Contents

1 Combinatorial optimization 1

1.1 Sparse max-flow . . . 1
1.2 Min-cost max-flow . . . 2
1.3 Push-relabel max-flow . . . 3
1.4 Min-cost matching . . . 4
1.5 Max bipartite matching . . . 5
1.6 Global min-cut . . . 5
1.7 Graph cut inference . . . 6

2 Geometry 7

2.1 Convex hull . . . 7
2.2 Miscellaneous geometry . . . 8
2.3 Java geometry . . . 10
2.4 3D geometry . . . 11
2.5 Slow Delaunay triangulation . . . 12

3 Numerical algorithms 12

3.1 Number theory (modular, Chinese remainder, linear Diophantine) . . . 12
3.2 Systems of linear equations, matrix inverse, determinant . . . 13
3.3 Reduced row echelon form, matrix rank . . . 14
3.4 Fast Fourier transform . . . 15
3.5 Simplex algorithm . . . 16

4 Graph algorithms 18

4.1 Fast Dijkstraś algorithm . . . 18
4.2 Strongly connected components . . . 18
4.3 Eulerian path . . . 18

5 Data structures 19

5.1 Suffix array . . . 19
5.2 Binary Indexed Tree . . . 19
5.3 Union-find set . . . 20
5.4 KD-tree . . . 20
5.5 Splay tree . . . 21
5.6 Lazy segment tree . . . 23
5.7 Lowest common ancestor . . . 23

6 Miscellaneous 24

6.1 Longest increasing subsequence . . . 24
6.2 Dates . . . 24
6.3 Regular expressions . . . 25
6.4 Prime numbers . . . 25
6.5 C++ input/output . . . 26
6.6 Knuth-Morris-Pratt . . . 26

```
          dad[e.to] = &G[x][i];
          Q[tail++] = e.to;
        }
      }
    }
    if (!dad[t]) return 0;

    long long totflow = 0;
    for (int i = 0; i < G[t].size(); i++) {
      Edge *start = &G[G[t][i].to][G[t][i].index];
      int amt = INF;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        if (!e) { amt = 0; break; }
        amt = min(amt, e->cap - e->flow);
      }
      if (amt == 0) continue;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        e->flow += amt;
        G[e->to][e->index].flow -= amt;
      }
      totflow += amt;
    }
    return totflow;
  }

  long long GetMaxFlow(int s, int t) {
    long long totflow = 0;
    while (long long flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main() {
  int n, m;
  scanf("%d%d", &n, &m);

  Dinic flow(n);
  for (int i = 0; i < m; i++) {
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    if (a == b) continue;
    flow.AddEdge(a-1, b-1, c);
    flow.AddEdge(b-1, a-1, c);
  }
  printf("%Ld\n", flow.GetMaxFlow(0, n-1));
  return 0;
}

// END CUT
```

1.2 Min-cost max-flow

```
// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972).  This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]).  For a regular max flow, set all edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
//
// INPUT:
```

```
    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};

// BEGIN CUT
// The following code solves UVA problem #10594: Data Flow

int main() {
  int N, M;

  while (scanf("%d%d", &N, &M) == 2) {
    VVL v(M, VL(3));
    for (int i = 0; i < M; i++)
      scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
    L D, K;
    scanf("%Ld%Ld", &D, &K);

    MinCostMaxFlow mcmf(N+1);
    for (int i = 0; i < M; i++) {
      mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i][2]);
      mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i][2]);
    }
    mcmf.AddEdge(0, 1, D, 0);

    pair<L, L> res = mcmf.GetMaxFlow(0, N);

    if (res.first == D) {
      printf("%Ld\n", res.second);
    } else {
      printf("Impossible.\n");
    }
  }

  return 0;
}

// END CUT
```

1.3 Push-relabel max-flow

```
// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic.  This implementation is
// significantly faster than straight Ford-Fulkerson.  It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
```

```
  void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2*N;
    for (int i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
        dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    Enqueue(v);
  }

  void Discharge(int v) {
    for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
    if (excess[v] > 0) {
      if (count[dist[v]] == 1)
        Gap(dist[v]);
      else
        Relabel(v);
    }
  }

  LL GetMaxFlow(int s, int t) {
    count[0] = N-1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
      excess[s] += G[s][i].cap;
      Push(G[s][i]);
    }

    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
      active[v] = false;
      Discharge(v);
    }

    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
  }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main() {
  int n, m;
  scanf("%d%d", &n, &m);

  PushRelabel pr(n);
  for (int i = 0; i < m; i++) {
   int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    if (a == b) continue;
    pr.AddEdge(a-1, b-1, c);
    pr.AddEdge(b-1, a-1, c);
  }
  printf("%Ld\n", pr.GetMaxFlow(0, n-1));
  return 0;
}

// END CUT
```

