ACM ICPC Reference

University of Pennsylvania

May 8, 2017

1 Data Structures

Treap (balanced binary search tree)

```
struct node {
  int v, key, size;
  node *c[2];
  void resize() { size = c[0]->size + c[1]->size + 1; }
};
node *newNode(int _v, node *n) {
  ++ref;
  pool[ref].v = _v, pool[ref].c[0] = pool[ref].c[1] = n, pool[ref].size = 1,
      pool[ref].key = rand();
  return &pool[ref];
}
struct Treap {
  node *root, *nil;
  void rotate(node *&t, int d) {
    node *c = t->c[d];
    t->c[d] = c->c[!d];
    c->c[!d] = t;
    t->resize(); c->resize();
    t = c;
  }
  void insert(node *&t, int x) {
    if (t == nil) t = newNode(x, nil);
    else {
      if (x == t->v) return;
      int d = x > t->v;
      insert(t->c[d], x);
      if (t->c[d]->key < t->key) rotate(t, d);
      else t->resize();
    }
  }
  void remove(node *&t, int x) {
    if (t == nil) return;
    if (t->v == x) {
      int d = t->c[1]->key < t->c[0]->key;
      if (t->c[d] == nil) {
        t = nil;
        return;
      }
      rotate(t, d);
      remove(t->c[!d], x);
    } else {
      int d = x > t->v;
      remove(t->c[d], x);
    }
    t->resize();
  }
  int rank(node *t, int x) {
    if (t == nil) return 0;
    int r = t->c[0]->size;
    if (x == t->v) return r + 1;
    if (x < t->v) return rank(t->c[0], x);
    return r + 1 + rank(t->c[1], x);
  }
  int select(node *t, int k) {
    int r = t->c[0]->size;
    if (k == r + 1) return t->v;
    if (k <= r) return select(t->c[0], k);
    return select(t->c[1], k - r - 1);
  }
  int size() {
    return root->size;
  }
  void init(int *a, int n) {
    nil = newNode(0, 0);
    nil->size = 0, nil->key = ~0U >> 1;
    root = nil;
  }
```

```
};
```

2 Geometry

Welzl's algorithm (minimum enclosing circle

```
// Minimum enclosing circle, Welzl's algorithm
// Expected linear time.
// If there are any duplicate points in the input, be sure to remove them
    first.
struct point {
  double x;
  double y;
};
struct circle {
  double x;
  double y;
  double r;
  circle() {}
  circle(double x, double y, double r): x(x), y(y), r(r) {}
};
circle b_md(vector<point> R) {
  if (R.size() == 0) {
    return circle(0, 0, -1);
  } else if (R.size() == 1) {
    return circle(R[0].x, R[0].y, 0);
  } else if (R.size() == 2) {
    return circle((R[0].x+R[1].x)/2.0,
                  (R[0].y+R[1].y)/2.0,
              hypot(R[0].x-R[1].x, R[0].y-R[1].y)/2.0);
  } else {
    double D = (R[0].x - R[2].x)*(R[1].y - R[2].y) - (R[1].x - R[2].x)*(R[0].y
        - R[2].y);
    double p0 = (((R[0].x - R[2].x)*(R[0].x + R[2].x) + (R[0].y - R[2].y)*(R
        [0].y + R[2].y)) / 2 * (R[1].y - R[2].y) - ((R[1].x - R[2].x)*(R[1].x
        + R[2].x) + (R[1].y - R[2].y)*(R[1].y + R[2].y)) / 2 * (R[0].y - R[2].
        y))/D;
    double p1 = (((R[1].x - R[2].x)*(R[1].x + R[2].x) + (R[1].y - R[2].y)*(R
        [1].y + R[2].y)) / 2 * (R[0].x - R[2].x) - ((R[0].x - R[2].x)*(R[0].x
        + R[2].x) + (R[0].y - R[2].y)*(R[0].y + R[2].y)) / 2 * (R[1].x - R[2].
        x))/D;
    return circle(p0, p1, hypot(R[0].x - p0, R[0].y - p1));
  }
}
circle b_minidisk(vector<point>& P, int i, vector<point> R) {
  if (i == P.size() || R.size() == 3) {
    return b_md(R);
  } else {
    circle D = b_minidisk(P, i+1, R);
    if (hypot(P[i].x-D.x, P[i].y-D.y) > D.r) {
      R.push_back(P[i]);
      D = b_minidisk(P, i+1, R);
    }
    return D;
  }
}

// Call this function.
circle minidisk(vector<point> P) {
  random_shuffle(P.begin(), P.end());
  return b_minidisk(P, 0, vector<point>());
}
```

