ACM-ICPC Team Reference Document

University of Massachusetts Boston

Contents

1 Graphs

1.1 Max Cardinality Bipartite Matching Recursive

```
typedef vector<int> vi;
vector<vi> lst; // adj list: left-nodes links
vi rima, visited; // rima = right nodes' (left-)match
int n, m; // n = # of left nodes, m = # of right nodes

bool find_match(int where) {
 if (where == -1) return 1;
 for (int i = 0; i < (int)lst[where].size(); i++) {
  int match = lst[where][i];
  if (!visited[match]) {
   visited[match] = 1;
   if (find_match(rima[match])) {
    rima[match] = where;
    return 1;
   }
  }
 }
 return 0;
}

int maximum_matching() // O(V*E)
{
 int ans = 0;

 visited.resize(m), rima.assign(m, -1);
 for (int i = 0; i < n; ++i) {
  fill(visited.begin(), visited.end(), 0);
  ans += find_match(i);
 }

 return ans;
}
```

1.2 Max Flow

```
struct MfEdge {
 int v, cap;
 int backid; // id to the back edge
};

struct MaxFlow {
 vector<vector<int>> g; // integers represent edges' ids
 vector<MfEdge> edges; // edges.size() should always be even
 int n, s, t; // n = # vertices, s = src vertex, t = sink vertex

 int find_path() {
  const int inf = int(1e9 + 7);
  vector<int> from(n, -1), used_edge(n, -1);

  vector<int> visited(n, -1); queue<int> q;
  q.push(s); visited[s] = true;
  while (!visited[t] && !q.empty()) {
   int u = q.front();
   q.pop();
   for (int eid : g[u]) {
    int v = edges[eid].v;
    if (edges[eid].cap > 0 && !visited[v]) {
     from[v] = u, used_edge[v] = eid;
     q.push(v); visited[v] = true;
     if (v == t) break;
    }
   }
  }

  int f = inf;
  if (from[t] != -1) {
   for (int v = t; from[v] > -1; v = from[v]) {
    f = min(edges[used_edge[v]].cap, f);
   }
   for (int v = t; from[v] > -1; v = from[v]) {
    int backid = edges[used_edge[v]].backid;
    edges[used_edge[v]].cap -= f;
    edges[backid].cap += f;
   }
  }
  return (f == inf ? 0 : f);
 }

 int get() {
  int mf = 0, d;
  while ((d = find_path())) mf += d;
  return mf;
 }
};
```

1.3 Disjoint Sets

```
struct UnionFind {
 vector<int> pset, set_size;
 int disjointSetsSize;

 void initSet(int N) {
  pset.assign(N, 0);
```

```
    set_size.assign(N, 1);
    disjointSetsSize = N;
    for (int i = 0; i < N; i++) pset[i] = i;
  }

  int findSet(int i) { return (pset[i] == i) ? i : (pset[i] = findSet(pset[i])); }

  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }

  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {
      set_size[findSet(j)] += set_size[findSet(i)];
      pset[findSet(i)] = findSet(j);
      disjointSetsSize--;
    }
  }

  int numDisjointsSets() { return disjointSetsSize; }

  int sizeOfSet(int i) { return set_size[findSet(i)]; }
};
```

1.4 Tree Center

```
#define MAX_SIZE 50000

vector<int> L[MAX_SIZE];

bool visited[MAX_SIZE];
int V, prev[MAX_SIZE], Q[MAX_SIZE], tail;

int most_distant(int s) {
  fill(visited, visited + V, false);
  visited[s] = true;
  Q[0] = s;
  tail = 1;
  prev[s] = -1;

  int ans = s;

  for (int k = 0; k < V; ++k) {
    int aux = Q[k];
    ans = aux;

    for (int i = L[aux].size() - 1; i >= 0; --i) {
      int v = L[aux][i];
      if (visited[v]) continue;
      visited[v] = true;
      Q[tail] = v;
      ++tail;
      prev[v] = aux;
    }
  }

  return ans;
}

void get_center() {
  int s = most_distant(0);
  int e = most_distant(s);
  int v = e, L = 0;

  while (v != -1) {
    Q[L] = v;
    v = prev[v];
    ++L;
  }

  if (L & 1)
    printf("%d\n", 1 + Q[L / 2]);
  else
    printf("%d %d\n", 1 + min(Q[L / 2 - 1], Q[L / 2]),
        1 + max(Q[L / 2 - 1], Q[L / 2]));
}
```

1.5 Bridges

```
#define SZ 100
bool M[SZ][SZ];
int N, colour[SZ], dfsNum[SZ], num, pos[SZ], leastAncestor[SZ], parent[SZ];

void dfs(int u) {
  int v;
  stack<int> S;
  S.push(u);

  while (!S.empty()) {
    v = S.top();
    if (colour[v] == 0) {
      colour[v] = 1;
      dfsNum[v] = num++;
      leastAncestor[v] = num;
    }

    for (; pos[v] < N; ++pos[v]) {
      if (M[v][pos[v]] && pos[v] != parent[v]) {
        if (colour[pos[v]] == 0) {
          parent[pos[v]] = v;
```

```
        S.push(pos[v]);
        break;
      } else
        leastAncestor[v] < ? = dfsNum[pos[v]];
    }
  }

  if (pos[v] == N) {
    colour[v] = 2;
    S.pop();

    if (v != u) leastAncestor[parent[v]] < ? = leastAncestor[v];
  }
 }
}

void Bridge_detection() {
 memset(colour, 0, sizeof(colour));
 memset(pos, 0, sizeof(pos));
 memset(parent, -1, sizeof(parent));
 num = 0;

 int ans = 0;

 for (int i = 0; i < N; i++)
  if (colour[i] == 0) dfs(i);

 for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
   if (parent[j] == i && leastAncestor[j] > dfsNum[i]) {
    printf("%d - %d\n", i, j);
    ++ans;
   }

 printf("%d bridges\n", ans);
}
```

1.6 Heavy Light Decomposition

```
#include <bits/stdc++.h>
#define pb push_back
#define sz size
#define all(X) (X).begin(), (X).end()
#define for_each(it, X) \
 for (__typeof((X).begin()) it = (X).begin(); it != (X).end(); it++)

using namespace std;

typedef long long int lld;
typedef pair<int, int> pii;

const int MaxN = 1 << 20;

int N, M, Level[MaxN], Parent[MaxN], Size[MaxN], Chain[MaxN];
vector<int> E[MaxN];

void DFS(int Curr, int Prev) {
 Parent[Curr] = Prev;
 Size[Curr] = 1;

 for_each(it, E[Curr]) if (*it != Prev) Level[*it] = Level[Curr] + 1,
                                        DFS(*it, Curr),
                                        Size[Curr] += Size[*it];
}

void HLD(int Curr, int Prev, int Color) {
 Chain[Curr] = Color;

 int idx = -1;
 for_each(it, E[Curr]) if (*it != Prev && (idx == -1 || Size[*it] > Size[idx]))
   idx = *it;

 if (idx != -1) HLD(idx, Curr, Color);
 for_each(it, E[Curr]) if (*it != Prev && *it != idx) HLD(*it, Curr, *it);
}

inline int LCA(int idx, int idy) {
 while (Chain[idx] != Chain[idy])
  if (Level[Chain[idx]] < Level[Chain[idy]])
   idy = Parent[Chain[idy]];
  else
   idx = Parent[Chain[idx]];
 return Level[idx] < Level[idy] ? idx : idy;
}

int main(void) {
 cin.sync_with_stdio(0);
 cout.sync_with_stdio(0);

 cin >> N >> M;
 for (int i = 0; i < N - 1; i++) {
  int idx, idy;
  cin >> idx >> idy;
  idx--;
  idy--;

  E[idx].pb(idy);
  E[idy].pb(idx);
 }

 DFS(0, -1);
 HLD(0, -1, 0);
```

```
  for (int i = 0; i < M; i++) {
    int idx, idy;
    cin >> idx >> idy;
    idx--;
    idy--;

    cout << LCA(idx, idy) + 1 << endl;
  }

  return 0;
}
```

