Default cpp

```cpp
#include <iostream>
#include <climits>
#include <cmath>
#include <cstring>
#include <string>
#include <algorithm>
#include <vector>
#include <stack>
#include <queue>
#include <list>
#include <map>
using namespace std;

void run() {

}

int main() {
    int n;
    cin >> n;
    while(n--) run();
    return 0;
}
```

~/.vimrc

```
:r $VIMRUNTIME/vimrc_example.vim
set tabstop=4
set shiftwidth=4
set softtabstop=4
set noexpandtab
set nu
map <F7> :w <ENTER> :!./compile.sh %:r <ENTER>
map <F5> <F7> <ENTER> :!./%:r <ENTER>
map <F4> <F7> <ENTER> :!./dosample.sh %:r <ENTER>
```

~/dosample.sh

```
#/bin/bash
./$1 < ~/samples/$1.in > $1.myout
echo “OUTPUT:”
cat $1.myout
echo “DIFF:”
diff ~/samples/$1.out $1.myout
```

~/compile.sh

```
#/bin/bash
g++ -Wall -O2 -g -static -o $1 $1.cpp
```

```
---------------------------------
alias dosample='./dosample.sh'
alias compile='./compile.sh'
chmod +x dosample.sh compile.sh
```

Covering problems

Minimum edge cover <-> Maximum independent set

Minimum Vert Cover
A set vertices (cover) such that each edge in the graph is incident to at least one vertex of the set.
Matching
A set of edges without common vertices (Maximum is the largest such set, maximal is a set which you cannot add more edges to without breaking the property)
Minimum Edge Cover
A set of edges (cover) such that every vertex is incident to at least one edge of the set.
Maximum Independent Set
A set of vertices in a graph such that no two of them are adjacent.
König's theorem
In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover

```
INT_MAX    = 2 147 483 647
INT_MIN    = -INT_MAX – 1
UINT_MAX   = 4 294 967 295U
LONG_MAX   = 9 223 372 036 854 775 807LL
ULLONG_MAX =18 446 744 073 709 551 615ULL
Two random large prime numbers 982451653 , 81253449
```

Augmenting Path (Bipartite Matching)

```
bool visited[512];     //visited[rightnode]
int parent[512];       //parent[rightnode] = leftnode
vector<int> adj[512]; //adj[leftnode][i] = rightnode

bool match(int node)
{
    for(int i = 0; i < (int)adj[node].size(); ++i)
    {
        if( visited[adj[node][i]] ) continue;
        visited[adj[node][i]] = true;
        if(parent[adj[node][i]] == -1 ||
            match(parent[adj[node][i]]))
        {
            parent[adj[node][i]] = node;
            return true;
        }
    }
    return false;
}

int matches = 0;
for(int i = 0; i < leftNodes; ++i)
{
    memset(visited, false, sizeof(visited));
    if(match(i)) ++matches;
}
```

Binary Search

```
int min = 0;
int max = 100; // not included
while(max - min > 1) {
    int c = (min + max) / 2;
    if(succes(c)) {
        min = c;
    } else {
        max = c;
    }
}
cout << min << endl;
```

Greatest Common Divisor

```
int gcd(int a, int b)
{
        if( b == 0 ) return a;
    if( a < 0 ) a = -a;
    if( b < 0 ) b = -b;
        int r;
        while(b)
        {
                int r = a % b;
                a = b;
                b = r;
        }
        return a;
}
```

Trie

```
struct trie {
    trie** child;
    bool word;

    string* str;

    trie(string* str) {
        child=new trie*[26];
        for(int i=0; i<26; i++)
            child[i]=NULL;
        word=false;
        this->str=str;
    }

    void add(string& str, int pos=0) {
        if(pos == str.length()) {
            word=true;
            return;
        }
        int index=str[pos]-'a';

        string *tmp;
        if(this->str == NULL) tmp=new string();
        else tmp=new string(*this->str);
        (*tmp)+=str[pos];

        if(child[index] == NULL)
            child[index]=new trie(tmp);
        child[index]->add(str,pos+1);
    }

    bool isWord(string& str, int pos=0) {
        if(pos == str.length())
            return word;
        int index=str[pos]-'a';
        if(child[index] == NULL) return false;
        return child[index]->isWord(str,pos+1);
    }
}
```