Monotone chain (convex hull) and rotating calipers (farthest pair)

```
typedef long double gtype;
const gtype pi = M_PI;
typedef complex<gtype> point;
#define x real()
#define y imag()
#define polar(r, t) polar((gtype) (r), (t))
// vector
#define rot(v, t) ( (v) * polar(1, t) )
#define crs(a, b) ( (conj(a) * (b)).y )
#define dot(a, b) ( (conj(a) * (b)).x )
#define pntLinDist(a, b, p) ( abs(crs((b)-(a), (p)-(a)) / abs((b)-(a))) )
bool cmp_point(point const& p1, point const& p2) {
    return p1.x == p2.x ? (p1.y < p2.y) : (p1.x < p2.x);
}

// O(n.log(n)) - monotone chain
vector<point> mcH;
void monotoneChain(vector<point> &ps) {
    vector<point> p(ps.begin(), ps.end() - 1);
    int n = p.size(), k = 0;
    mcH = vector<point>(2 * n);
```

```
    sort(p.begin(), p.end(), cmp_point);
    for (int i = 0; i < n; i++) {
        while (k >= 2 && crs(mcH[k - 1] - mcH[k - 2], p[i] - mcH[k - 2]) <= 0)
            k--;
        mcH[k++] = p[i];
    }
    for (int i = n - 2, t = k + 1; i >= 0; i--) {
        while (k >= t && crs(mcH[k - 1] - mcH[k - 2], p[i] - mcH[k - 2]) <= 0)
            k--;
        mcH[k++] = p[i];
    }
    mcH.resize(k);
}
// O(n) - rotating calipers (works on a ccw closed convex hull)
gtype rotatingCalipers(vector<point> &ps) {
    int aI = 0, bI = 0;
    for (size_t i = 1; i < ps.size(); ++i)
        aI = (ps[i].y < ps[aI].y ? i : aI), bI = (ps[i].y > ps[bI].y ? i : bI)
            ;
    gtype minWidth = ps[bI].y - ps[aI].y, aAng, bAng;
    point aV = point(1, 0), bV = point(-1, 0);
    for (gtype ang = 0; ang < pi; ang += min(aAng, bAng)) {
        aAng = acos(dot(ps[aI + 1] - ps[aI], aV)
            / abs(aV) / abs(ps[aI + 1] - ps[aI]));
        bAng = acos(dot(ps[bI + 1] - ps[bI], bV)
            / abs(bV) / abs(ps[bI + 1] - ps[bI]));
        aV = rot(aV, min(aAng, bAng)), bV = rot(bV, min(aAng, bAng));
        if (aAng < bAng)
            minWidth = min(minWidth, pntLinDist(ps[aI], ps[aI] + aV, ps[bI]))
            , aI = (aI + 1) % (ps.size() - 1);
        else
            minWidth = min(minWidth, pntLinDist(ps[bI], ps[bI] + bV, ps[aI]))
            , bI = (bI + 1) % (ps.size() - 1);
    }
    return minWidth;
}
```

3d Convex Hull

```
int n, bf[maxn][maxn], fcnt;
point3_t pt[maxn];
struct face_t {
  int a, b, c;
  bool vis;
} fc[maxn << 5]; /* Number of Faces(Unknown) */

bool remove(int p, int b, int a) {
  int f = bf[b][a];
  face_t ff;
  if (fc[f].vis) {
    if (dblcmp(volume(pt[p], pt[fc[f].a], pt[fc[f].b], pt[fc[f].c])) >= 0) {
      return true;
    } else {
      ff.a = a, ff.b = b, ff.c = p;
      bf[ff.a][ff.b] = bf[ff.b][ff.c] = bf[ff.c][ff.a] = ++fcnt;
      ff.vis = true;
      fc[fcnt] = ff;
    }
  }
  return false;
}

void dfs(int p, int f) {
  fc[f].vis = false;
  if (remove(p, fc[f].b, fc[f].a)) dfs(p, bf[fc[f].b][fc[f].a]);
  if (remove(p, fc[f].c, fc[f].b)) dfs(p, bf[fc[f].c][fc[f].b]);
  if (remove(p, fc[f].a, fc[f].c)) dfs(p, bf[fc[f].a][fc[f].c]);
}

void hull3d() {
  for (int i = 2; i <= n; ++i) {
    if (dblcmp((pt[i] - pt[1]).length()) > 0) swap(pt[i], pt[2]);
  }
  for (int i = 3; i <= n; ++i) {
    if (dblcmp(fabs(area(pt[1], pt[2], pt[i]))) > 0) swap(pt[i], pt[3]);
  }
  for (int i = 4; i <= n; ++i) {
    if (dblcmp(fabs(volume(pt[1], pt[2], pt[3], pt[i]))) > 0) swap(pt[i], pt
        [4]);
  }
  zm(fc), fcnt = 0, zm(bf);
  for (int i = 1; i <= 4; ++i) {
    face_t f;
    f.a = i + 1, f.b = i + 2, f.c = i + 3;
    if (f.a > 4) f.a -= 4;
    if (f.b > 4) f.b -= 4;
    if (f.c > 4) f.c -= 4;
    if (dblcmp(volume(pt[i], pt[f.a], pt[f.b], pt[f.c])) > 0) swap(f.a, f.b);
    f.vis = true;
```

```
    bf[f.a][f.b] = bf[f.b][f.c] = bf[f.c][f.a] = ++fcnt;
    fc[fcnt] = f;
  }
  random_shuffle(pt + 5, pt + 1 + n);
  for (int i = 5; i <= n; ++i) {
    for (int j = 1; j <= fcnt; ++j) {
      if (!fc[j].vis) continue;
      if (dblcmp(volume(pt[i], pt[fc[j].a], pt[fc[j].b], pt[fc[j].c])) >= 0) {
        dfs(i, j);
        break;
      }
    }
  }
  for (int i = 1; i <= fcnt; ++i) if (!fc[i].vis) swap(fc[i--], fc[fcnt--]);
}
```