1.4 Min-cost matching

```
    while (Lmate[s] != -1) s++;

    // initialize Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
      dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    while (true) {

      // find closest
      j = -1;
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        if (j == -1 || dist[k] < dist[j]) j = k;
      }
      seen[j] = 1;

      // termination condition
      if (Rmate[j] == -1) break;

      // relax neighbors
      const int i = Rmate[j];
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
        if (dist[k] > new_dist) {
          dist[k] = new_dist;
          dad[k] = j;
        }
      }
    }

    // update dual variables
    for (int k = 0; k < n; k++) {
      if (k == j || !seen[k]) continue;
      const int i = Rmate[k];
      v[k] += dist[k] - dist[j];
      u[i] -= dist[k] - dist[j];
    }
    u[s] += dist[j];

    // augment along path
    while (dad[j] >= 0) {
      const int d = dad[j];
      Rmate[j] = Rmate[d];
      Lmate[Rmate[j]] = j;
      j = d;
    }
    Rmate[j] = s;
    Lmate[s] = j;

    mated++;
  }

  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

  return value;
}
```

1.5 Max bipartite matching

```
// This code performs maximum bipartite matching.
```

```
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
        if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
        for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
        for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
        used[last] = true;
        cut.push_back(last);
        if (best_weight == -1 || w[last] < best_weight) {
          best_cut = cut;
          best_weight = w[last];
        }
      } else {
        for (int j = 0; j < N; j++)
          w[j] += weights[last][j];
        added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}

// BEGIN CUT
// The following code solves UVA problem #10989: Bomb, Divide and Conquer
int main() {
  int N;
  cin >> N;
  for (int i = 0; i < N; i++) {
    int n, m;
    cin >> n >> m;
    VVI weights(n, VI(n));
    for (int j = 0; j < m; j++) {
      int a, b, c;
      cin >> a >> b >> c;
      weights[a-1][b-1] = weights[b-1][a-1] = c;
    }
    pair<int, VI> res = GetMinCut(weights);
    cout << "Case #" << i+1 << ": " << res.first << endl;
  }
}
// END CUT
```

1.7 Graph cut inference

```
// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
//        minimize         sum_i  psi_i(x[i])
//  x[1]...x[n] in {0,1}      + sum_{i < j}  phi_{ij}(x[i], x[j])
//
// where
//      psi_i : {0, 1} --> R
//   phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
//   phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)  (*)
```

```
    int c = 0;

    for (int i = 0; i < M; i++) {
      b[i] += psi[i][1] - psi[i][0];
      c += psi[i][0];
      for (int j = 0; j < i; j++)
        b[i] += phi[i][j][1][1] - phi[i][j][0][1];
      for (int j = i+1; j < M; j++) {
        cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j][0][0] - phi[i][j][1][1];
        b[i] += phi[i][j][1][0] - phi[i][j][0][0];
        c += phi[i][j][0][0];
      }
    }

#ifdef MAXIMIZATION
    for (int i = 0; i < M; i++) {
      for (int j = i+1; j < M; j++)
        cap[i][j] *= -1;
      b[i] *= -1;
    }
    c *= -1;
#endif

    for (int i = 0; i < M; i++) {
      if (b[i] >= 0) {
        cap[M][i] = b[i];
      } else {
        cap[i][M+1] = -b[i];
        c += b[i];
      }
    }

    int score = GetMaxFlow(M, M+1);
    fill(reached.begin(), reached.end(), 0);
    Augment(M, M+1, INF);
    x = VI(M);
    for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
    score += c;
#ifdef MAXIMIZATION
    score *= -1;
#endif

    return score;
  }

};

int main() {

  // solver for "Cat vs. Dog" from NWERC 2008

  int numcases;
  cin >> numcases;
  for (int caseno = 0; caseno < numcases; caseno++) {
    int c, d, v;
    cin >> c >> d >> v;

    VVVVI phi(c+d, VVVI(c+d, VVI(2, VI(2))));
    VVI psi(c+d, VI(2));
    for (int i = 0; i < v; i++) {
      char p, q;
      int u, v;
      cin >> p >> u >> q >> v;
      u--; v--;
      if (p == 'C') {
        phi[u][c+v][0][0]++;
        phi[c+v][u][0][0]++;
```

```
    while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
#endif
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the Fence (BSHEEP)

int main() {
  int t;
  scanf("%d", &t);
  for (int caseno = 0; caseno < t; caseno++) {
    int n;
    scanf("%d", &n);
    vector<PT> v(n);
    for (int i = 0; i < n; i++) scanf("%lf%lf", &v[i].x, &v[i].y);
    vector<PT> h(v);
    map<PT,int> index;
    for (int i = n-1; i >= 0; i--) index[v[i]] = i+1;
    ConvexHull(h);

    double len = 0;
    for (int i = 0; i < h.size(); i++) {
      double dx = h[i].x - h[(i+1)%h.size()].x;
      double dy = h[i].y - h[(i+1)%h.size()].y;
      len += sqrt(dx*dx+dy*dy);
    }

    if (caseno > 0) printf("\n");
    printf("%.2f\n", len);
    for (int i = 0; i < h.size(); i++) {
      if (i > 0) printf(" ");
      printf("%d", index[h[i]]);
    }
    printf("\n");
  }
}

// END CUT
```