1.7 Strongly Connected Components

```
vector<vector<int> > g, gt;
stack<int> S;
int n;
vi scc;

void scc_dfs(const vector<vector<int> > &g, int u, bool addToStack = false) {
  for (int i = 0; i < (int)g[u].size(); ++i) {
    int v = g[u][i];
    if (scc[v] == inf) {
      scc[v] = scc[u];
      scc_dfs(g, v, addToStack);
    }
  }
  if (addToStack) S.push(u);
}

int kosaraju() {
  const int inf = int(1e9 + 7);
  int ans = 0;

  scc.assign(n, inf);
  for (int u = 0; u < n; ++u) {
    if (scc[u] != inf) continue;
    scc[u] = true;
    scc_dfs(g, u, true);
  }
  scc.assign(n, inf);
  while (!S.empty()) {
    int u = S.top();
    S.pop();
    if (scc[u] != inf) continue;
    scc[u] = ans++;
    scc_dfs(gt, u);
  }
  return ans;
}
```

1.8 Erdos Gallai

```
// Receives a sorted degree sequence (non ascending)
bool isGraphicSequence(const vector<int> &seq) // O(n lg n)
{
  vector<int> sum;
  int n = seq.size();

  if (n == 1 && seq[0] != 0) return false;

  sum.reserve(n + 1);
  sum.push_back(0);
  for (int i = 0; i < n; ++i) sum.push_back(sum[i] + seq[i]);
  if ((sum[n] & 1) == 1) return false;

  for (long long k = 1; k <= n - 1 && seq[k - 1] >= k; ++k) {
    int j = distance(seq.begin(), upper_bound(seq.begin() + k, seq.end(), k,
                                              greater<int>())) +
        1;
    long long left = sum[k];
    long long right = k * (k - 1) + (j - k - 1) * k + (sum[n] - sum[j - 1]);

    if (left > right) return false;
  }

  return true;
}
```

1.9 Max Cardinality Bipartite Matching Iterative

```
struct MCBM {
  // n = # of left elements, m = # of right elements
  int n, m;
  // adj list for left elements
  // left elements are [0..n-1], right elements are [0..m-1]
  vector<vector<int> > lst;

  bool find_match(int s, vector<int>& lema, vector<int>& rima) {
    vector<int> from(n, -1);
    queue<int> q;
    int where, match, next;
```

```
    bool found = false;

    q.push(s), from[s] = s;
    while (!found && !q.empty()) {
      where = q.front();
      q.pop();
      for (int i = 0; i < (int)lst[where].size(); ++i) {
        match = lst[where][i];
        next = rima[match];
        if (where != next) {
          if (next == -1) {
            found = true;
            break;
          }
          if (from[next] == -1) q.push(next), from[next] = where;
        }
      }
    }

    if (found) {
      while (from[where] != where) {
        next = lema[where];
        lema[where] = match, rima[match] = where;
        where = from[where], match = next;
      }
      lema[where] = match, rima[match] = where;
    }

    return found;
  }

  int maximum_matching() // O(V*E)
  {
    int ans = 0;
    vector<int> lema(n, -1), rima(m, -1);
    for (int i = 0; i < n; ++i) {
      ans += find_match(i, lema, rima);
    }
    return ans;
  }
};
```

1.10 Max Flow With Min Cost

```
struct McMaxFlow {
  struct MfEdge { int v, cap, cpu, backid; };
  struct FlowResult { int flow, cost; };
  vector<vector<int>> g; // integers represent edges' ids
  vector<MfEdge> edges; // edges.size() should always be even
  int n, s, t; // n = # vertices, s = src vertex, t = sink vertex

  // Directed Edge u - > v with capacity 'cap' and cost per unit 'cpu'
  void add_edge(int u, int v, int cap, int cpu) {
    int eid = edges.size();
    g[u].push_back(eid);
    g[v].push_back(eid + 1);
    edges.push_back((MfEdge){v, cap, cpu, eid + 1});
    edges.push_back((MfEdge){u, 0, -cpu, eid});
  }

  FlowResult find_path() {
    const int inf = int(1e9 + 7);
    vector<int> from(n, -1), used_edge(n, -1);

    vector<int> dist(n, inf);
    queue<int> q; vector<bool> queued(n, false);
    dist[s] = 0; q.push(s); queued[s] = true;

    while (!q.empty()) {
      const int u = q.front(); q.pop();
      queued[u] = false;

      for (int eid : g[u]) {
        int v = edges[eid].v;
        int cand_dist = dist[u] + edges[eid].cpu;
        if (edges[eid].cap > 0 && cand_dist < dist[v]) {
          dist[v] = cand_dist;
          from[v] = u; used_edge[v] = eid;
          if (!queued[v]) { q.push(v); queued[v] = true; }
        }
      }
    }

    int f = 0, fcost = 0;
    if (from[t] != -1) {
      f = inf;
      for (int v = t; from[v] > -1; v = from[v]) {
        f = min(edges[used_edge[v]].cap, f);
        fcost += edges[used_edge[v]].cpu;
      }
      for (int v = t; from[v] > -1; v = from[v]) {
        int backid = edges[used_edge[v]].backid;
        edges[used_edge[v]].cap -= f;
        edges[backid].cap += f;
      }
      fcost *= f;
    }

    return (FlowResult){f, fcost};
  }

  FlowResult get() {
```

```
    FlowResult res = {0, 0};
    while (true) {
      FlowResult fr = find_path();
      if (fr.flow == 0) break;
      res.flow += fr.flow;
      res.cost += fr.cost;
    }
    return res;
  }
};
```

1.11 Articulation Points

```
#define SZ 100
bool M[SZ][SZ];
int N, colour[SZ], dfsNum[SZ], num, pos[SZ], leastAncestor[SZ], parent[SZ];

int dfs(int u) {
  int ans = 0, cont = 0, v;

  stack<int> S;
  S.push(u);

  while (!S.empty()) {
    v = S.top();
    if (colour[v] == 0) {
      colour[v] = 1;
      dfsNum[v] = num++;
      leastAncestor[v] = num;
    }

    for (; pos[v] < N; ++pos[v]) {
      if (M[v][pos[v]] && pos[v] != parent[v]) {
        if (colour[pos[v]] == 0) {
          parent[pos[v]] = v;
          S.push(pos[v]);
          if (v == u) ++cont;
          break;
        } else
          leastAncestor[v] < ? = dfsNum[pos[v]];
      }
    }

    if (pos[v] == N) {
      colour[v] = 2;
      S.pop();

      if (v != u) leastAncestor[parent[v]] < ? = leastAncestor[v];
    }
  }

  if (cont > 1) {
    ++ans;
    printf("%d\n", u);
  }

  for (int i = 0; i < N; ++i) {
    if (i == u) continue;
    for (int j = 0; j < N; j++)
      if (M[i][j] && parent[j] == i && leastAncestor[j] >= dfsNum[i]) {
        printf("%d\n", i);
        ++ans;
        break;
      }
  }

  return ans;
}

void Articulation_points() {
  memset(colour, 0, sizeof(colour));
  memset(pos, 0, sizeof(pos));
  memset(parent, -1, sizeof(parent));
  num = 0;

  int total = 0;
  for (int i = 0; i < N; ++i)
    if (colour[i] == 0) total += dfs(i);

  printf("# Articulation Points : %d\n", total);
}
```

1.12 Dfs

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
        int V; // No. of vertices

        // Pointer to an array containing
        // adjacency lists
```

```
        list<int> *adj;

        // A recursive function used by DFS
        void DFSUtil(int v, bool visited[]);
public:
        Graph(int V); // Constructor

        // function to add an edge to graph
        void addEdge(int v, int w);

        // DFS traversal of the vertices
        // reachable from v
        void DFS(int v);
};

Graph::Graph(int V)
{
        this->V = V;
        adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
        // Mark the current node as visited and
        // print it
        visited[v] = true;
        cout << v << " ";

        // Recur for all the vertices adjacent
        // to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (!visited[*i])
                        DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
                visited[i] = false;