Floyd-Warshall

```
int n = 100; //aantal edges
int d[n][n];

//initialize heel d op INT_MAX / 3
//initialize d[a][b] als er een edge is tussen a,b op de edge-waarde
for(int i = 0; i < n; ++i)
    for(int j = 0; j < n; ++j)
        for(int k = 0; k < n; ++k)
            d[j][k] = min( d[j][k] , d[j][i] + d[i][k] );
```

Strongly Connected Components

```
//Graph
vector< vector<int> > adj;
//Algorithm internals
vector<int> index;
vector<int> lowlink; //lowest index reachable
vector<bool> inStack; //true iff in tarjanStack
stack<int> tarjanStack;
int newId;
//Output
vector< vector<int> > strongComponents; //collection of vertex sets

void tarjan(int v)
{
    index[v] = newId;
    lowlink[v] = newId;
    ++newId;
    tarjanStack.push(v);
    inStack[v] = true;

    for(int i = 0; i < (int)adj[v].size(); ++i)
    {
        int w = adj[v][i];
        if( index[w] == 0 )
        {
            tarjan(w);
            if( lowlink[w] < lowlink[v] ) lowlink[v] = lowlink[w];
        }
        else if( inStack[w] )
        {
            if( index[w] < lowlink[v] ) lowlink[v] = index[w];
        }
    }

    if( lowlink[v] == index[v] )
    {
        strongComponents.push_back(vector<int>());
        while(true)
        {
            int w = tarjanStack.top();
            strongComponents.back().push_back(w);
            inStack[w] = false;
            tarjanStack.pop();
            if(w == v) break;
        }
    }
}
void findSCC()
{
    //Init
    newId = 1;
    index.clear();     index.resize(n+1, 0);
    lowlink.clear(); lowlink.resize(n+1, 0);
    inStack.clear(); inStack.resize(n+1, false);
    while(!tarjanStack.empty()) tarjanStack.pop();
    strongComponents.clear();
    //Start
    for(int i = 0; i < nodecount; ++i)
        if(index[i] == 0)
            tarjan(i);
    cout << strongComponents.size() << endl;
}
```

Longest Common Subsequence

```
//IS NOT longest common subSTRING
//substring is consecutive characters, subsequence is not
//Taken from wikipedia untested

int table[1024][1024];

int LCSLength(const string& word1, const string& word2)
{
    for(int i = 0; i <= (int)word1.size(); ++i) table[i,0] = 0;
    for(int j = 0; j <= (int)word2.size(); ++j) table[0,j] = 0;

    for(int i = 1; i < (int)word1.size(); ++i) {
        for(int j = 1; j < (int)word2.size(); ++j) {
            if( word1[i-1] == word2[j-1] ) table[i][j] = table[i-1][j-1] + 1;
            else table[i][j] = max( table[i-1][j] , table[i][j-1] );
        }
    }
    return table[word1.size()][word2.size()];
}

//Get the actual LCS by backtracking through the table
string word1;
string word2;

string getLCS(int i, int j)
{
    if( i == 0 || j == 0 ) return "";
    if( word1[i-1] == word2[j-1] ) return getLCS(i-1, j-1) + word1[i-1];
    if( table[i][j-1] > table[i-1][j] ) return getLCS(i, j-1);
    else return getLCS(i-1, j);
}
```