2.2 Miscellaneous geometry

```
// C++ routines for computational geometry.

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

```
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
```

```
  return true;
}

int main() {

  // expected: (-5,2)
  cerr << RotateCCW90(PT(2,5)) << endl;

  // expected: (5,-2)
  cerr << RotateCW90(PT(2,5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 0 0 1
  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 1 1 1 0
  cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

  // expected: (1,2)
  cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

  // expected: (1,1)
  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

  vector<PT> v;
  v.push_back(PT(0,0));
  v.push_back(PT(5,0));
  v.push_back(PT(5,5));
  v.push_back(PT(0,5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2,2)) << " "
       << PointInPolygon(v, PT(2,0)) << " "
       << PointInPolygon(v, PT(0,2)) << " "
       << PointInPolygon(v, PT(5,2)) << " "
       << PointInPolygon(v, PT(2,5)) << endl;

  // expected: 0 1 1 1 1
  cerr << PointOnPolygon(v, PT(2,2)) << " "
       << PointOnPolygon(v, PT(2,0)) << " "
       << PointOnPolygon(v, PT(0,2)) << " "
       << PointOnPolygon(v, PT(5,2)) << " "
       << PointOnPolygon(v, PT(2,5)) << endl;
```

```
    return ret;
}

// make an Area object from the coordinates of a polygon
static Area makeArea(double[] pts) {
    Path2D.Double p = new Path2D.Double();
    p.moveTo(pts[0], pts[1]);
    for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i], pts[i+1]);
    p.closePath();
    return new Area(p);
}

// compute area of polygon
static double computePolygonArea(ArrayList<Point2D.Double> points) {
    Point2D.Double[] pts = points.toArray(new Point2D.Double[points.size()]);
    double area = 0;
    for (int i = 0; i < pts.length; i++){
        int j = (i+1) % pts.length;
        area += pts[i].x * pts[j].y - pts[j].x * pts[i].y;
    }
    return Math.abs(area)/2;
}

// compute the area of an Area object containing several disjoint polygons
static double computeArea(Area area) {
    double totArea = 0;
    PathIterator iter = area.getPathIterator(null);
    ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>();

    while (!iter.isDone()) {
        double[] buffer = new double[6];
        switch (iter.currentSegment(buffer)) {
        case PathIterator.SEG_MOVETO:
        case PathIterator.SEG_LINETO:
            points.add(new Point2D.Double(buffer[0], buffer[1]));
            break;
        case PathIterator.SEG_CLOSE:
            totArea += computePolygonArea(points);
            points.clear();
            break;
        }
        iter.next();
    }
    return totArea;
}

// notice that the main() throws an Exception -- necessary to
// avoid wrapping the Scanner object for file reading in a
// try { ... } catch block.
public static void main(String args[]) throws Exception {