        // Call the recursive helper function
        // to print DFS traversal
        DFSUtil(v, visited);
}

int main()
{
        // Create a graph given in the above diagram
        Graph g(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        cout << "Following is Depth First Traversal"
                        " (starting from vertex 2) \n";
        g.DFS(2);

        return 0;
}
```

2 Java

2.1 Fast Input Output Template

```
import java.io.*;
import java.math.*;
import java.util.*;
import java.lang.*;

public class Main {
  public static void main(String[] args) {
    InputReader in = new InputReader(System.in);
    OutputWriter out = new OutputWriter(System.out);
    // Do your thing
    out.close();
  }
}

class InputReader {
  private InputStream stream;
  private byte[] buf = new byte[1024];
  private int curChar;
  private int numChars;
  private SpaceCharFilter filter;

  public InputReader(InputStream stream) {
    this.stream = stream;
```

```
}

public int read() {
 if (numChars == -1) {
  throw new InputMismatchException();
 }

 if (curChar >= numChars) {
  curChar = 0;
  try {
   numChars = stream.read(buf);
  } catch (IOException e) {
   throw new InputMismatchException();
  }
  if (numChars <= 0) {
   return -1;
  }
 }

 return buf[curChar++];
}

public int readInt() {
 int c = read();
 while (isSpaceChar(c)) {
  c = read();
 }

 int sgn = 1;
 if (c == '-') {
  sgn = -1;
  c = read();
 }

 int res = 0;
 do {
  if (c < '0' || c > '9') {
   throw new InputMismatchException();
  }
  res *= 10;
  res += c - '0';
  c = read();
 } while (!isSpaceChar(c));

 return res * sgn;
}

public String readString() {
 int c = read();
 while (isSpaceChar(c)) {
  c = read();
 }
 StringBuilder res = new StringBuilder();
 do {
  res.appendCodePoint(c);
  c = read();
 } while (!isSpaceChar(c));
 return res.toString();
}

public boolean isSpaceChar(int c) {
 if (filter != null) {
  return filter.isSpaceChar(c);
 }
 return c == ' ' || c == '\n' || c == '\r' || c == '\t' || c == -1;
}

public String next() {
 return readString();
}

public interface SpaceCharFilter {
 public boolean isSpaceChar(int ch);
}
}

class OutputWriter {
 private final PrintWriter writer;

 public OutputWriter(OutputStream outputStream) {
  writer = new PrintWriter(
   new BufferedWriter(new OutputStreamWriter(outputStream))
  );
 }

 public OutputWriter(Writer writer) {
  this.writer = new PrintWriter(writer);
 }

 public void print(Object... objects) {
  for (int i = 0; i < objects.length; i++) {
   if (i != 0) {
    writer.print(' ');
   }
   writer.print(objects[i]);
  }
 }

 public void printLine(Object... objects) {
  print(objects);
  writer.println();
 }

 public void close() {
  writer.close();
```

```
  }

  public void flush() {
    writer.flush();
  }
}

class IOUtils {
  public static int[] readIntArray(InputReader in, int size) {
    int[] array = new int[size];
    for (int i = 0; i < size; i++) {
      array[i] = in.readInt();
    }
    return array;
  }
}
```

3 Geometry

3.1 Geometry

```
#define EPS 1e-8
#define PI acos(-1)
#define Vector Point

struct Point {
  double x, y;
  Point() {}
  Point(double a, double b) {
    x = a;
    y = b;
  }
  double mod2() { return x * x + y * y; }
  double mod() { return sqrt(x * x + y * y); }
  double arg() { return atan2(y, x); }
  Point ort() { return Point(-y, x); }
  Point unit() {
    double k = mod();
    return Point(x / k, y / k);
  }
};

Point operator+(const Point &a, const Point &b) {
  return Point(a.x + b.x, a.y + b.y);
}
Point operator-(const Point &a, const Point &b) {
  return Point(a.x - b.x, a.y - b.y);
}
Point operator/(const Point &a, double k) { return Point(a.x / k, a.y / k); }
Point operator*(const Point &a, double k) { return Point(a.x * k, a.y * k); }

bool operator==(const Point &a, const Point &b) {
  return abs(a.x - b.x) < EPS && abs(a.y - b.y) < EPS;
}
bool operator!=(const Point &a, const Point &b) { return !(a == b); }
bool operator<(const Point &a, const Point &b) {
  if (abs(a.x - b.x) > EPS) return a.x < b.x;
  return a.y + EPS < b.y;
}

//### FUNCIONES BASICAS
//############################################################

double dist(const Point &A, const Point &B) {
  return hypot(A.x - B.x, A.y - B.y);
}
double cross(const Vector &A, const Vector &B) { return A.x * B.y - A.y * B.x; }
double dot(const Vector &A, const Vector &B) { return A.x * B.x + A.y * B.y; }
double area(const Point &A, const Point &B, const Point &C) {
  return cross(B - A, C - A);
}

// Heron triangulo y cuadrilatero ciclico
// http://mathworld.wolfram.com/CyclicQuadrilateral.html
// http://www.spoj.pl/problems/QUADAREA/

double areaHeron(double a, double b, double c) {
  double s = (a + b + c) / 2;
  return sqrt(s * (s - a) * (s - b) * (s - c));
}

double circumradius(double a, double b, double c) {
  return a * b * c / (4 * areaHeron(a, b, c));
}

double areaHeron(double a, double b, double c, double d) {
  double s = (a + b + c + d) / 2;
  return sqrt((s - a) * (s - b) * (s - c) * (s - d));
}

double circumradius(double a, double b, double c, double d) {
  return sqrt((a * b + c * d) * (a * c + b * d) * (a * d + b * c)) /
         (4 * areaHeron(a, b, c, d));
}

//### DETERMINA SI P PERTENECE AL SEGMENTO AB
//###########################################
bool between(const Point &A, const Point &B, const Point &P) {
  return P.x + EPS >= min(A.x, B.x) && P.x <= max(A.x, B.x) + EPS &&
         P.y + EPS >= min(A.y, B.y) && P.y <= max(A.y, B.y) + EPS;
```

```
}

bool onSegment(const Point &A, const Point &B, const Point &P) {
 return abs(area(A, B, P)) < EPS && between(A, B, P);
}

//### DETERMINA SI EL SEGMENTO P1Q1 SE INTERSECTA CON EL SEGMENTO P2Q2
//#####################
// funciona para cualquiera P1, P2, P3, P4
bool intersects(const Point &P1, const Point &P2, const Point &P3,
           const Point &P4) {
 double A1 = area(P3, P4, P1);
 double A2 = area(P3, P4, P2);
 double A3 = area(P1, P2, P3);
 double A4 = area(P1, P2, P4);

 if (((A1 > 0 && A2 < 0) || (A1 < 0 && A2 > 0)) &&
    ((A3 > 0 && A4 < 0) || (A3 < 0 && A4 > 0)))
  return true;

 else if (A1 == 0 && onSegment(P3, P4, P1))
  return true;
 else if (A2 == 0 && onSegment(P3, P4, P2))
  return true;
 else if (A3 == 0 && onSegment(P1, P2, P3))
  return true;
 else if (A4 == 0 && onSegment(P1, P2, P4))
  return true;
 else
  return false;
}

//### DETERMINA SI A, B, M, N PERTENECEN A LA MISMA RECTA
//##############################
bool sameLine(Point P1, Point P2, Point P3, Point P4) {
 return area(P1, P2, P3) == 0 && area(P1, P2, P4) == 0;
}
//### SI DOS SEGMENTOS O RECTAS SON PARALELOS
//###################################################
bool isParallel(const Point &P1, const Point &P2, const Point &P3,
           const Point &P4) {
 return cross(P2 - P1, P4 - P3) == 0;
}

//### PUNTO DE INTERSECCION DE DOS RECTAS NO PARALELAS
//#################################
Point lineIntersection(const Point &A, const Point &B, const Point &C,
              const Point &D) {
 return A + (B - A) * (cross(C - A, D - C) / cross(B - A, D - C));
}

Point circumcenter(const Point &A, const Point &B, const Point &C) {
 return (A + B + (A - B).ort() * dot(C - B, A - C) / cross(A - B, A - C)) / 2;
}

//### FUNCIONES BASICAS DE POLIGONOS
//################################################
bool isConvex(const vector<Point> &P) {
 int n = P.size(), pos = 0, neg = 0;
 for (int i = 0; i < n; i++) {
  double A = area(P[i], P[(i + 1) % n], P[(i + 2) % n]);
  if (A < 0)
   neg++;
  else if (A > 0)
   pos++;
 }
 return neg == 0 || pos == 0;
}

double area(const vector<Point> &P) {
 int n = P.size();
 double A = 0;
 for (int i = 1; i <= n - 2; i++) A += area(P[0], P[i], P[i + 1]);
 return abs(A / 2);
}

bool pointInPoly(const vector<Point> &P, const Point &A) {
 int n = P.size(), cnt = 0;
 for (int i = 0; i < n; i++) {
  int inf = i, sup = (i + 1) % n;
  if (P[inf].y > P[sup].y) swap(inf, sup);
  if (P[inf].y <= A.y && A.y < P[sup].y)
   if (area(A, P[inf], P[sup]) > 0) cnt++;
 }
 return (cnt % 2) == 1;
}