3 Graph

Min cost flow

```
/* Min cost max flow (Edmonds-Karp relabelling + fast heap Dijkstra)
 * Based on code by Frank Chu and Igor Naverniouk
 * (http://shygypsy.com/tools/mcmf4.cpp)
 *
 * COMPLEXITY:
 *      - Worst case: O(min(m*log(m)*flow, n*m*log(m)*fcost))
 * FIELD TESTING:
 *      - Valladolid 10594: Data Flow
 * REFERENCE:
 *      Edmonds, J., Karp, R.  "Theoretical Improvements in Algorithmic
 *          Efficieincy for Network Flow Problems".
 *      This is a slight improvement of Frank Chu's implementation.
 **/

#define Inf (LLONG_MAX/2)
#define BUBL { \
    t = q[i]; q[i] = q[j]; q[j] = t; \
    t = inq[q[i]]; inq[q[i]] = inq[q[j]]; inq[q[j]] = t; }
#define Pot(u,v) (d[u] + pi[u] - pi[v])
struct MinCostMaxFlow {
  typedef long long LL;
  int n, qs;
  vector<vector<LL> > cap, cost, fnet;
  vector<vector<int> > adj;
  vector<LL> d, pi;
  vector<int> deg, par, q, inq;

  // n = number of vertices
  MinCostMaxFlow(int n): n(n), qs(0), deg(n+1), par(n+1), d(n+1), q(n+1), inq(
      n+1), pi(n+1), cap(n+1, vector<LL>(n+1)), cost(cap), fnet(cap), adj(n+1,
       vector<int>(n+1)) {}

  // call to add a directed edge. vertices are 0-based
  // ALL COSTS MUST BE NON-NEGATIVE
  void AddEdge(int from, int to, LL cap_, LL cost_) {
    cap[from][to] = cap_; cost[from][to] = cost_;
  }

  bool dijkstra( int s, int t ) {
    fill(d.begin(), d.end(), 0x3f3f3f3f3f3f3f3fLL);
    fill(par.begin(), par.end(), -1);
    fill(inq.begin(), inq.end(), -1);
    d[s] = qs = 0;
    inq[q[qs++] = s] = 0;
    par[s] = n;
    while( qs ) {
      int u = q[0]; inq[u] = -1;
      q[0] = q[--qs];
      if( qs ) inq[q[0]] = 0;
      for( int i = 0, j = 2*i + 1, t; j < qs; i = j, j = 2*i + 1 ) {
        if( j + 1 < qs && d[q[j + 1]] < d[q[j]] ) j++;
        if( d[q[j]] >= d[q[i]] ) break;
        BUBL;
      }
      for( int k = 0, v = adj[u][k]; k < deg[u]; v = adj[u][++k] ) {
        if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u] )
          d[v] = Pot(u,v) - cost[v][par[v] = u];
        if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v] )
          d[v] = Pot(u,v) + cost[par[v] = u][v];
        if( par[v] == u ) {
          if( inq[v] < 0 ) { inq[q[qs] = v] = qs; qs++; }
          for( int i = inq[v], j = ( i - 1 )/2, t;
               d[q[i]] < d[q[j]]; i = j, j = ( i - 1 )/2 )
            BUBL;
        }
      }
    }
    for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i] += d[i];
    return par[t] >= 0;
```

```
  }

  // Returns: (flow, total cost) between source s and sink t
  // Call this once only. fnet[i][j] contains the flow from i to j. Careful,
     fnet[i][j] and fnet[j][i] could both be positive.
  pair<LL, LL> mcmf4(int s, int t) {
    for( int i = 0; i < n; i++ )
      for( int j = 0; j < n; j++ )
        if( cap[i][j] || cap[j][i] ) adj[i][deg[i]++] = j;
    LL flow = 0; LL fcost = 0;
    while( dijkstra( s, t ) ) {
      LL bot = LLONG_MAX;
      for( int v = t, u = par[v]; v != s; u = par[v = u] )
        bot = min(bot, fnet[v][u] ? fnet[v][u] : ( cap[u][v] - fnet[u][v] ));
      for( int v = t, u = par[v]; v != s; u = par[v = u] )
        if( fnet[v][u] ) { fnet[v][u] -= bot; fcost -= bot * cost[v][u]; }
        else { fnet[u][v] += bot; fcost += bot * cost[u][v]; }
      flow += bot;
    }
    return make_pair(flow, fcost);
  }
};
```

Dinic's (VE^2 max flow)

```
// C++ implementation of Dinic's Algorithm
#include<bits/stdc++.h>
using namespace std;

// A structure to represent a edge between
// two vertex
struct Edge
{
    int v ;  // Vertex v (or "to" vertex)
             // of a directed edge u-v. "From"
             // vertex u can be obtained using
             // index in adjacent array.

    int flow ; // flow of data in edge

    int C;    // capacity

    int rev ; // To store index of reverse
               // edge in adjacency list so that
               // we can quickly find it.
};

// Residual Graph
class Graph
{
    int V; // number of vertex
    int *level ; // stores level of a node
    vector< Edge > *adj;
public :
    Graph(int V)
    {
        adj = new vector<Edge>[V];
        this->V = V;
        level = new int[V];
    }

    // add edge to the graph
    void addEdge(int u, int v, int C)
    {
        // Forward edge : 0 flow and C capacity
        Edge a;
        a.v = v;
        a.flow = 0;
        a.C = C;
        a.rev = adj[v].size();

        // Back edge : 0 flow and 0 capacity
        Edge b;
        b.v = u;
        b.flow = 0;
        b.C = 0;
        b.rev = adj[u].size();

        adj[u].push_back(a);
        adj[v].push_back(b); // reverse edge
    }

    bool BFS(int s, int t);
    int sendFlow(int s, int flow, int t, int ptr[]);
    int DinicMaxflow(int s, int t);
};

// Finds if more flow can be sent from s to t.
// Also assigns levels to nodes.
```

```
bool Graph::BFS(int s, int t)
{
    for (int i = 0 ; i < V ; i++)
        level[i] = -1;

    level[s] = 0;  // Level of source vertex

    // Create a queue, enqueue source vertex
    // and mark source vertex as visited here
    // level[] array works as visited array also.
    list< int > q;
    q.push_back(s);

    vector<Edge>::iterator i ;
    while (!q.empty())
    {
        int u = q.front();
        q.pop_front();
        for (i = adj[u].begin(); i != adj[u].end(); i++)
        {
            Edge &e = *i;
            if (level[e.v] < 0  && e.flow < e.C)
            {
                // Level of current vertex is,
                // level of parent + 1
                level[e.v] = level[u] + 1;

                q.push_back(e.v);
            }
        }
    }

    // IF we can not reach to the sink we
    // return false else true
    return level[t] < 0 ? false : true ;
}

// A DFS based function to send flow after BFS has
// figured out that there is a possible flow and
// constructed levels. This function called multiple
// times for a single call of BFS.
// flow : Current flow send by parent function call
// start[] : To keep track of next edge to be explored.
//           start[i] stores  count of edges explored
//           from i.
//  u : Current vertex
//  t : Sink
int Graph::sendFlow(int u, int flow, int t, int start[])
{
    // Sink reached
    if (u == t)
        return flow;

    // Traverse all adjacent edges one -by - one.
    for (  ; start[u] < adj[u].size(); start[u]++)
    {
        // Pick next edge from adjacency list of u
        Edge &e = adj[u][start[u]];

        if (level[e.v] == level[u]+1 && e.flow < e.C)
        {
            // find minimum flow from u to t
            int curr_flow = min(flow, e.C - e.flow);

            int temp_flow = sendFlow(e.v, curr_flow, t, start);

            // flow is greater than zero
            if (temp_flow > 0)
            {
                // add flow  to current edge
                e.flow += temp_flow;

                // subtract flow from reverse edge
                // of current edge
                adj[e.v][e.rev].flow -= temp_flow;
                return temp_flow;
            }
        }
    }

    return 0;
}

// Returns maximum flow in graph
int Graph::DinicMaxflow(int s, int t)
{
    // Corner case
    if (s == t)
        return -1;

    int total = 0;  // Initialize result
```

```
    // Augment the flow while there is path
    // from source to sink
    while (BFS(s, t) == true)
    {
        // store how many edges are visited
        // from V { 0 to V }
        int *start = new int[V+1];

        // while flow is not zero in graph from S to D
        while (int flow = sendFlow(s, INT_MAX, t, start))

            // Add path flow to overall flow
            total += flow;
    }

    // return maximum flow
    return total;
}

// Driver program to test above functions
int main()
{
    Graph g(6);
    g.addEdge(0, 1, 16 );
    g.addEdge(0, 2, 13 );
    g.addEdge(1, 2, 10 );
    g.addEdge(1, 3, 12 );
    g.addEdge(2, 1, 4 );
    g.addEdge(2, 4, 14);
    g.addEdge(3, 2, 9 );
    g.addEdge(3, 5, 20 );
    g.addEdge(4, 3, 7 );
    g.addEdge(4, 5, 4);

    // next exmp
    /*g.addEdge(0, 1, 3 );
      g.addEdge(0, 2, 7 ) ;
      g.addEdge(1, 3, 9);
      g.addEdge(1, 4, 9 );
      g.addEdge(2, 1, 9 );
      g.addEdge(2, 4, 9);
      g.addEdge(2, 5, 4);
      g.addEdge(3, 5, 3);
      g.addEdge(4, 5, 7 );
      g.addEdge(0, 4, 10);

     // next exp
     g.addEdge(0, 1, 10);
     g.addEdge(0, 2, 10);
     g.addEdge(1, 3, 4 );
     g.addEdge(1, 4, 8 );
     g.addEdge(1, 2, 2 );
     g.addEdge(2, 4, 9 );
     g.addEdge(3, 5, 10 );
     g.addEdge(4, 3, 6 );
     g.addEdge(4, 5, 10 ); */

    cout << "Maximum flow " << g.DinicMaxflow(0, 5);
    return 0;
}
```