    Scanner scanner = new Scanner(new File("input.txt"));
    // also,
    //   Scanner scanner = new Scanner (System.in);

    double[] pointsA = readPoints(scanner.nextLine());
    double[] pointsB = readPoints(scanner.nextLine());
    Area areaA = makeArea(pointsA);
    Area areaB = makeArea(pointsB);
    areaB.subtract(areaA);
    // also,
    //   areaB.exclusiveOr (areaA);
    //   areaB.add (areaA);
    //   areaB.intersect (areaA);

    // (1) determine whether B - A is a single closed shape (as
    //     opposed to multiple shapes)
```

```
  public static double ptLineDistSq(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);

    double x, y, z;
    if (pd2 == 0) {
      x = x1;
      y = y1;
      z = z1;
    } else {
      double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) / pd2;
      x = x1 + u * (x2 - x1);
      y = y1 + u * (y2 - y1);
      z = z1 + u * (z2 - z1);
      if (type != LINE && u < 0) {
        x = x1;
        y = y1;
        z = z1;
      }
      if (type == SEGMENT && u > 1.0) {
        x = x2;
        y = y2;
        z = z2;
      }
    }

    return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);
  }

  public static double ptLineDist(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type));
  }
}
```

2.5 Slow Delaunay triangulation

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   triples = a vector containing m triples of indices
//                     corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
        int n = x.size();
        vector<T> z(n);
        vector<triple> ret;
```

```
}

// computes lcm(a,b)
int lcm(int a, int b) {
  return a/gcd(a,b)*b;
}

// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}

// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[i]).  Return (z,M).  On
// failure, M = -1.  Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
```

```
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) { cerr << "Matrix is singular." << endl; exit(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;
}

int main() {
  const int n = 4;
  const int m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++) {
    a[i] = VT(A[i], A[i] + n);
    b[i] = VT(B[i], B[i] + m);
  }

  double det = GaussJordan(a, b);

  // expected: 60
  cout << "Determinant: " << det << endl;

  // expected: -0.233333 0.166667 0.133333 0.0666667
  //           0.166667 0.166667 0.333333 -0.333333
  //           0.233333 0.833333 -0.133333 -0.0666667
  //           0.05 -0.75 -0.1 0.2
  cout << "Inverse: " << endl;
```

```
  const int n = 5;
  const int m = 4;
  double A[n][m] = { {16,2,3,13},{5,11,10,8},{9,7,6,12},{4,14,15,1},{13,21,21,13} };
  VVT a(n);
  for (int i = 0; i < n; i++)
    a[i] = VT(A[i], A[i] + n);

  int rank = rref (a);

  // expected: 4
  cout << "Rank: " << rank << endl;

  // expected: 1 0 0 1
  //           0 1 0 3
  //           0 0 1 -3
  //           0 0 0 2.78206e-15
  //           0 0 0 3.22398e-15
  cout << "rref: " << endl;
  for (int i = 0; i < 5; i++){
    for (int j = 0; j < 4; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

}
```

3.4 Fast Fourier transform

```
#include <cassert>
#include <cstdio>
#include <cmath>

struct cpx
{
  cpx(){}
  cpx(double aa):a(aa){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const
  {
    return a * a + b * b;
  }
  cpx bar(void) const
  {
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b)
{
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b)
{
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b)
{
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta)
```

```
    printf("%7.2lf%7.2lf", Ai.a, Ai.b);
  }
  printf("\n");

  cpx AB[8];
  for(int i = 0 ; i < 8 ; i++)
    AB[i] = A[i] * B[i];
  cpx aconvb[8];
  FFT(AB, aconvb, 1, 8, -1);
  for(int i = 0 ; i < 8 ; i++)
    aconvb[i] = aconvb[i] / 8;
  for(int i = 0 ; i < 8 ; i++)
  {
    printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);
  }
  printf("\n");
  for(int i = 0 ; i < 8 ; i++)
  {
    cpx aconvbi(0,0);
    for(int j = 0 ; j < 8 ; j++)
    {
      aconvbi = aconvbi + a[j] * b[(8 + i - j) % 8];
    }
    printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);
  }
  printf("\n");

  return 0;
}
```