//### CONVEX HULL
//####################################################################
// O(nh)
vector<Point> ConvexHull(vector<Point> S) {
 sort(all(S));

 int it = 0;
 Point primero = S[it], ultimo = primero;

 int n = S.size();

 vector<Point> convex;
 do {
  convex.push_back(S[it]);
  it = (it + 1) % n;

  for (int i = 0; i < S.size(); i++) {
   if (S[i] != ultimo && S[i] != S[it]) {
    if (area(ultimo, S[it], S[i]) < EPS) it = i;
```

```
    }
  }

  ultimo = S[it];
 } while (ultimo != primero);

 return convex;
}

// O(n log n)
vector<Point> ConvexHull(vector<Point> P) {
 sort(P.begin(), P.end());
 int n = P.size(), k = 0;
 Point H[2 * n];

 for (int i = 0; i < n; ++i) {
  while (k >= 2 && area(H[k - 2], H[k - 1], P[i]) <= 0) --k;
  H[k++] = P[i];
 }

 for (int i = n - 2, t = k; i >= 0; --i) {
  while (k > t && area(H[k - 2], H[k - 1], P[i]) <= 0) --k;
  H[k++] = P[i];
 }

 return vector<Point>(H, H + k - 1);
}

//### DETERMINA SI P ESTA EN EL INTERIOR DEL POLIGONO CONVEXO A
//########################

// O (log n)
bool isInConvex(vector<Point> &A, const Point &P) {
 int n = A.size(), lo = 1, hi = A.size() - 1;

 if (area(A[0], A[1], P) <= 0) return 0;
 if (area(A[n - 1], A[0], P) <= 0) return 0;

 while (hi - lo > 1) {
  int mid = (lo + hi) / 2;

  if (area(A[0], A[mid], P) > 0)
   lo = mid;
  else
   hi = mid;
 }

 return area(A[lo], A[hi], P) > 0;
}

// O(n)
Point norm(const Point &A, const Point &O) {
 Vector V = A - O;
 V = V * 10000000000.0 / V.mod();
 return O + V;
}

bool isInConvex(vector<Point> &A, vector<Point> &B) {
 if (!isInConvex(A, B[0]))
  return 0;
 else {
  int n = A.size(), p = 0;

  for (int i = 1; i < B.size(); i++) {
   while (!intersects(A[p], A[(p + 1) % n], norm(B[i], B[0]), B[0]))
    p = (p + 1) % n;

   if (area(A[p], A[(p + 1) % n], B[i]) <= 0) return 0;
  }

  return 1;
 }
}

//##### SMALLEST ENCLOSING CIRCLE O(n)
//##############################################
// http://www.cs.uu.nl/docs/vakken/ga/slides4b.pdf
// http://www.spoj.pl/problems/ALIENS/

pair<Point, double> enclosingCircle(vector<Point> P) {
 random_shuffle(P.begin(), P.end());

 Point O(0, 0);
 double R2 = 0;

 for (int i = 0; i < P.size(); i++) {
  if ((P[i] - O).mod2() > R2 + EPS) {
   O = P[i], R2 = 0;
   for (int j = 0; j < i; j++) {
    if ((P[j] - O).mod2() > R2 + EPS) {
     O = (P[i] + P[j]) / 2, R2 = (P[i] - P[j]).mod2() / 4;
     for (int k = 0; k < j; k++)
      if ((P[k] - O).mod2() > R2 + EPS)
       O = circumcenter(P[i], P[j], P[k]), R2 = (P[k] - O).mod2();
    }
   }
  }
 }
 return make_pair(O, sqrt(R2));
}

//##### CLOSEST PAIR OF POINTS
//#####################################################
bool XYorder(Point P1, Point P2) {
 if (P1.x != P2.x) return P1.x < P2.x;
 return P1.y < P2.y;
```

```
}
bool YXorder(Point P1, Point P2) {
 if (P1.y != P2.y) return P1.y < P2.y;
 return P1.x < P2.x;
}
double closest_recursive(vector<Point> vx, vector<Point> vy) {
 if (vx.size() == 1) return 1e20;
 if (vx.size() == 2) return dist(vx[0], vx[1]);

 Point cut = vx[vx.size() / 2];

 vector<Point> vxL, vxR;
 for (int i = 0; i < vx.size(); i++)
  if (vx[i].x < cut.x || (vx[i].x == cut.x && vx[i].y <= cut.y))
   vxL.push_back(vx[i]);
  else
   vxR.push_back(vx[i]);

 vector<Point> vyL, vyR;
 for (int i = 0; i < vy.size(); i++)
  if (vy[i].x < cut.x || (vy[i].x == cut.x && vy[i].y <= cut.y))
   vyL.push_back(vy[i]);
  else
   vyR.push_back(vy[i]);

 double dL = closest_recursive(vxL, vyL);
 double dR = closest_recursive(vxR, vyR);
 double d = min(dL, dR);

 vector<Point> b;
 for (int i = 0; i < vy.size(); i++)
  if (abs(vy[i].x - cut.x) <= d) b.push_back(vy[i]);

 for (int i = 0; i < b.size(); i++)
  for (int j = i + 1; j < b.size() && (b[j].y - b[i].y) <= d; j++)
   d = min(d, dist(b[i], b[j]));

 return d;
}
double closest(vector<Point> points) {
 vector<Point> vx = points, vy = points;
 sort(vx.begin(), vx.end(), XYorder);
 sort(vy.begin(), vy.end(), YXorder);

 for (int i = 0; i + 1 < vx.size(); i++)
  if (vx[i] == vx[i + 1]) return 0.0;

 return closest_recursive(vx, vy);
}

// INTERSECCION DE CIRCULOS
vector<Point> circleCircleIntersection(Point O1, double r1, Point O2,
                                       double r2) {
 vector<Point> X;

 double d = dist(O1, O2);

 if (d > r1 + r2 || d < max(r2, r1) - min(r2, r1))
  return X;
 else {
  double a = (r1 * r1 - r2 * r2 + d * d) / (2.0 * d);
  double b = d - a;
  double c = sqrt(abs(r1 * r1 - a * a));

  Vector V = (O2 - O1).unit();
  Point H = O1 + V * a;

  X.push_back(H + V.ort() * c);

  if (c > EPS) X.push_back(H - V.ort() * c);
 }

 return X;
}

// LINEA AB vs CIRCULO (O, r)
// 1. Mucha perdida de precision, reemplazar por resultados de formula.
// 2. Considerar line o segment

vector<Point> lineCircleIntersection(Point A, Point B, Point O, long double r) {
 vector<Point> X;

 Point H1 = O + (B - A).ort() * cross(O - A, B - A) / (B - A).mod2();
 long double d2 = cross(O - A, B - A) * cross(O - A, B - A) / (B - A).mod2();

 if (d2 <= r * r + EPS) {
  long double k = sqrt(abs(r * r - d2));

  Point P1 = H1 + (B - A) * k / (B - A).mod();
  Point P2 = H1 - (B - A) * k / (B - A).mod();

  if (between(A, B, P1)) X.push_back(P1);

  if (k > EPS && between(A, B, P2)) X.push_back(P2);
 }

 return X;
}

//### PROBLEMAS BASICOS
//#################################################################
void CircumscribedCircle() {
 int x1, y1, x2, y2, x3, y3;
 scanf("%d %d %d %d %d %d", &x1, &y1, &x2, &y2, &x3, &y3);

 Point A(x1, y1), B(x2, y2), C(x3, y3);
```

```
  Point P1 = (A + B) / 2.0;
  Point P2 = P1 + (B - A).ort();
  Point P3 = (A + C) / 2.0;
  Point P4 = P3 + (C - A).ort();