Edmond's algorithm (unweighted general matching)

```
// Unweighted general matching.
// Vertices are numbered from 1 to V.
// G is an adjlist.
// G[x][0] contains the number of neighbours of x.
// The neigbours are then stored in G[x][1] .. G[x][G[x][0]].
// Mate[x] will contain the matching node for x.
// V and E are the number of edges and vertices.
// Slow Version (2x on random graphs) of Gabow's implementation
// of Edmonds' algorithm (O(V^3)).
const int MAXV = 250;
int G[MAXV][MAXV];
int VLabel[MAXV];
int  Queue[MAXV];
int   Mate[MAXV];
int   Save[MAXV];
int   Used[MAXV];
int    Up, Down;
int           V;

void ReMatch(int x, int y)
{
  int m = Mate[x]; Mate[x] = y;
  if (Mate[m] == x)
    {
      if (VLabel[x] <= V)
        {
```

```
          Mate[m] = VLabel[x];
          ReMatch(VLabel[x], m);
        }
      else
        {
          int a = 1 + (VLabel[x] - V - 1) / V;
          int b = 1 + (VLabel[x] - V - 1) % V;
          ReMatch(a, b); ReMatch(b, a);
        }
    }
}

void Traverse(int x)
{
  for (int i = 1; i <= V; i++) Save[i] = Mate[i];
  ReMatch(x, x);
  for (int i = 1; i <= V; i++)
    {
      if (Mate[i] != Save[i]) Used[i]++;
      Mate[i] = Save[i];
    }
}

void ReLabel(int x, int y)
{
  for (int i = 1; i <= V; i++) Used[i] = 0;
  Traverse(x); Traverse(y);
  for (int i = 1; i <= V; i++)
    {
      if (Used[i] == 1 && VLabel[i] < 0)
        {
          VLabel[i] = V + x + (y - 1) * V;
          Queue[Up++] = i;
        }
    }
}

// Call this after constructing G
void Solve()
{
  for (int i = 1; i <= V; i++)
    if (Mate[i] == 0)
      {
        for (int j = 1; j <= V; j++) VLabel[j] = -1;
        VLabel[i] = 0; Down = 1; Up = 1; Queue[Up++] = i;
        while (Down != Up)
          {
            int x = Queue[Down++];
            for (int p = 1; p <= G[x][0]; p++)
              {
                int y = G[x][p];
                if (Mate[y] == 0 && i != y)
                  {
                    Mate[y] = x; ReMatch(x, y);
                    Down = Up; break;
                  }
                if (VLabel[y] >= 0)
                  {
                    ReLabel(x, y);
                    continue;
                  }
                if (VLabel[Mate[y]] < 0)
                  {
                    VLabel[Mate[y]] = x;
                    Queue[Up++] = Mate[y];
                  }
              }
          }
      }
}

// Call this after Solve(). Returns number of edges in matching (half the
    number of matched vertices)
int Size()
{
  int Count = 0;
  for (int i = 1; i <= V; i++)
    if (Mate[i] > i) Count++;
  return Count;
}
```