3.5 Simplex algorithm

```
// 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).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

// BEGIN CUT
#define ACM_assert(x) {if(!(x))*((long *)0)=666;}
//#define TEST_LEAD_OR_GOLD
#define TEST_HAPPINESS
// END CUT
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
```

```
    cout << "B = "; for (int i = 0; i < B.size(); i++) printf("%8d", B[i]); cout << endl;
    cout << endl;
    for (int i = 0; i < D.size(); i++) {
      for (int j = 0; j < D[i].size(); j++)
        printf("%8.2f", double(D[i][j]));
      }
      printf("\n");
    }
    printf("\n");
  }
  // END CUT
};

// BEGIN CUT
#ifdef TEST_HAPPINESS
int main() {
  int n, m;
  while (cin >> n >> m) {
    ACM_assert(3 <= n && n <= 20);
    ACM_assert(3 <= m && m <= 20);

    VVD A(m, VD(n));
    VD b(m), c(n);
    for (int i = 0; i < n; i++) {
      cin >> c[i];
      ACM_assert(c[i] >= 0);
      ACM_assert(c[i] <= 10);
    }
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++)
        cin >> A[i][j];
      cin >> b[i];
      ACM_assert(b[i] >= 0);
      ACM_assert(b[i] <= 1000);
    }

    LPSolver solver(A, b, c);
    VD sol;
    DOUBLE primal_answer = m * solver.Solve(sol);

    VVD AT(A[0].size(), VD(A.size()));
    for (int i = 0; i < A.size(); i++)
      for (int j = 0; j < A[0].size(); j++)
        AT[j][i] = -A[i][j];
    for (int i = 0; i < c.size(); i++)
      c[i] = -c[i];
    for (int i = 0; i < b.size(); i++)
      b[i] = -b[i];

    LPSolver solver2(AT, c, b);
    DOUBLE dual_answer = -m * solver2.Solve(sol);
    ACM_assert(fabs(primal_answer - dual_answer) < 1e-10);

    int primal_rounded_answer = (int) ceil(primal_answer);
    int dual_rounded_answer = (int) ceil(dual_answer);
    // The following assert fails b/c of the input data.
    // ACM_assert(primal_rounded_answer == dual_rounded_answer);

    cout << "Nasa can spend " << primal_rounded_answer << " taka." << endl;
  }
}

#else
#ifdef TEST_LEAD_OR_GOLD

int main() {
  int n;
```

4 Graph algorithms

4.1 Fast Dijkstraś algorithm

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)

#include <queue>
#include <stdio.h>

using namespace std;
const int INF = 2000000000;
typedef pair<int,int> PII;

int main(){

  int N, s, t;
  scanf ("%d%d%d", &N, &s, &t);
  vector<vector<PII> > edges(N);
  for (int i = 0; i < N; i++){
    int M;
    scanf ("%d", &M);
    for (int j = 0; j < M; j++){
      int vertex, dist;
      scanf ("%d%d", &vertex, &dist);
      edges[i].push_back (make_pair (dist, vertex)); // note order of arguments here
    }
  }

  // use priority queue in which top element has the "smallest" priority
  priority_queue<PII, vector<PII>, greater<PII> > Q;
  vector<int> dist(N, INF), dad(N, -1);
  Q.push (make_pair (0, s));
  dist[s] = 0;
  while (!Q.empty()){
    PII p = Q.top();
    if (p.second == t) break;
    Q.pop();

    int here = p.second;
    for (vector<PII>::iterator it=edges[here].begin(); it!=edges[here].end(); it++){
      if (dist[here] + it->first < dist[it->second]){
        dist[it->second] = dist[here] + it->first;
        dad[it->second] = here;
        Q.push (make_pair (dist[it->second], it->second));
      }
    }
  }

  printf ("%d\n", dist[t]);
  if (dist[t] < INF)
    for(int i=t;i!=-1;i=dad[i])
      printf ("%d%c", i, (i==s?'\n':' '));

  return 0;
}
```

4.2 Strongly connected components

```
#include<memory.h>
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
```

```
void add_edge(int a, int b)
{
        adj[a].push_front(Edge(b));
        iter ita = adj[a].begin();
        adj[b].push_front(Edge(a));
        iter itb = adj[b].begin();
        ita->reverse_edge = itb;
        itb->reverse_edge = ita;
}
```

5 Data structures

5.1 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.

#include <vector>
#include <iostream>
#include <string>

using namespace std;

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;
  }
};
```

```
#define LOGSZ 17

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

// add v to value at x
void set(int x, int v) {
  while(x <= N) {
    tree[x] += v;
    x += (x & -x);
  }
}

// get cumulative sum up to and including x
int get(int x) {
  int res = 0;
  while(x) {
    res += tree[x];
    x -= (x & -x);
  }
  return res;
}

// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
  int idx = 0, mask = N;
  while(mask && idx < N) {
    int t = idx + mask;
    if(x >= tree[t]) {
      idx = t;
      x -= tree[t];
    }
    mask >>= 1;
  }
  return idx;
}
```