Levenshtein (more general)

```
int costs[1002][1002];

int levDistance(const string& word1, const string& word2)
{
    for(unsigned int i = 0; i <= word1.length(); ++i)
        costs[i][0] = i; //removal_cost * i
    for(unsigned int j = 0; j <= word2.length(); ++j)
        costs[0][j] = j; //insertion_cost * j

    int a, b, c;
    for(unsigned int i = 1; i <= word1.length(); ++i)
       for(unsigned int j = 1; j <= word2.length(); ++j)
       {
          a = costs[i-1][j]+1; //removal cost of word1[i-1]
          b = costs[i][j-1]+1; //insertion cost of word2[j-1]
          c = costs[i-1][j-1] + (word1[i-1]!=word2[j-1] ? 1 : 0); //replacement cost
          costs[i][j] = min(min(a, b), c);
       }

    return costs[word1.length()][word2.length()];
}
```

Dijkstra

```cpp
typedef pair<int, int> pii;

struct Edge
{
  Edge(int _to, int _w)
  {
    to = _to;
    w = _w;
  }

  int to;
  int w;
};

// vector<Vertex> vert;
vector<vector<Edge> > adj(N_MAX);
vector<int> distances(N_MAX);
vector<bool> visited(N_MAX, false);

void dijkstra()
{
  fill(visited.begin(), visited.end(), false);

  priority_queue<pii, vector<pii>, greater<pii> > q; // dist, id
  q.push(make_pair(0, 0));
  pii v;

  while(!q.empty())
  {
    v = q.top();
    q.pop();

    if(visited[v.second])
      continue;

    visited[v.second] = true;
    for(int i = 0; i < adj[v.second].size(); ++i)
      q.push(make_pair(v.first + adj[v.second][i].w, adj[v.second][i].to));

    distances[v.second] = v.first;
  }

  // OUTPUT
  for(int i = 0; i < N_MAX; ++i)
    cout << "Distance to " << i << " is " << distances[i] << endl;
}
```

Cycle Detection

```
// assumes bidirected graph, adjust accordingly
vector<vector<int> > adj;
vector<bool> visited(N_MAX, false);
vector<int> parent(N_MAX, 0);

void cycle_detection() {
  stack<int> s;
  s.push(0);
  int current;
  while(!s.empty())
  {
    current = s.top(); s.pop();

    for(int i = 0; i < adj[current].size(); ++i)
    {
      if(visited[i])
        if(parent[current] != i)
          cout << "cycle!!!" << endl;

      s.push(adj[current][i]);
      parent[i] = current;
      visited[i] = true;
    }
  }
}
```

BFS

```
vector<vector<Edge> > adj(N_MAX);
vector<int> parent(N_MAX, -1);
vector<int> distances(N_MAX);
vector<bool> visited(N_MAX, false);
//Distance:distances[target], Path:Volg parent[target] tot -1
void bfs(int root, int target) {
    fill(distances.begin(), distances.end(), INT_MAX/2);//Init
    fill(visited.begin(), visited.end(), false);
    distances[root]=0;
    parent[root]=-1;
    visited[root]=true;

    queue<int> q;//DFS: stack<int> q;
    q.push(root);
    while(!q.empty()) {
        int curr=q.front();//DFS: int curr=q.top();
        q.pop();
        int depth=distances[curr];
        for(int i=0; i<adj[curr].size(); i++) {
            int neigh=adj[curr][i].to;
            if(!visited[neigh]) {
                visited[neigh]=true;
                parent[neigh]=curr;
                distances[neigh]=depth+1;
                if(neigh == target)    return;
                q.push(neigh);
            }
        }
    }
}
```

Bellman Ford

```
struct Edge
{
    Edge(int _from, int _to, int _w){ from = _from; to = _to; w = _w; }
    int from, to, w;
};

vector<Edge> edges;
vector<int> distances(N_MAX);

void bellman_ford()
{
    fill(distances.begin(), distances.end(), 10000);
    distances[0] = 0;

    bool updated = true;
    while(updated){
        updated = false;
        for(int i = 0; i < edges.size(); ++i)
        {
            if( distances[edges[i].to] > distances[edges[i].from] + edges[i].w){
                distances[edges[i].to] = distances[edges[i].from] + edges[i].w;
                updated = true;
            }
            //if bidirectional:
            if( distances[edges[i].from] > distances[edges[i].to] + edges[i].w){
                distances[edges[i].from] = distances[edges[i].to] + edges[i].w;
                updated = true;
            }
        }
    }
}
```