Link cut tree

```
struct node_t {
  node_t();
  node_t *ch[2], *p;
  int size, root;
  int dir() { return this == p->ch[1]; }
  void setc(node_t *c, int d) { ch[d] = c, c->p = this; }
```

```
  void update() { size = ch[0]->size + ch[1]->size + 1; }
} s[maxn], *nil = s;

node_t::node_t() {
  size = 1, root = true;
  ch[0] = ch[1] = p = nil;
}

void rotate(node_t *t) {
  node_t *p = t->p;
  int d = t->dir();
  if (!p->root) {
    p->p->setc(t, p->dir());
  } else {
    p->root = false, t->root = true;
    t->p = p->p; // Path Parent
  }
  p->setc(t->ch[!d], d);
  t->setc(p, !d);
  p->update(), t->update();
}

void splay(node_t *t) {
  // t->update(); // tag!
  while (!t->root) {
    // if (!t->p->root) t->p->p->update(); t->p->update(), t->update(); // !
    if (!t->p->root) rotate(t->dir() == t->p->dir() ? t->p : t);
    rotate(t);
  }
}

void access(node_t *x) { // Ask u, v: access(u), access(v, true), x = LCA
  node_t *y = nil;
  while (x != nil) {
    splay(x);
    // if (x->p == nil) at second call, x->ch[1](rev) + (x)_single + y
    x->ch[1]->root = true;
    x->ch[1] = y, y->root = false;
    x->update();
    y = x, x = x->p;
  }
}

void cut(node_t *x) {
  access(x);
  splay(x);
  x->ch[0]->root = true;
  x->ch[0]->p = nil;
  x->ch[0] = nil;
}

void link(node_t *x, node_t *y) {
  access(y);
  splay(y);
  y->p = x;
  access(y);
}

void init() { nil->size = 0; }
```

