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>());
}
```

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);
  }
};
```

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;
}
```

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;
```

```
}
```