  Point CC = lineIntersection(P1, P2, P3, P4);
  double r = dist(A, CC);

  printf("(%.6lf,%.6lf,%.6lf)\n", CC.x, CC.y, r);
}

void InscribedCircle() {
  int x1, y1, x2, y2, x3, y3;
  scanf("%d %d %d %d %d %d", &x1, &y1, &x2, &y2, &x3, &y3);

  Point A(x1, y1), B(x2, y2), C(x3, y3);

  Point AX = A + (B - A).unit() + (C - A).unit();
  Point BX = B + (A - B).unit() + (C - B).unit();

  Point CC = lineIntersection(A, AX, B, BX);
  double r = abs(area(A, B, CC) / dist(A, B));

  printf("(%.6lf,%.6lf,%.6lf)\n", CC.x, CC.y, r);
}

vector<Point> TangentLineThroughPoint(Point P, Point C, long double r) {
  vector<Point> X;

  long double h2 = (C - P).mod2();
  if (h2 < r * r)
    return X;
  else {
    long double d = sqrt(h2 - r * r);

    long double m1 = (r * (P.x - C.x) + d * (P.y - C.y)) / h2;
    long double n1 = (P.y - C.y - d * m1) / r;

    long double n2 = (d * (P.x - C.x) + r * (P.y - C.y)) / h2;
    long double m2 = (P.x - C.x - d * n2) / r;

    X.push_back(C + Point(m1, n1) * r);
    if (d != 0) X.push_back(C + Point(m2, n2) * r);

    return X;
  }
}

void TangentLineThroughPoint() {
  int xc, yc, r, xp, yp;
  scanf("%d %d %d %d %d", &xc, &yc, &r, &xp, &yp);

  Point C(xc, yc), P(xp, yp);

  double hyp = dist(C, P);
  if (hyp < r)
    printf("[]\n");
  else {
    double d = sqrt(hyp * hyp - r * r);

    double m1 = (r * (P.x - C.x) + d * (P.y - C.y)) / (r * r + d * d);
    double n1 = (P.y - C.y - d * m1) / r;
    double ang1 = 180 * atan(-m1 / n1) / PI + EPS;
    if (ang1 < 0) ang1 += 180.0;

    double n2 = (d * (P.x - C.x) + r * (P.y - C.y)) / (r * r + d * d);
    double m2 = (P.x - C.x - d * n2) / r;
    double ang2 = 180 * atan(-m2 / n2) / PI + EPS;
    if (ang2 < 0) ang2 += 180.0;

    if (ang1 > ang2) swap(ang1, ang2);

    if (d == 0)
      printf("[%.6lf]\n", ang1);
    else
      printf("[%.6lf,%.6lf]\n", ang1, ang2);
  }
}

void CircleThroughAPointAndTangentToALineWithRadius() {
  int xp, yp, x1, y1, x2, y2, r;
  scanf("%d %d %d %d %d %d %d", &xp, &yp, &x1, &y1, &x2, &y2, &r);

  Point P(xp, yp), A(x1, y1), B(x2, y2);

  Vector V = (B - A).ort() * r / (B - A).mod();

  Point X[2];
  int cnt = 0;

  Point H1 = P + (B - A).ort() * cross(P - A, B - A) / (B - A).mod2() + V;
  double d1 = abs(r + cross(P - A, B - A) / (B - A).mod());

  if (d1 - EPS <= r) {
    double k = sqrt(abs(r * r - d1 * d1));

    X[cnt++] = Point(H1 + (B - A).unit() * k);

    if (k > EPS) X[cnt++] = Point(H1 - (B - A).unit() * k);
  }

  Point H2 = P + (B - A).ort() * cross(P - A, B - A) / (B - A).mod2() - V;
  double d2 = abs(r - cross(P - A, B - A) / (B - A).mod());

  if (d2 - EPS <= r) {
```

```
    double k = sqrt(abs(r * r - d2 * d2));

    X[cnt++] = Point(H2 + (B - A).unit() * k);

    if (k > EPS) X[cnt++] = Point(H2 - (B - A).unit() * k);
  }

  sort(X, X + cnt);

  if (cnt == 0)
    printf("[]\n");
  else if (cnt == 1)
    printf("[(%.6lf,%.6lf)]\n", X[0].x, X[0].y);
  else if (cnt == 2)
    printf("[(%.6lf,%.6lf),(%.6lf,%.6lf)]\n", X[0].x, X[0].y, X[1].x, X[1].y);
}

void CircleTangentToTwoLinesWithRadius() {
  int x1, y1, x2, y2, x3, y3, x4, y4, r;
  scanf("%d %d %d %d %d %d %d %d %d", &x1, &y1, &x2, &y2, &x3, &y3, &x4, &y4,
      &r);

  Point A1(x1, y1), B1(x2, y2), A2(x3, y3), B2(x4, y4);

  Vector V1 = (B1 - A1).ort() * r / (B1 - A1).mod();
  Vector V2 = (B2 - A2).ort() * r / (B2 - A2).mod();

  Point X[4];
  X[0] = lineIntersection(A1 + V1, B1 + V1, A2 + V2, B2 + V2);
  X[1] = lineIntersection(A1 + V1, B1 + V1, A2 - V2, B2 - V2);
  X[2] = lineIntersection(A1 - V1, B1 - V1, A2 + V2, B2 + V2);
  X[3] = lineIntersection(A1 - V1, B1 - V1, A2 - V2, B2 - V2);

  sort(X, X + 4);
  printf("[(%.6lf,%.6lf),(%.6lf,%.6lf),(%.6lf,%.6lf),(%.6lf,%.6lf)]\n", X[0].x,
      X[0].y, X[1].x, X[1].y, X[2].x, X[2].y, X[3].x, X[3].y);
}

void CircleTangentToTwoDisjointCirclesWithRadius() {
  int x1, y1, r1, x2, y2, r2, r;
  scanf("%d %d %d %d %d %d %d", &x1, &y1, &r1, &x2, &y2, &r2, &r);

  Point A(x1, y1), B(x2, y2);

  r1 += r;
  r2 += r;

  double d = dist(A, B);

  if (d > r1 + r2 || d < max(r1, r2) - min(r1, r2))
    printf("[]\n");
  else {
    double a = (r1 * r1 - r2 * r2 + d * d) / (2.0 * d);
    double b = d - a;
    double c = sqrt(abs(r1 * r1 - a * a));

    Vector V = (B - A).unit();
    Point H = A + V * a;

    Point P1 = H + V.ort() * c;
    Point P2 = H - V.ort() * c;

    if (P2 < P1) swap(P1, P2);

    if (P1 == P2)
      printf("[(%.6lf,%.6lf)]\n", P1.x, P1.y);
    else
      printf("[(%.6lf,%.6lf),(%.6lf,%.6lf)]\n", P1.x, P1.y, P2.x, P2.y);
  }
}
```

4 Strings

4.1 Trie

```
#include <bits/stdc++.h>
using namespace std;

template <int alphabet_size>
struct TrieNode {
  int n_words, n_prefixes;
  int child[alphabet_size] = {0};

  TrieNode() : n_words(0), n_prefixes(0) { }
};

template <int alphabet_size>
struct Trie {
  static constexpr int npos = -1;
  using TNode = TrieNode<alphabet_size>;
  vector<TNode> nodes;

  Trie() { nodes.emplace_back(); }

  /*
  ** Maps the given char to an unsigned integer
  ** inside the range [0..alphabet_size)
  */
  int char_to_child(char c) {
    int result = c - '0';
```

```
  //assert(0 <= result && result < alphabet_size);
  return result;
}

/*
** Adds the given word to the trie
*/
void add_word(const char *s) {
  int current = 0;

  for (int i = 0; s[i]; i++) {
    nodes[current].n_prefixes += 1;

    int next_child = char_to_child(s[i]);

    int next_node = nodes[current].child[next_child];
    if (next_node == 0) {
      next_node = nodes.size();
      nodes[current].child[next_child] = next_node;
      nodes.emplace_back();
    }
    current = next_node;
  }

  nodes[current].n_prefixes += 1;
  nodes[current].n_words += 1;
}

/*
** Traverses the trie, following the content of string 's'.
** Returns the node ID where the traversal stopped, or
** Trie::npos if it couldn't follow the whole string.
*/
int traverse(const char *s) {
  int current = 0;

  for (int i = 0; s[i]; i++) {
    int next_child = char_to_child(s[i]);

    int next_node = nodes[current].child[next_child];
    if (next_node == 0) {
      return Trie::npos;
    }

    current = next_node;
  }

  return current;
}

int count_prefixes(const char *s) {
  int node = traverse(s);
  int result = (node == Trie::npos ? 0 : nodes[node].n_prefixes);
    return result;
  }

  int count_words(const char *s) {
    int node = traverse(s);
    int result = (node == Trie::npos ? 0 : nodes[node].n_words);
    return result;
  }
};
```

4.2 Z Function