5.3 Union-find set

```
//union-find set: the vector/array contains the parent of each node
int find(vector <int>& C, int x){return (C[x]==x) ? x : C[x]=find(C, C[x]);} //C++
int find(int x){return (C[x]==x)?x:C[x]=find(C[x]);} //C
```

5.4 KD-tree

```
// -----------------------------------------------------------------
// A straightforward, but probably sub-optimal KD-tree implmentation
// that's probably good enough for most things (current it's a
// 2D-tree)
//
//  - constructs from n points in O(n lg^2 n) time
//  - handles nearest-neighbor query in O(lg n) if points are well
//    distributed
//  - worst case for nearest-neighbor may be linear in pathological
//    case
//
// Sonny Chan, Stanford University, April 2009
// -----------------------------------------------------------------

#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>
```

```
        }
    }
};

// stores a single node of the kd-tree, either internal or leaf
struct kdnode
{
    bool leaf;      // true if this is a leaf node (has one point)
    point pt;       // the single point of this is a leaf
    bbox bound;     // bounding box for set of points in children

    kdnode *first, *second; // two children of this kd-node

    kdnode() : leaf(false), first(0), second(0) {}
    ~kdnode() { if (first) delete first; if (second) delete second; }

    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }

    // recursively builds a kd-tree from a given cloud of points
    void construct(vector<point> &vp)
    {
        // compute bounding box for points at this node
        bound.compute(vp);

        // if we're down to one point, then we're a leaf node
        if (vp.size() == 1) {
            leaf = true;
            pt = vp[0];
        }
        else {
            // split on x if the bbox is wider than high (not best heuristic...)
            if (bound.x1-bound.x0 >= bound.y1-bound.y0)
                sort(vp.begin(), vp.end(), on_x);
            // otherwise split on y-coordinate
            else
                sort(vp.begin(), vp.end(), on_y);

            // divide by taking half the array for each child
            // (not best performance if many duplicates in the middle)
            int half = vp.size()/2;
            vector<point> vl(vp.begin(), vp.begin()+half);
            vector<point> vr(vp.begin()+half, vp.end());
            first = new kdnode();   first->construct(vl);
            second = new kdnode();  second->construct(vr);
        }
    }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree
{
    kdnode *root;

    // constructs a kd-tree from a points (copied here, as it sorts them)
    kdtree(const vector<point> &vp) {
        vector<point> v(vp.begin(), vp.end());
        root = new kdnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }

    // recursive search method returns squared distance to nearest point
    ntype search(kdnode *node, const point &p)
    {
```

```
  Node *ch[2], *pre;
  int val, size;
  bool isTurned;
} nodePool[N_MAX], *null, *root;

Node *allocNode(int val)
{
  static int freePos = 0;
  Node *x = &nodePool[freePos ++];
  x->val = val, x->isTurned = false;
  x->ch[0] = x->ch[1] = x->pre = null;
  x->size = 1;
  return x;
}

inline void update(Node *x)
{
  x->size = x->ch[0]->size + x->ch[1]->size + 1;
}

inline void makeTurned(Node *x)
{
  if(x == null)
    return;
  swap(x->ch[0], x->ch[1]);
  x->isTurned ^= 1;
}

inline void pushDown(Node *x)
{
  if(x->isTurned)
  {
    makeTurned(x->ch[0]);
    makeTurned(x->ch[1]);
    x->isTurned ^= 1;
  }
}

inline void rotate(Node *x, int c)
{
  Node *y = x->pre;
  x->pre = y->pre;
  if(y->pre != null)
    y->pre->ch[y == y->pre->ch[1]] = x;
  y->ch[!c] = x->ch[c];
  if(x->ch[c] != null)
    x->ch[c]->pre = y;
  x->ch[c] = y, y->pre = x;
  update(y);
  if(y == root)
    root = x;
}

void splay(Node *x, Node *p)
{
  while(x->pre != p)
  {
    if(x->pre->pre == p)
      rotate(x, x == x->pre->ch[0]);
    else
    {
      Node *y = x->pre, *z = y->pre;
      if(y == z->ch[0])
      {
        if(x == y->ch[0])
          rotate(y, 1), rotate(x, 1);
        else
```

```
    for(int i = 1; i <= n; i ++)
    {
      select(i + 1, null);
      printf("%d ", root->val);
    }
  }
```

