7.1 Divide and conquer

Listing – closest pair.cc, p. 24

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

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

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

6.7.2 Simpler method

Listing – closest pair simple.cc, p. 24

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

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

Hull

Listing 6.13: convex hull.cc

35 lines,

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

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

Listing 6.14: convex hull space.cc

50 lines, <set>

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

template <class V, class L>
bool convex_hull_space(V p, int n, L &trilist) {
  typedef typename V::value_type P3;
  typedef typename P3::coord_type T;
  typedef typename L::value_type I3;
  int a, b, c; // Find a proper tetrahedron
  for (a = 1; a < n; ++a) if (dist2(p[a]-p[0]) != T()) break;
  for (b = a + 1; b < n; ++b) if (dist2(cross(p[a]-p[0],p[b]-p[0] \
))) break;
  for (c = b + 1; c < n; ++c) if (dot(cross(p[a]-p[0],p[b]-p[0]), \
 p[c]-p[0])
                                  != T()) break;
  if (c >= n) return false;
  if (dot(cross(p[a]-p[0],p[b]-p[0]), p[c]-p[0]) > T()) swap(a, \
b);
  trilist.push_back(I3(0, a, b)); // Use it as initial hull
  trilist.push_back(I3(0, b, c));
  trilist.push_back(I3(0, c, a));
  trilist.push_back(I3(a, c, b));
  for (int i = 1; i < n; ++i) {
    typedef pair<int, int> I2;
    set< pair<int, int> > edges;
    P3 &P = p[i];
    {
      typename L::iterator it = trilist.begin();
      while (it != trilist.end()) {
        int a = it->a, b = it->b, c = it->c;
        P3 &A = p[a], &B = p[b], &C = p[c];
        P3 normal = cross(B-A, C-A);
        T d = dot(normal, P-A);
        if (d > T()) {
          edges.insert(make_pair(a, b));
          edges.insert(make_pair(b, c));
          edges.insert(make_pair(c, a));
          trilist.erase(it++); // ugly!!
        }
        else
          ++it;
      }
    }
    for (set<I2>::iterator it = edges.begin(); it != edges.end(); \
 ++it)
      if (edges.count(make_pair(it->second, it->first)) == 0)
        trilist.push_back(I3(i, it->first, it->second));
  }
  return true;
}
```

Listing 6.15: inside hull.cc

27 lines, "../geometry.h.cpp", "../pointline.cpp"

```
template <class V, class T>
int inside_hull_sub(const V &p, int n, const point<T> &t, int i1, \
 int i2) {
  if (i2 - i1 <= 2) {
    int s0 = sideof(p[0], p[i1], t);
    int s1 = sideof(p[i1], p[i2], t);
    int s2 = sideof(p[i2], p[0], t);
    if (s0 < 0 || s1 < 0 || s2 < 0)
      return -1;
    if (i1 == 1 && s0 == 0 || s1 == 0 || i2 == n - 1 && s2 == 0)
      return 0;
    return 1;
  }
  int i = (i1 + i2) / 2;
  int side = sideof(p[0], p[i], t);
  if (side > 0)
    return inside_hull_sub(p, n, t, i, i2);
  else
    return inside_hull_sub(p, n, t, i1, i);
}

template <class V, class T>
int inside_hull(const V &p, int n, const point<T> &t) {
  if (n < 3)
    return onsegment(p[0], p[n - 1], t) ? 0 : -1;
  else
    return inside_hull_sub(p, n, t, 1, n - 1);
}
```

Listing 6.16: inside hull simple.cc

27 lines,

```
// If the hull only consist of non-colinear points the \
degenerated-hull-check
// can be replaced with a onsegment-call if end-begin==2.

template <class It, class T>
int inside_hull_simple(It begin, It end, const point<T> &t) {
  bool on_edge = false;

  point<T> p, q; // degenerated hulls
  p = q = *begin; //

  for( It i=begin, j=end-1; i!=end; j=i++ ) {
    T d = cross(*i-*j,t-*j);
    if( d<0 )
      return -1;
    if( d==0 ) on_edge = true;

    p.x = min(p.x,i->x); // degenerated hulls
    p.y = min(p.y,i->y); //
    q.x = max(q.x,i->x); //
    q.y = max(q.y,i->y); //
  }

  // Extra check for degenerated hulls
  if( on_edge ) {
    if( t.x<p.x || t.x>q.x || t.y<p.y || t.y>q.y )
      return -1;
  }

  return on_edge ? 0:1;
}
```

Listing 6.17: hull diameter.cc

21 lines, "../point_ops.cpp"