Hungarian algorithm

```
/*******************************************************************
 *******************************************************************
 **
 ** libhungarian by Cyrill Stachniss, 2004
 **
 **
 ** Solving the Minimum Assignment Problem using the
 ** Hungarian Method.
 **
 ** ** This file may be freely copied and distributed! **
 **
 ** Parts of the used code was originally provided by the
 ** "Stanford GraphGase", but I made changes to this code.
 ** As asked by  the copyright node of the "Stanford GraphGase",
 ** I hereby proclaim that this file are *NOT* part of the
 ** "Stanford GraphGase" distrubition!
 **
 ** This file is distributed in the hope that it will be useful,
 ** but WITHOUT ANY WARRANTY; without even the implied
 ** warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
 ** PURPOSE.
 **
 *******************************************************************
 *******************************************************************/


#include <stdio.h>
```

```c
#include <stdlib.h>
#include "hungarian.h"
/*
typedef struct {
  int num_rows;
  int num_cols;
  int** cost;
  int** assignment;
} hungarian_problem_t;
 */

#define INF (0x7FFFFFFF)
#define verbose (0)

#define hungarian_test_alloc(X) do {if ((void *)(X) == NULL) fprintf(stderr, "
    Out␣of␣memory␣in␣%s,␣(%s,␣line␣%d).\n", __FUNCTION__, __FILE__, __LINE__);
     } while (0)


void hungarian_print_matrix(int** C, int rows, int cols) {
  int i,j;
  fprintf(stderr , "\n");
  for(i=0; i<rows; i++) {
    fprintf(stderr, "␣[");
    for(j=0; j<cols; j++) {
      fprintf(stderr, "%5d␣",C[i][j]);
    }
    fprintf(stderr, "]\n");
  }
  fprintf(stderr, "\n");
}

void hungarian_print_assignment(hungarian_problem_t* p) {
  hungarian_print_matrix(p->assignment, p->num_rows, p->num_cols) ;
}

void hungarian_print_costmatrix(hungarian_problem_t* p) {
  hungarian_print_matrix(p->cost, p->num_rows, p->num_cols) ;
}

void hungarian_print_status(hungarian_problem_t* p) {

  fprintf(stderr,"cost:\n");
  hungarian_print_costmatrix(p);

  fprintf(stderr,"assignment:\n");
  hungarian_print_assignment(p);

}

int hungarian_imax(int a, int b) {
  return (a<b)?b:a;
}

int hungarian_init(hungarian_problem_t* p, int** cost_matrix, int rows, int
    cols, int mode) {

  int i,j, org_cols, org_rows;
  int max_cost;
  max_cost = 0;

  org_cols = cols;
  org_rows = rows;

  // is the number of cols  not equal to number of rows ?
  // if yes, expand with 0-cols / 0-cols
  rows = hungarian_imax(cols, rows);
  cols = rows;

  p->num_rows = rows;
  p->num_cols = cols;

  p->cost = (int**)calloc(rows,sizeof(int*));
  hungarian_test_alloc(p->cost);
  p->assignment = (int**)calloc(rows,sizeof(int*));
  hungarian_test_alloc(p->assignment);

  for(i=0; i<p->num_rows; i++) {
    p->cost[i] = (int*)calloc(cols,sizeof(int));
    hungarian_test_alloc(p->cost[i]);
    p->assignment[i] = (int*)calloc(cols,sizeof(int));
    hungarian_test_alloc(p->assignment[i]);
    for(j=0; j<p->num_cols; j++) {
      p->cost[i][j] =  (i < org_rows && j < org_cols) ? cost_matrix[i][j] : 0;
      p->assignment[i][j] = 0;

      if (max_cost < p->cost[i][j])
  max_cost = p->cost[i][j];
    }
  }
```

```
  if (mode == HUNGARIAN_MODE_MAXIMIZE_UTIL) {
    for(i=0; i<p->num_rows; i++) {
      for(j=0; j<p->num_cols; j++) {
  p->cost[i][j] =  max_cost - p->cost[i][j];
      }
    }
  }
  else if (mode == HUNGARIAN_MODE_MINIMIZE_COST) {
    // nothing to do
  }
  else
    fprintf(stderr,"%s: unknown mode. Mode was set to 
        HUNGARIAN_MODE_MINIMIZE_COST !\n", __FUNCTION__);

  return rows;
}




void hungarian_free(hungarian_problem_t* p) {
  int i;
  for(i=0; i<p->num_rows; i++) {
    free(p->cost[i]);
    free(p->assignment[i]);
  }
  free(p->cost);
  free(p->assignment);
  p->cost = NULL;
  p->assignment = NULL;
}



void hungarian_solve(hungarian_problem_t* p)
{
  int i, j, m, n, k, l, s, t, q, unmatched, cost;
  int* col_mate;
  int* row_mate;
  int* parent_row;
  int* unchosen_row;
  int* row_dec;
  int* col_inc;
  int* slack;
  int* slack_row;

  cost=0;
  m =p->num_rows;
  n =p->num_cols;

  col_mate = (int*)calloc(p->num_rows,sizeof(int));
  hungarian_test_alloc(col_mate);
  unchosen_row = (int*)calloc(p->num_rows,sizeof(int));
  hungarian_test_alloc(unchosen_row);
  row_dec  = (int*)calloc(p->num_rows,sizeof(int));
  hungarian_test_alloc(row_dec);
  slack_row  = (int*)calloc(p->num_rows,sizeof(int));
  hungarian_test_alloc(slack_row);

  row_mate = (int*)calloc(p->num_cols,sizeof(int));
  hungarian_test_alloc(row_mate);
  parent_row = (int*)calloc(p->num_cols,sizeof(int));
  hungarian_test_alloc(parent_row);
  col_inc = (int*)calloc(p->num_cols,sizeof(int));
  hungarian_test_alloc(col_inc);
  slack = (int*)calloc(p->num_cols,sizeof(int));
  hungarian_test_alloc(slack);

  for (i=0;i<p->num_rows;i++) {
    col_mate[i]=0;
    unchosen_row[i]=0;
    row_dec[i]=0;
    slack_row[i]=0;
  }
  for (j=0;j<p->num_cols;j++) {
    row_mate[j]=0;
    parent_row[j] = 0;
    col_inc[j]=0;
    slack[j]=0;
  }

  for (i=0;i<p->num_rows;++i)
    for (j=0;j<p->num_cols;++j)
      p->assignment[i][j]=HUNGARIAN_NOT_ASSIGNED;

  // Begin subtract column minima in order to start with lots of zeroes 12
  if (verbose)
    fprintf(stderr, "Using heuristic\n");
  for (l=0;l<n;l++)
    {
      s=p->cost[0][l];
      for (k=1;k<m;k++)
```

```
if (p->cost[k][l]<s)
  s=p->cost[k][l];
    cost+=s;
    if (s!=0)
for (k=0;k<m;k++)
  p->cost[k][l]-=s;
  }
// End subtract column minima in order to start with lots of zeroes 12

// Begin initial state 16
t=0;
for (l=0;l<n;l++)
  {
    row_mate[l]= -1;
    parent_row[l]= -1;
    col_inc[l]=0;
    slack[l]=INF;
  }
for (k=0;k<m;k++)
  {
    s=p->cost[k][0];
    for (l=1;l<n;l++)
if (p->cost[k][l]<s)
  s=p->cost[k][l];
    row_dec[k]=s;
    for (l=0;l<n;l++)
if (s==p->cost[k][l] && row_mate[l]<0)
  {
    col_mate[k]=l;
    row_mate[l]=k;
    if (verbose)
      fprintf(stderr, "matching␣col␣%d==row␣%d\n",l,k);
    goto row_done;
  }
    col_mate[k]= -1;
    if (verbose)
fprintf(stderr, "node␣%d:␣unmatched␣row␣%d\n",t,k);
    unchosen_row[t++]=k;
  row_done:
    ;
  }
// End initial state 16

// Begin Hungarian algorithm 18
if (t==0)
  goto done;
```

```
unmatched=t;
while (1)
  {
    if (verbose)
fprintf(stderr, "Matched␣%d␣rows.\n",m-t);
    q=0;
    while (1)
{
  while (q<t)
    {
      // Begin explore node q of the forest 19
      {
  k=unchosen_row[q];
  s=row_dec[k];
  for (l=0;l<n;l++)
    if (slack[l])
      {
        int del;
        del=p->cost[k][l]-s+col_inc[l];
        if (del<slack[l])
    {
      if (del==0)
        {
          if (row_mate[l]<0)
      goto breakthru;
          slack[l]=0;
          parent_row[l]=k;
          if (verbose)
      fprintf(stderr, "node␣%d:␣row␣%d==col␣%d--row␣%d\n",
             t,row_mate[l],l,k);
          unchosen_row[t++]=row_mate[l];
        }
      else
        {
          slack[l]=del;
          slack_row[l]=k;
        }
    }
      }
      }
      // End explore node q of the forest 19
      q++;
    }

  // Begin introduce a new zero into the matrix 21
  s=INF;
```

```
    for (l=0;l<n;l++)
      if (slack[l] && slack[l]<s)
        s=slack[l];
    for (q=0;q<t;q++)
      row_dec[unchosen_row[q]]+=s;
    for (l=0;l<n;l++)
      if (slack[l])
        {
    slack[l]-=s;
    if (slack[l]==0)
      {
        // Begin look at a new zero 22
        k=slack_row[l];
        if (verbose)
          fprintf(stderr,
           "Decreasing␣uncovered␣elements␣by␣%d␣produces␣zero␣at␣[%d,%d]\n",
           s,k,l);
        if (row_mate[l]<0)
          {
      for (j=l+1;j<n;j++)
        if (slack[j]==0)
          col_inc[j]+=s;
      goto breakthru;
          }
        else
          {
      parent_row[l]=k;
      if (verbose)
        fprintf(stderr, "node␣%d:␣row␣%d==col␣%d--row␣%d\n",t,row_mate[l],l,k)
            ;
      unchosen_row[t++]=row_mate[l];
          }
        // End look at a new zero 22
      }
        }
      else
        col_inc[l]+=s;
    // End introduce a new zero into the matrix 21
  }
    breakthru:
      // Begin update the matching 20
      if (verbose)
  fprintf(stderr, "Breakthrough␣at␣node␣%d␣of␣%d!\n",q,t);
      while (1)
  {
    j=col_mate[k];
    col_mate[k]=l;
    row_mate[l]=k;
    if (verbose)
      fprintf(stderr, "rematching␣col␣%d==row␣%d\n",l,k);
    if (j<0)
      break;
    k=parent_row[j];
    l=j;
  }
      // End update the matching 20
      if (--unmatched==0)
  goto done;
      // Begin get ready for another stage 17
      t=0;
      for (l=0;l<n;l++)
  {
    parent_row[l]= -1;
    slack[l]=INF;
  }
      for (k=0;k<m;k++)
  if (col_mate[k]<0)
    {
      if (verbose)
        fprintf(stderr, "node␣%d:␣unmatched␣row␣%d\n",t,k);
      unchosen_row[t++]=k;
    }
      // End get ready for another stage 17
    }
done:

  // Begin doublecheck the solution 23
  for (k=0;k<m;k++)
    for (l=0;l<n;l++)
      if (p->cost[k][l]<row_dec[k]-col_inc[l])
  exit(0);
  for (k=0;k<m;k++)
    {
      l=col_mate[k];
      if (l<0 || p->cost[k][l]!=row_dec[k]-col_inc[l])
  exit(0);
    }
  k=0;
  for (l=0;l<n;l++)
    if (col_inc[l])
      k++;
  if (k>m)
```

```
    exit(0);
  // End doublecheck the solution 23
  // End Hungarian algorithm 18

  for (i=0;i<m;++i)
    {
      p->assignment[i][col_mate[i]]=HUNGARIAN_ASSIGNED;
      /*TRACE("%d - %d\n", i, col_mate[i]);*/
    }
  for (k=0;k<m;++k)
    {
      for (l=0;l<n;++l)
  {
    /*TRACE("%d ",p->cost[k][l]-row_dec[k]+col_inc[l]);*/
    p->cost[k][l]=p->cost[k][l]-row_dec[k]+col_inc[l];
  }
      /*TRACE("\n");*/
    }
  for (i=0;i<m;i++)
    cost+=row_dec[i];
  for (i=0;i<n;i++)
    cost-=col_inc[i];
  if (verbose)
    fprintf(stderr, "Cost is %d\n",cost);


  free(slack);
  free(col_inc);
  free(parent_row);
  free(row_mate);
  free(slack_row);
  free(row_dec);
  free(unchosen_row);
  free(col_mate);
}
```