```
/*
** Given a string S of length n, the Z Algorithm produces
** an array Z where Z[i] is the length of the longest substring
** starting from S[i] which is also a prefix of S, i.e. the
** maximum k such that S[j] = S[i + j] for all 0 <= j < k.
** Note that Z[i] = 0 means that S[0] != S[i].
*/
void z_func(const string &s) {
  const int length = s.size();
  int left = 0, right = 0;

  vi z(length);
  z[0] = 0;

  for (int i = 1; i < length; i++) {
    if (i > right) {
      int j;
      for (j = 0; i + j < length && s[i + j] == s[j]; j++)
        ;
      z[i] = j;
      left = i;
      right = i + j - 1;
    } else if (z[i - left] < right - i + 1)
      z[i] = z[i - left];
    else {
      int j;
      for (j = 1; right + j < length && s[right + j] == s[right - i + j]; j++)
        ;
      z[i] = right - i + j;
      left = i;
      right = right + j - 1;
    }
  }

  return z;
}
```

4.3 Minimum String Rotation

```
int minimumExpression(string s) {
  s = s + s;
  int len = s.size(), i = 0, j = 1, k = 0;
  while (i + k < len && j + k < len) {
    if (s[i + k] == s[j + k])
      k++;
    else if (s[i + k] > s[j + k]) {
      i = i + k + 1;
      if (i <= j) i = j + 1;
      k = 0;
    } else if (s[i + k] < s[j + k]) {
      j = j + k + 1;
      if (j <= i) j = i + 1;
      k = 0;
    }
  }
  return min(i, j);
}
```

4.4 Knuth Morris Pratt

```
#define MAX_L 70
int f[MAX_L];

void prefixFunction(string P) {
  int n = P.size(), k = 0;
  f[0] = 0;

  for (int i = 1; i < n; ++i) {
    while (k > 0 && P[k] != P[i]) k = f[k - 1];
    if (P[k] == P[i]) ++k;
    f[i] = k;
  }
}

int KMP(string P, string T) {
  int n = P.size(), L = T.size(), k = 0, ans = 0;

  for (int i = 0; i < L; ++i) {
    while (k > 0 && P[k] != T[i]) k = f[k - 1];
    if (P[k] == T[i]) ++k;

    if (k == n) {
      ++ans;
      k = f[k - 1];
    }
  }

  return ans;
}
```

5 Data Structures

5.1 Ranged Fenwick Tree

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstring>
using namespace std;

// Implementation based on the code provided at Petr's blog. Nevertheless,
// an easy and helpful explanation can be found in TopCoder's forums
// http://petr-mitrichev.blogspot.com/2013/05/fenwick-tree-range-updates.html
// http://apps.topcoder.com/forums/?module=RevisionHistory&messageID=1407869
template <typename T>
class RangedFenwickTree {
 public:
  RangedFenwickTree() {}

  RangedFenwickTree(unsigned int n) { Init(n); }

  T Query(int at) const {
    T mul = 0, add = 0;
    int start = at;
    while (at >= 0) {
      mul += dataMul[at];
      add += dataAdd[at];
      at = (at & (at + 1)) - 1;
    }
    return mul * start + add;
  }

  T QueryInterval(int x, int y) const { return Query(y) - Query(x - 1); }

  void Update(int x, int y, T delta) {
    InternalUpdate(x, delta, -delta * (x - 1));
    if (y + 1 < (int)this->size()) InternalUpdate(y + 1, -delta, delta * y);
  }

  unsigned int size() const { return dataMul.size(); }
```

```
  void Init(unsigned int n) {
    dataMul.assign(n, 0);
    dataAdd.assign(n, 0);
  }

  vector<T> dataMul, dataAdd;

 private:
  void InternalUpdate(int x, T mul, T add) {
    for (int i = x; i < (int)this->size(); i = (i | (i + 1))) {
      dataMul[i] += mul;
      dataAdd[i] += add;
    }
  }
};

// Extension of the Ranged Fenwick Tree to 2D
template <typename T>
class RangedFenwickTree2D {
 public:
  RangedFenwickTree2D() {}

  RangedFenwickTree2D(unsigned int m, unsigned int n) { Init(m, n); }

  T Query(int x, int y) const {
    T mul = 0, add = 0;
    for (int i = x; i >= 0; i = (i & (i + 1)) - 1) {
      mul += dataMul[i].Query(y);
      add += dataAdd[i].Query(y);
    }
    return mul * x + add;
  }

  T QuerySubmatrix(int x1, int y1, int x2, int y2) const {
    T result = Query(x2, y2);
    if (x1 > 0) result -= Query(x1 - 1, y2);
    if (y1 > 0) result -= Query(x2, y1 - 1);
    if (x1 > 0 && y1 > 0) result += Query(x1 - 1, y1 - 1);
    return result;
  }

  void Update(int x1, int y1, int x2, int y2, T delta) {
    for (int i = x1; i < (int)dataMul.size(); i |= i + 1) {
      dataMul[i].Update(y1, y2, delta);
      dataAdd[i].Update(y1, y2, -delta * (x1 - 1));
    }
    for (int i = x2 + 1; i < (int)dataMul.size(); i |= i + 1) {
      dataMul[i].Update(y1, y2, -delta);
      dataAdd[i].Update(y1, y2, delta * x2);
    }
  }

  void Init(unsigned int m, unsigned int n) {
    // dataMul efficient initialization
    if (dataMul.size() == m) {
      for (int i = 0; i < (int)m; i++) dataMul[i].Init(n);
    } else {
      dataMul.assign(m, RangedFenwickTree<T>(n));
    }
    // dataAdd efficient initialization
    if (dataAdd.size() == m) {
      for (int i = 0; i < (int)m; i++) dataAdd[i].Init(n);
    } else {
      dataAdd.assign(m, RangedFenwickTree<T>(n));
    }
  }

  vector<RangedFenwickTree<T> > dataMul, dataAdd;
};

int main() {
  // EXAMPLE USAGE
  // Solution for http://www.spoj.com/problems/USUBQSUB/

  ios_base::sync_with_stdio(0);
  cin.tie(0);

  int n, m;
  cin >> n >> m;
  RangedFenwickTree2D<long long> f(n + 1, n + 1);
  while (m--) {
    int kind, x1, y1, x2, y2;
    cin >> kind >> x1 >> y1 >> x2 >> y2;
    if (kind == 1) {
      cout << f.QuerySubmatrix(x1, y1, x2, y2) << '\n';
    } else {
      int value;
      cin >> value;
      f.Update(x1, y1, x2, y2, value);
    }
  }

  return 0;
}
```

5.2 Fenwick Tree

```
// Most of the implementation comes from e-maxx.ru, although several
// things can also be found on the TopCoder tutorial about BITs
// community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees
// e-maxx.ru/algo/fenwick_tree
template <typename T>
```

```
class FenwickTree {
 public:
  FenwickTree() {}

  FenwickTree(unsigned int n) { Init(n); }

  T Query(int x) const {
    T result = 0;
    for (int i = x; i >= 0; i = (i & (i + 1)) - 1) result += data[i];
    return result;
  }

  T QueryInterval(int x, int y) const { return Query(y) - Query(x - 1); }

  T QuerySingle(int x) const {
    T result = data[x];
    if (x > 0) {
      int y = (x & (x + 1)) - 1;
      x -= 1;
      while (x != y) {
        result -= data[x];
        x = (x & (x + 1)) - 1;
      }
    }
    return result;
  }

  void Update(int x, T delta) {
    for (int i = x; i < (int)data.size(); i = (i | (i + 1))) data[i] += delta;
  }

  unsigned int size() const { return data.size(); }

  void Init(unsigned int n) { data.assign(n, 0); }

  vector<T> data;
};

// Extension of the Fenwick Tree to 2D
template <typename T>
class FenwickTree2D {
 public:
  FenwickTree2D() {}

  FenwickTree2D(unsigned int m, unsigned int n) { Init(m, n); }

  T Query(int x, int y) const {
    T result = 0;
    for (int i = x; i >= 0; i = (i & (i + 1)) - 1) result += data[i].Query(y);
    return result;
  }

  void Update(int x, int y, T delta) {
    for (int i = x; i < (int)data.size(); i = (i | (i + 1)))
      data[i].Update(y, delta);
  }

  void Init(unsigned int m, unsigned int n) {
    if (data.size() == m) {
      for (int i = 0; i < (int)m; i++) data[i].Init(n);
    } else {
      data.assign(m, FenwickTree<T>(n));
    }
  }

  vector<FenwickTree<T> > data;
};

/*
** BIT Linear Construction Snippet
**

class Fenwick{
  int *m, N;
public:
  Fenwick(int a[], int n);
};

Fenwick::Fenwick(int a[], int n){
  N = n;
  m = new int[N];
  memset(m, 0, sizeof(int)*N);
  for(int i=0;i<N;++i){
    m[i] += a[i];
    if((i|(i+1))<N) m[i|(i+1)] += m[i];
  }
}
*/
```