```
template <class V>
double hull_diameter2(const V &p, int n, int &i1, int &i2) {
  typedef typename V::value_type::coord_type T;
  if (n < 2) { i1 = i2 = 0; return 0; }
  T m = 0;
  int i, j = 1, k = 0;
  // wander around
  for (i = 0; i <= k; i++) {
    // find opposite
    T d2 = dist2(p[j]-p[i]);
    while (j + 1 < n) {
      T t = dist2(p[j+1]-p[i]);
      if (t > d2) d2 = t; else break;
      j++;
    }
    if (i == 0) k = j; // remember first opposite index
    if (d2 > m) m = d2, i1 = i, i2 = j;
  }
  // cout << "first opposite: " << k << endl;
  return m;
}
```

Listing 6.18: mec.cc

22 lines, "hull_diameter.cpp", "../incircle.cpp"

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

template <class V, class P>
double mec(V p, int n, P &c, double eps = 1e-13) {
  int i1, i2, i3;
  mec(p, n, c, i1, i2, i3, eps);
  return dist(c, p[i1]);
}
```

Listing 6.19: line hull intersect.cc

52 lines, "../point.cpp", "../geometry.h.cpp", "../pointline.cpp"

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

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

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

Voronoi

Listing 6.20: delaunay simple.cc

26 lines, "../point.cpp"

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

Listing 6.21: delaunay hull.cc

14 lines, <vector>, <list>, "../point3.cpp", "../hull/convex_hull_space.cpp"

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

Closest pair

Listing 6.22: closest pair.cc

99 lines, <iterator>, <vector>

```
struct x_sort {
  template<class P>
  bool operator()(const P &p1, const P &p2) const
  { return p1.x < p2.x; }
};
struct y_sort {
  template<class P>
  bool operator()(const P &p1, const P &p2) const
  { return p1.x < p2.x; }
};

// Gives square distance of closest pair.
template<class V, class R>
double closestpair_sub(const V &p, int n, R xa, R ya, int &i1, \
int &i2) {
  typedef typename iterator_traits<V>::value_type P;
  vector< int > lefty, righty;

  // 2 or 3 points
  if( n <= 3 ) {
    // Largest dist is either between the two farthest in x or y.
    double a = dist2( p[xa[1]]-p[xa[0]] );
    if( n == 3 ) {
      double b = dist2( p[xa[2]]-p[xa[0]] );
      double c = dist2( p[xa[2]]-p[xa[1]] );

      return min(a,min(b,c));
    } else
      return a;
  }

  // Divide
  int split = n/2;
  double splitx = p[xa[split]].x;

  for( int i=0; i<n; i++ ) {
    if( p[ya[i]].x < splitx )
      lefty.push_back( ya[i] );
    else
      righty.push_back( ya[i] );
  }

  // Conquer
  int j1,j2;
  double a = closestpair_sub( p, split, xa, lefty.begin(), i1, \
i2 );
  double b = closestpair_sub( p, n-split, xa+split, righty.begin( \
), j1, j2 );

  if( b<a ) a = b, i1=j1, i2=j2;

  // Combine: Create strip (with sorted y)
  vector<int> stripy;

  for( int i=0; i<n; i++ ) {
    double x = p[ya[i]].x;
    if( x >= splitx-a && x <= splitx+a )
      stripy.push_back( ya[i] );
  }

  int nStrip = stripy.size();
  double a2 = a*a;

  // cout << "Combining " << nStrip << " points...";
  // cout.flush();

  for( int i=0; i<nStrip; i++ ) {
    P &p1 = p[stripy[i]];

    for( int j=i+1; j<nStrip; j++ ) { // This loop will be run < \
8 times/"i"
      P &p2 = p[stripy[j]];

      if( dy(p1,p2) > a )
        break;

      double d2 = dist2(p2-p1);
      if( d2<a2 ) {
        i1 = stripy[i];
        i2 = stripy[j];
        a2 = d2;
      }
    }
  }

  // cout << " done" << endl;
  return sqrt(a2);
}

template<class V> // R is random access iterators of point<T> \
s
double closestpair( const V &p, int n, int &i1, int &i2 ) {
  vector< int > xa, ya;

  if( n < 2 )
    throw "closestpair called with less than 2 points";

  xa.resize( n );
  ya.resize( n );
  isort( p, n, xa.begin(), x_sort() );
  isort( p, n, ya.begin(), y_sort() );

  return closestpair_sub( p, n, xa.begin(), ya.begin(), i1, i2 );
}
```

Listing 6.23: closest pair simple.cc

32 lines, "../datastructures/indexed.cpp", "../combinatorial/isort.cpp", <iterator>, <vector>

```
template<class R> // R is random access iterators of point<T> \
s
double closestpair_simple( R p, int n, int &i1, int &i2 ) {
  typedef typename iterator_traits<R>::value_type P;
  vector< int > idx;

  if( n < 2 )
    throw "closestpair called with less than 2 points";

  // Sort points "naturally" (i.e. first after x then after y)
  idx.resize( n );
  isort( p, n, idx.begin() );

  indexed<R, vector<int>::iterator > q(p, idx.begin() );

  double minDist = dist2(q[1]-q[0]);
  i1 = 0; i2 = 1;
  for( int i=0; i<N; i++ ) {
    double stopX = q[i].x+sqrt(minDist);
    for( int j=i+1; j<N; j++ ) {
      if( q[j].x >= stopX )
        break;
      double d = dist2(q[j]-q[i]);
      if( d<minDist ) {
        i1 = i;
        i2 = j;
        minDist = d;
      }
    }
  }

  return sqrt(minDist);
}
```

Index