4 Strings

KMP (linear string search)

```
typedef vector<int> VI;

void buildTable(string& w, VI& t)
{
  t = VI(w.length());
  int i = 2, j = 0;
  t[0] = -1; t[1] = 0;

  while(i < w.length())
  {
    if(w[i-1] == w[j]) { t[i] = j+1; i++; j++; }
    else if(j > 0) j = t[j];
    else { t[i] = 0; i++; }
  }
}

int KMP(string& s, string& w)
{
  int m = 0, i = 0;
  VI t;

  buildTable(w, t);
  while(m+i < s.length())
  {
    if(w[i] == s[m+i])
    {
      i++;
      if(i == w.length()) return m;
    }
    else
    {
      m += i-t[i];
      if(i > 0) i = t[i];
    }
  }
  return s.length();
}
```

Manacher (max palindrome substring)

```
// Manacher's algorithm: finds maximal palindrome lengths centered around each
// position in a string (including positions between characters) and returns
// them in left-to-right order of centres. Linear time.
// Ex: "opposes" -> [0, 1, 0, 1, 4, 1, 0, 1, 0, 1, 0, 3, 0, 1, 0]
vector<int> fastLongestPalindromes(string str) {
    int i=0,j,d,s,e,lLen,palLen=0;
```

```
    vector<int> res;
    while (i < str.length()) {
        if (i > palLen && str[i-palLen-1] == str[i]) {
            palLen += 2; i++; continue;
        }
        res.push_back(palLen);
        s = res.size()-2;
        e = s-palLen;
        bool b = true;
        for (j=s; j>e; j--) {
            d = j-e-1;
            if (res[j] == d) { palLen = d; b = false; break; }
            res.push_back(min(d, res[j]));
        }
        if (b) { palLen = 1; i++; }
    }
    res.push_back(palLen);
    lLen = res.size();
    s = lLen-2;
    e = s-(2*str.length()+1-lLen);
    for (i=s; i>e; i--) { d = i-e-1; res.push_back(min(d, res[i])); }
    return res;
}
```

Suffix array

```
// Suffix array construction in O(L log^2 L) time.  Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i...L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)),
      M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector<int>(L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level
            -1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P
            [level][M[i-1].second] : i;
    }
  }

  vector<int> GetSuffixArray() { return P.back(); }

  // returns the length of the longest common prefix of s[i...L-1] and s[j...L
    -1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};
```

5 Math

Chinese Remainder Theorem

```
/* Extended Euclidean Algorithm
 * find x, y s.t. ax+by=gcd(a,b)
 */
void eea(int a, int b, int &x, int &y) {
    int r[3] = {a, b}, s[3] = {1, 0}, t[3] = {0, 1};
    while (r[1]) {
```

```
        int q = r[0] / r[1];
        r[2] = r[0] - q * r[1];
        s[2] = s[0] - q * s[1];
        t[2] = t[0] - q * t[1];;
        r[0] = r[1]; r[1] = r[2];
        s[0] = s[1]; s[1] = s[2];
        t[0] = t[1]; t[1] = t[2];
    }
    x = s[0]; y = t[0];
}

/* Chinese Remainder Theorem
 * find x s.t. x = a[i] mod b[i]
 */
int crt(int *a, int *b, int n) {
    int B = 1;
    for (int i = 0; i < n; ++i)
        B *= b[i];
    int x = 0;
    for (int i = 0; i < n; ++i) {
        int c, d;
        eea(b[i], B / b[i], c, d);
        x = (x + B / b[i] * d * a[i]) % B;
    }
    x = (x + B) % B;
    return x;
}
```