5.3 Longest Common Ancestor

```
#define MAX_N 100000
#define LOG2_MAXN 16

// NOTA : memset(parent,-1,sizeof(parent));
int N, parent[MAX_N], L[MAX_N];
int P[MAX_N][LOG2_MAXN + 1];

int get_level(int u) {
  if (L[u] != -1)
    return L[u];
  else if (parent[u] == -1)
```

```
    return 0;
  return 1 + get_level(parent[u]);
}

void init() {
  memset(L, -1, sizeof(L));
  for (int i = 0; i < N; ++i) L[i] = get_level(i);

  memset(P, -1, sizeof(P));

  for (int i = 0; i < N; ++i) P[i][0] = parent[i];

  for (int j = 1; (1 << j) < N; ++j)
    for (int i = 0; i < N; ++i)
      if (P[i][j - 1] != -1) P[i][j] = P[P[i][j - 1]][j - 1];
}

int LCA(int p, int q) {
  if (L[p] < L[q]) swap(p, q);

  int log = 1;
  while ((1 << log) <= L[p]) ++log;
  --log;

  for (int i = log; i >= 0; --i)
    if (L[p] - (1 << i) >= L[q]) p = P[p][i];

  if (p == q) return p;

  for (int i = log; i >= 0; --i) {
    if (P[p][i] != -1 && P[p][i] != P[q][i]) {
      p = P[p][i];
      q = P[q][i];
    }
  }

  return parent[p];
}
```

5.4 Segment Tree

```
#include <bits/stdc++.h>
using namespace std;

/*
** Generic segment tree with lazy propagation (requires C++11)
** Sample node implementation that supports
** Query: sum of the elements in range [a, b)
** Update: add a given value X to every element in range [a, b)
*/

struct StNode {
  using NodeType = StNode;
  using i64 = long long;
  i64 val; // Sum of the interval
  i64 lazy; // Sumation pending to apply to children

  // Used, while creating the tree, to update the Node content according to
  // the value given by the ValueProvider
  void set(const NodeType& from) {
    val = from.val;
    lazy = identity().lazy;
  }

  // Updates the Node content to store the result of the 'merge' operation
  // applied on the children.
  // The tree will always call push_lazy() on the Node *before* calling merge()
  void merge(const NodeType& le, const NodeType& ri) {
    val = le.val + ri.val;
    lazy = identity().lazy;
  }

  // Used to update the Node content in a tree update command
  void update(const NodeType& from) {
    auto new_value = from.val;
    val += (e - s) * new_value;
    lazy += new_value;
  }

  // Pushes any pending lazy updates to children
  void push_lazy(NodeType& le, NodeType& ri) {
    if (lazy == identity().lazy) {
      return;
    }

    le.lazy += lazy;
    le.val += (le.e - le.s) * lazy;

    ri.lazy += lazy;
    ri.val += (ri.e - ri.s) * lazy;

    lazy = identity().lazy;
  }

  // This function should return a NodeType instance such that calling
  // Y.merge(X, identity()) or Y.merge(identity(), X) for any Node X with no
  // pending updates should make Y match X exactly.
  static NodeType identity() {
    static auto tmp = (NodeType){0, 0};
    return tmp;
  }
```

```
  // Internal tree data
  int son[2]; // Children of this node
  int s, e; // Interval [s, e), covered by this node
};

template <class Node>
struct SegmentTree {
  using ValueProvider = function<Node(int)>;
  vector<Node> T;

  SegmentTree(int n, const ValueProvider& vp = [](int pos) {
    return Node::identity();
  }) {
    Node nd;
    nd.son[0] = nd.son[1] = -1;
    nd.s = 0, nd.e = n;

    T.reserve(4 * n);
    T.emplace_back(std::move(nd));

    init(vp, 0);
  }

  void init(const ValueProvider& vp, int u) {
    Node& n = T[u];

    if (n.e - n.s == 1) {
      n.set(vp(n.s));
      return;
    }

    Node le(n), ri(n);

    le.e = (n.s + n.e) / 2;
    n.son[0] = T.size();
    T.emplace_back(std::move(le));
    init(vp, n.son[0]);

    ri.s = le.e;
    n.son[1] = T.size();
    T.emplace_back(std::move(ri));
    init(vp, n.son[1]);

    n.merge(T[n.son[0]], T[n.son[1]]);
  }

  void update(int le, int ri, const Node& val, int u = 0) {
    Node& n = T[u];
    if (le >= n.e || n.s >= ri) return;

    if (n.s == le && n.e == ri) {
      n.update(val);
      return;
    }

    n.push_lazy(T[n.son[0]], T[n.son[1]]);

    update(le, min(T[n.son[0]].e, ri), val, n.son[0]);
    update(max(T[n.son[1]].s, le), ri, val, n.son[1]);
    n.merge(T[n.son[0]], T[n.son[1]]);
  }

  Node query(int le, int ri, int u = 0) {
    Node& n = T[u];
    if (n.e <= le || n.s >= ri) return Node::identity();
    if (n.s == le && n.e == ri) return n;

    n.push_lazy(T[n.son[0]], T[n.son[1]]);

    Node r1, r2, r3;
    r1 = query(le, min(T[n.son[0]].e, ri), n.son[0]);
    r2 = query(max(T[n.son[1]].s, le), ri, n.son[1]);
    r3.merge(r1, r2);
    return r3;
  }
};

/*
** USAGE SAMPLE
** http://www.spoj.com/problems/HORRIBLE/
*/
int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  int tc;
  cin >> tc;

  for (int cas = 1; cas <= tc; ++cas) {
    int n, c;
    cin >> n >> c;

    SegmentTree<StNode> st(n);

    for (int i = 0; i < c; ++i) {
      int k, p, q;
      cin >> k >> p >> q;
      p -= 1;

      if (k == 0) {
        int v;
        cin >> v;
        st.update(p, q, (StNode){v});
      } else {
        auto sum = st.query(p, q).val;
        cout << sum << '\n';
```

```
      }
    }
  }

  return 0;
}
```

6 Math

6.1 Combinations

```
long long comb(int n, int m) {
  if (m > n - m) m = n - m;

  long long C = 1;
  // C^{n}_{i} -> C^{n}_{i+1}
  for (int i = 0; i < m; ++i) C = C * (n - i) / (1 + i);
  return C;
}

// Cuando n y m son grandes y se pide comb(n, m) % MOD,
// donde MOD es un numero primo, se puede usar el Teorema de Lucas.

#define MOD 3571
int C[MOD][MOD];

void FillLucasTable() {
  memset(C, 0, sizeof(C));

  for (int i = 0; i < MOD; ++i) C[i][0] = 1;
  for (int i = 1; i < MOD; ++i) C[i][i] = 1;
  for (int i = 2; i < MOD; ++i)
    for (int j = 1; j < i; ++j) C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % MOD;
}

int comb(int n, int k) {
  long long ans = 1;

  while (n != 0) {
    int ni = n % MOD, ki = k % MOD;
    n /= MOD;
    k /= MOD;
    ans = (ans * C[ni][ki]) % MOD;
  }

  return (int)ans;
}
```