5.6 Lazy segment tree

```
public class SegmentTreeRangeUpdate {
        public long[] leaf;
        public long[] update;
        public int origSize;
        public SegmentTreeRangeUpdate(int[] list)       {
                origSize = list.length;
                leaf = new long[4*list.length];
                update = new long[4*list.length];
                build(1,0,list.length-1,list);
        }
        public void build(int curr, int begin, int end, int[] list)     {
                if(begin == end)
                        leaf[curr] = list[begin];
                else    {
                        int mid = (begin+end)/2;
                        build(2 * curr, begin, mid, list);
                        build(2 * curr + 1, mid+1, end, list);
                        leaf[curr] = leaf[2*curr] + leaf[2*curr+1];
                }
        }
        public void update(int begin, int end, int val) {
                update(1,0,origSize-1,begin,end,val);
        }
        public void update(int curr,  int tBegin, int tEnd, int begin, int end, int val)        {
                if(tBegin >= begin && tEnd <= end)
                        update[curr] += val;
                else    {
                        leaf[curr] += (Math.min(end,tEnd)-Math.max(begin,tBegin)+1) * val;
                        int mid = (tBegin+tEnd)/2;
                        if(mid >= begin && tBegin <= end)
                                update(2*curr, tBegin, mid, begin, end, val);
                        if(tEnd >= begin && mid+1 <= end)
                                update(2*curr+1, mid+1, tEnd, begin, end, val);
                }
        }
        public long query(int begin, int end)   {
                return query(1,0,origSize-1,begin,end);
        }
        public long query(int curr, int tBegin, int tEnd, int begin, int end)   {
                if(tBegin >= begin && tEnd <= end)      {
                        if(update[curr] != 0)   {
                                leaf[curr] += (tEnd-tBegin+1) * update[curr];
                                if(2*curr < update.length){
                                        update[2*curr] += update[curr];
                                        update[2*curr+1] += update[curr];
                                }
                                update[curr] = 0;
                        }
                        return leaf[curr];
                }
                else    {
                        leaf[curr] += (tEnd-tBegin+1) * update[curr];
                        if(2*curr < update.length){
                                update[2*curr] += update[curr];
                                update[2*curr+1] += update[curr];
                        }
```

```
int main(int argc,char* argv[])
{
    // read num_nodes, the total number of nodes
    log_num_nodes=lb(num_nodes);

    for(int i = 0; i < num_nodes; i++)
    {
        int p;
        // read p, the parent of node i or -1 if node i is the root

        A[i][0] = p;
        if(p != -1)
            children[p].push_back(i);
        else
            root = i;
    }

    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
        for(int i = 0; i < num_nodes; i++)
            if(A[i][j-1] != -1)
                A[i][j] = A[A[i][j-1]][j-1];
            else
                A[i][j] = -1;

    // precompute L
    DFS(root, 0);


    return 0;
}
```

6 Miscellaneous

6.1 Longest increasing subsequence

```
// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest increasing subsequence

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

using namespace std;

typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
  VPII best;
  VI dad(v.size(), -1);

  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator it = lower_bound(best.begin(), best.end(), item);
```

```
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```

6.3 Regular expressions

```
// Code which demonstrates the use of Java's regular expression libraries.
// This is a solution for
//
//   Loglan: a logical language
//   http://acm.uva.es/p/v1/134.html
//
// In this problem, we are given a regular language, whose rules can be
// inferred directly from the code.  For each sentence in the input, we must
// determine whether the sentence matches the regular expression or not.  The
// code consists of (1) building the regular expression (which is fairly
// complex) and (2) using the regex to match sentences.

import java.util.*;
import java.util.regex.*;

public class LogLan {

    public static String BuildRegex (){
        String space = " +";

        String A = "([aeiou])";
        String C = "([a-z&&[^aeiou]])";
        String MOD = "(g" + A + ")";
        String BA = "(b" + A + ")";
        String DA = "(d" + A + ")";
        String LA = "(l" + A + ")";
        String NAM = "([a-z]*" + C + ")";
        String PREDA = "(" + C + C + A + C + A + "|" + C + A + C + C + A + ")";