Fast Fourier Transform

```
typedef complex<double> cd;

int const NMAX = 1 << 9;
double const PI2 = atan(1.0) * 8;
cd a[NMAX], b[NMAX];

// fft(src, num, stride, dst, nth root of unity)
// e.g. fft(a, n, 1, b, polar(1.0, -PI2 / n))
void fft(cd *a, int n, int s, cd *b, cd unit) {
    if (n == 1) {
        *b = *a;
        return;
    }
    int nh = n / 2;
    fft(a    , nh, s * 2, b     , unit * unit);
    fft(a + s, nh, s * 2, b + nh, unit * unit);
    cd coef = 1;
    for (int i = 0; i < nh; ++i) {
        cd ofs = coef * b[i + nh];
        b[i + nh] = b[i] - ofs;
        b[i]      = b[i] + ofs;
        coef *= unit;
    }
}
```

Simplex Algorithm (Linear programming)

```
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;
```

```
LPSolver(const VVD &A, const VD &b, const VD &c) :
  m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
  for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
  for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];
      }
  for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
  N[n] = -1; D[m+1][n] = 1;
}

void Pivot(int r, int s) {
  DOUBLE inv = 1.0 / D[r][s];
  for (int i = 0; i < m+2; i++) if (i != r)
    for (int j = 0; j < n+2; j++) if (j != s)
      D[i][j] -= D[r][j] * D[i][s] * inv;
  for (int j = 0; j < n+2; j++) if (j != s) D[r][j] *= inv;
  for (int i = 0; i < m+2; i++) if (i != r) D[i][s] *= -inv;
  D[r][s] = inv;
  swap(B[r], N[s]);
}

bool Simplex(int phase) {
  int x = phase == 1 ? m+1 : m;
  while (true) {
    int s = -1;
    for (int j = 0; j <= n; j++) {
      if (phase == 2 && N[j] == -1) continue;
      if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s])
           s = j;
    }
    if (s < 0 || D[x][s] > -EPS) return true;
    int r = -1;
    for (int i = 0; i < m; i++) {
      if (D[i][s] < EPS) continue;
      if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
          D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
    }
    if (r == -1) return false;
    Pivot(r, s);
  }
}

DOUBLE Solve(VD &x) {
  int r = 0;
  for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
  if (D[r][n+1] <= -EPS) {
    Pivot(r, n);
    if (!Simplex(1) || D[m+1][n+1] < -EPS) return -numeric_limits<DOUBLE>::
        infinity();
    for (int i = 0; i < m; i++) if (B[i] == -1) {
      int s = -1;
      for (int j = 0; j <= n; j++)
        if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s
            ]) s = j;
      Pivot(i, s);
    }
  }
  if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
  x = VD(n);
  for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
  return D[m][n+1];
 }
};
```

Gaussian elimination

```
// Gauss-Jordan elimination with full pivoting.
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//           A MUST BE INVERTIBLE!
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
```

```
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
        if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { return 0; }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}
```

Karatsuba multiplication

```
class Karatsuba {
  typedef typename vector<T>::iterator vTi;
  int cut;

  void convolve_naive(vTi a, vTi b, vTi c, int n) {
    int n2 = n * 2;
    for (int i = 0; i < n2; ++i)
      c[i] = 0;
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        c[i + j] += a[i] * b[j];
  }
  /*
   * v----d----v                                        v-dh-v
   * --------------------------------------------------------------
   * | al * bl | ah * bh | as * bs |          |         | as | bs |
   * --------------------------------------------------------------
   * ^x0  ^xh  ^x1       ^x2
   */
  void karatsuba(vTi a, vTi b, vTi c, int n) {
    if(n <= cut) {
      convolve_naive(a, b, c, n);
      return;
    }
    int nh = n / 2;
    vTi al = a, ah = a + nh, as = c + nh * 10;
    vTi bl = b, bh = b + nh, bs = c + nh * 11;
    vTi x0 = c, x1 = c + n, x2 = c + n * 2, xh = c + nh;
    for (int i = 0; i < nh; ++i) {
      as[i] = al[i] + ah[i];
      bs[i] = bl[i] + bh[i];
    }
    karatsuba(al, bl, x0, nh);
    karatsuba(ah, bh, x1, nh);
    karatsuba(as, bs, x2, nh);
    for (int i = 0; i < n; ++i) x2[i] -= x0[i] + x1[i];
    for (int i = 0; i < n; ++i) xh[i] += x2[i];
  }
public:
  Karatsuba(int _cut = 1 << 5) : cut(_cut) {}
  vector<T> convolve(vector<T> &_a, vector<T> &_b) {
    vector<T> a = _a, b = _b, c;
    int sz = max(a.size(), b.size()), sz2;
    for (sz2 = 1; sz2 < sz; sz2 *= 2);
    a.resize(sz2); b.resize(sz2); c.resize(sz2 * 6);
    karatsuba(a.begin(), b.begin(), c.begin(), sz2);
    c.resize(_a.size() + _b.size() - 1);
    return c;
  }
};
```

Miller-Rabin (probabilistic primality testing)

```
// modulo(a,b,c) = (a^b) % c
// mulmod(a,b,c) = (a*b) % c
bool Miller(long long p,int iteration){
    if(p<2){
        return false;
    }
    if(p!=2 && p%2==0){
        return false;
    }
    long long s=p-1;
    while(s%2==0){
        s/=2;
    }
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1,temp=s;
        long long mod=modulo(a,temp,p);
        while(temp!=p-1 && mod!=1 && mod!=p-1){
            mod=mulmod(mod,mod,p);
            temp *= 2;
        }
        if(mod!=p-1 && temp%2==0){
            return false;
        }
    }
    return true;
}
```