6.2 Chinese Remainer Theorem

```
// rem y mod tienen el mismo numero de elementos
long long chinese_remainder(vector<int> rem, vector<int> mod) {
  long long ans = rem[0], m = mod[0];
  int n = rem.size();

  for (int i = 1; i < n; ++i) {
    int a = modular_inverse(m, mod[i]);
    int b = modular_inverse(mod[i], m);
    ans = (ans * b * mod[i] + rem[i] * a * m) % (m * mod[i]);
    m *= mod[i];
  }

  return ans;
}
```

6.3 Deterministic Miller Rabin

```
/*
** Deterministic Miller-Rabin
** if n < 3,825,123,056,546,413,051, it is enough to test
** a = 2, 3, 5, 7, 11, 13, 17, 19, and 23.
*/
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long ll;

vector<ll> mr_values({2, 3, 5, 7, 11, 13, 17, 19, 23});

ll mulmod(ll a, ll b, ll n) {
  ll erg = 0;
  ll r = 0;
  while (b > 0) {
    // unsigned long long gives enough room for base 10 operations
    ll x = ((a % n) * (b % 10)) % n;
    for (ll i = 0; i < r; i++) x = (x * 10) % n;
    erg = (erg + x) % n;
    r++;
    b /= 10;
  }
  return erg;
}

ll fastexp(ll a, ll b, ll n) {
  if (b == 0) return 1;
```

```
  if (b == 1) return a % n;
  ll res = 1;
  while (b > 0) {
    if (b % 2 == 1) res = mulmod(a, res, n);
    a = mulmod(a, a, n);
    b /= 2;
  }
  return res;
}

bool mrtest(ll n) {
  if (n == 1) return false;
  ll d = n - 1;
  ll s = 0;
  while (d % 2 == 0) {
    s++;
    d /= 2;
  }
  for (ll j = 0; j < (ll)mr_values.size(); j++) {
    if (mr_values[j] > n - 1) continue;
    ll ad = fastexp(mr_values[j], d, n);
    if (ad % n == 1) continue;
    bool notcomp = false;
    for (ll r = 0; r <= max(0ull, s - 1); r++) {
      ll rr = fastexp(2, r, n);
      ll ard = fastexp(ad, rr, n);
      if (ard % n == n - 1) {
        notcomp = true;
        break;
      }
    }
    if (!notcomp) {
      return false;
    }
  }
  return true;
}

bool isprime(ll n) {
  if (n <= 1) return false;
  if (n == 2) return true;
  if (n % 2 == 0) return false;
  return mrtest(n);
}
```

6.4 Big Integer

```
string trim_zeros(const string& a) {
  size_t idx = 0;
  while (a[idx] == '0' && idx < a.size()) idx++;
  if (idx == a.size()) idx--;

  return a.substr(idx);
}

string big_sub(const string& n1, const string& n2) {
  string a = trim_zeros(n1);
  string b = trim_zeros(n2);

  bool minus = false;
  if (esMayor(b, a)) {
    swap(a, b);
    minus = true;
  }

  int i, j, d = (a.length() - b.length());
  for (i = b.length() - 1; i >= 0; i--) {
    if (a[i + d] >= b[i])
      a[i + d] -= b[i] - '0';
    else {
      j = -1;
      while (a[i + d + j] == '0') {
        a[i + d + j] = '9';
        j--;
      }
      a[i + d + j]--;

      a[i + d] += 10 - b[i] + '0';
    }
  }

  return (minus ? "-" : "") + trim_zeros(a);
}

string big_add(const string& a, const string& b) {
  int LA = a.size(), LB = b.size(), L = max(LA, LB), carry = 0;

  string x = string(L, '0');

  while (L--) {
    LA--;
    LB--;

    if (LA >= 0) carry += a[LA] - '0';
    if (LB >= 0) carry += b[LB] - '0';

    if (carry < 10)
      x[L] = '0' + carry, carry = 0;
    else
      x[L] = '0' + carry - 10, carry = 1;
  }
```

```
  if (carry) x = '1' + x;
  return x;
}

string big_mult(string a, string b) {
  if (a == "0" || b == "0")
    return "0";
  else if (a.size() == 1) {
    int m = a[0] - '0';

    string ans = string(b.size(), '0');

    int lleva = 0;

    for (int i = b.size() - 1; i >= 0; i--) {
      int d = (b[i] - '0') * m + lleva;
      lleva = d / 10;
      ans[i] += d % 10;
    }
    if (lleva) ans = (char)(lleva + '0') + ans;
    return ans;
  } else if (b.size() == 1)
    return big_mult(b, a);
  else {
    string ans = "0";
    string ceros = "";
    for (int i = a.size() - 1; i >= 0; i--) {
      string s = big_mult(string(1, a[i]), b) + ceros;
      ceros += "0";
      ans = big_add(ans, s);
    }
    return ans;
  }
}
```

6.5 Factorial Prime Factors

```
vector<int> primes; // Filled with prime numbers <= n (at least)

void factorial_prime_factor(const int n, vector<int>& v) {
  v.clear();
  for (size_t i = 0; primes[i] <= n && i < primes.size(); i++) {
    const int& p = primes[i];
    double q = (n / (double)p);
    int d = int(q);

    while (q >= p) {
      q /= p;
      d += int(q);
    }

    v.push_back(d);
  }
}
```

6.6 Extended Gcd

```
// a*x + b*y = gcd(a,b)
int extGcd(int a, int b, int &x, int &y) {
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }

  int g = extGcd(b, a % b, y, x);
  y -= a / b * x;
  return g;
}

// ASSUME: gcd(a, m) == 1
int modInv(int a, int m) {
  int x, y;
  extGcd(a, m, x, y);
  return (x % m + m) % m;
}
```

6.7 Least Significant Bit Position

```
// http://supertech.csail.mit.edu/papers/debruijn.pdf
int lsbpos(unsigned int v) {
  static const int t[32] = {0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20,
                            15, 25, 17, 4, 8, 31, 27, 13, 23, 21, 19,
                            16, 7, 26, 12, 18, 6, 11, 5, 10, 9};
  return t[((unsigned int)((v & -v) * 0x077CB531U)) >> 27];
}
```

7 General

7.1 Longest Increasing Subsequence

```
#include <vector>
/* Finds longest strictly increasing subsequence. O(n log k) algorithm. */
void find_lis(vector<int> &a, vector<int> &b) {
 vector<int> p(a.size());
 int u, v;

 if (a.empty()) return;

 b.push_back(0);
 for (size_t i = 1; i < a.size(); i++) {
  // If next element a[i] is greater than last element of current longest
  // subsequence a[b.back()], just push it at back of "b" and continue
  if (a[b.back()] < a[i]) {
   p[i] = b.back();
   b.push_back(i);
   continue;
  }
  // Binary search to find the smallest element referenced by b which is just
  // bigger than a[i]
  // Note : Binary search is performed on b (and not a). Size of b is always
  // <= k and hence contributes O(log k) to complexity.
  for (u = 0, v = b.size() - 1; u < v;) {
   int c = (u + v) / 2;
   if (a[b[c]] < a[i])
    u = c + 1;
   else
    v = c;
  }
  // Update b if new value is smaller then previously referenced value
  if (a[i] < a[b[u]]) {
   if (u > 0) p[i] = b[u - 1];
   b[u] = i;
  }
 }
 for (u = b.size(), v = b.back(); u--; v = p[v]) b[u] = v;
}
```

7.2 Inversion Counting

```
int v[MAX], sortedV[MAX];

// Merge sort with inversion counting
long long mergeSort(int *V, int lo, int hi) {
 if (lo >= hi) {
  return 0;
 } else {
  int m1 = (lo + hi) / 2, m2 = m1 + 1;
  long long r = 0, rA, rB;
  int i = lo, j = m2, k = 0;

  rA = mergeSort(V, lo, m1);
  rB = mergeSort(V, m2, hi);

  while (i <= m1 && j <= hi) {
   if (V[j] < V[i]) {
    r += (m1 - i + 1);
    sortedV[k++] = V[j++];
   } else {
    sortedV[k++] = V[i++];
   }
  }

  if (i > m1) {
   i = j;
   j = hi;
  } else {
   j = m1;
  }

  while (i <= j) {
   sortedV[k++] = V[i++];
  }

  memcpy(V + lo, sortedV, (hi - lo + 1) * sizeof(int));

  return r + rA + rB;
 }
}
```