        String predstring = "(" + PREDA + "(" + space + PREDA + ")*)";
        String predname = "(" + LA + space + predstring + "|" + NAM + ")";
        String preds = "(" + predstring + "(" + space + A + space + predstring + ")*)";
        String predclaim = "(" + predname + space + BA + space + preds + "|" + DA + space +
            preds + ")";
        String verbpred = "(" + MOD + space + predstring + ")";
        String statement = "(" + predname + space + verbpred + space + predname + "|" +
            predname + space + verbpred + ")";
        String sentence = "(" + statement + "|" + predclaim + ")";

        return "^" + sentence + "$";
    }

    public static void main (String args[]){

        String regex = BuildRegex();
        Pattern pattern = Pattern.compile (regex);

        Scanner s = new Scanner(System.in);
        while (true) {
```

```
//    439   443   449   457   461   463   467   479   487   491   499   503
//    509   521   523   541   547   557   563   569   571   577   587   593
//    599   601   607   613   617   619   631   641   643   647   653   659
//    661   673   677   683   691   701   709   719   727   733   739   743
//    751   757   761   769   773   787   797   809   811   821   823   827
//    829   839   853   857   859   863   877   881   883   887   907   911
//    919   929   937   941   947   953   967   971   977   983   991   997

// Other primes:
//    The largest prime smaller than 10 is 7.
//    The largest prime smaller than 100 is 97.
//    The largest prime smaller than 1000 is 997.
//    The largest prime smaller than 10000 is 9973.
//    The largest prime smaller than 100000 is 99991.
//    The largest prime smaller than 1000000 is 999983.
//    The largest prime smaller than 10000000 is 9999991.
//    The largest prime smaller than 100000000 is 99999989.
//    The largest prime smaller than 1000000000 is 999999937.
//    The largest prime smaller than 10000000000 is 9999999967.
//    The largest prime smaller than 100000000000 is 99999999977.
//    The largest prime smaller than 1000000000000 is 999999999989.
//    The largest prime smaller than 10000000000000 is 9999999999971.
//    The largest prime smaller than 100000000000000 is 99999999999973.
//    The largest prime smaller than 1000000000000000 is 999999999999989.
//    The largest prime smaller than 10000000000000000 is 9999999999999937.
//    The largest prime smaller than 100000000000000000 is 99999999999999997.
//    The largest prime smaller than 1000000000000000000 is 999999999999999989.
```

6.5 C++ input/output

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed); cout << setprecision(5);
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf(ios::showpoint);
    cout << 100.0 << endl;
    cout.unsetf(ios::showpoint);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl;
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
}
```

6.6 Knuth-Morris-Pratt

```
/*
Searches for the string w in the string s (of length k). Returns the
0-based index of the first match (k if no match is found). Algorithm
runs in O(k) time.
*/
```

```cpp
#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
  double r, lat, lon;
};

struct rect
{
  double x, y, z;
};

ll convert(rect& P)
{
  ll Q;
  Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
  Q.lat = 180/M_PI*asin(P.z/Q.r);
  Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y));

  return Q;
}

rect convert(ll& Q)
{
  rect P;
  P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.z = Q.r*sin(Q.lat*M_PI/180);

  return P;
}

int main()
{
  rect A;
  ll B;

  A.x = -1.0; A.y = 2.0; A.z = -3.0;

  B = convert(A);
  cout << B.r << " " << B.lat << " " << B.lon << endl;

  A = convert(B);
  cout << A.x << " " << A.y << " " << A.z << endl;
}
```

6.8 Emacs settings

```
;; Jack's .emacs file

(global-set-key "\C-z"      'scroll-down)
(global-set-key "\C-x\C-p"  '(lambda() (interactive) (other-window -1)) )
(global-set-key "\C-x\C-o"  'other-window)
(global-set-key "\C-x\C-n"  'other-window)
(global-set-key "\M-."      'end-of-buffer)
(global-set-key "\M-,"      'beginning-of-buffer)
(global-set-key "\M-g"      'goto-line)
(global-set-key "\C-c\C-w"  'compare-windows)

(tool-bar-mode 0)
(scroll-bar-mode -1)

(global-font-lock-mode 1)
```