Modular Exponentiation

```
//Calculate a^b % c in log[b] time
int powmod(int a, int b, int c)
{
        int res = 1;
        long long k = a;
        while(b)
        {
                if( b & 1 ) res = (k*res)%c;
                k = (k*k)%c;
                b >>= 1;
        }
        return res;
}
```

Max Flow

```
struct Edge
{
  Edge(int _a, int _b, int _c, int _f) {
    a = _a; b = _b; c = _c; f = _f;
  }
  ~Edge() { };
  int a;
  int b;
  int c;
  int f;
  Edge* r;
};

vector< vector<Edge*> > adj;
bool* visited;
int node_count;

bool DFS(int from, int to, vector<Edge*>& path)
{
    if(from == to) return true;
    visited[from] = true;
    for(int i = 0; i < adj[from].size(); ++i)
    {
        Edge* e = adj[from][i];
        if(visited[e->b]) continue;
        if(e->f >= e->c) continue;
        visited[e->b] = true;
        path.push_back(e);
        if( DFS(e->b, to, path) ) return true;
        path.pop_back();
    }
    return false;
}

bool find_path(int from, int to, vector<Edge*>& output)
{
    output.clear();
    memset(visited, false, node_count * sizeof(bool));
    return DFS(from, to, output);
}

int max_flow(int source, int sink)
{
    vector<Edge*> p;
    while(find_path(source, sink, p))
    {
        int flow = INT_MAX;
        for(int i = 0; i < p.size(); ++i)
            if(p[i]->c - p[i]->f < flow) flow = p[i]->c - p[i]->f;

        for(int i = 0; i < p.size(); ++i) {
            p[i]->f += flow;
            p[i]->r->f -= flow;
        }
    }
  int total_flow = 0;
  for(int i = 0; i < adj[source].size(); ++i)
  {
    total_flow += adj[source][i]->f;
```

```
  }
  return total_flow;
}

void add_edge(int a, int b, int c)
{
  Edge* e = new Edge(a, b, c, 0);
  Edge* re = new Edge(b, a, 0, 0);
  e->r = re;
  re->r = e;
  adj[a].push_back(e);
  adj[b].push_back(re);
}

void run()
{
  node_count = 6;
  adj.clear();
  adj.resize(node_count);

  add_edge(0, 5, 3);
  add_edge(0, 1, 3);
  add_edge(1, 5, 2);
  add_edge(1, 2, 3);
  add_edge(5, 4, 2);
  add_edge(2, 4, 4);
  add_edge(2, 3, 2);
  add_edge(4, 3, 3);

  visited = new bool[node_count];

  int m = max_flow(0, node_count - 1);

  cout << m << endl;

  for(unsigned int i = 0; i < adj.size(); ++i)
      for(unsigned int j = 0; j < adj[i].size(); ++j)
          delete adj[i][j];
  adj.clear();
  delete[] visited;
}
```

Min Cost Max Flow

```
struct Edge
{
    Edge(int _a, int _b, int _c, int _f, int _w) {
        a = _a; b = _b; c = _c; f = _f; w = _w;
    }
    ~Edge() { };
    int a; //from
    int b; //to
    int c; //capacity
    int f; //flow
    int w; //weight
    Edge* r;
};

const int MAX_NODES = 2000;
const int MAX_DIST = 2000000; //do not choose INT_MAX because you are adding weights
vector<Edge*> adj[MAX_NODES];
int distances[MAX_NODES];
Edge* parents[MAX_NODES];

int node_count;

bool find_path(int from, int to, vector<Edge*>& output)
{
    fill(distances, distances+nodecount, MAX_DIST);
    fill(parents, parents+nodecount, (Edge*)0);
    distances[from] = 0;

    bool updated = true;
    while(updated)
    {
        updated = false;
        for(int j = 0; j < nodecount; ++j)
            for(int k = 0; k < (int)adj[j].size(); ++k){
                Edge* e = adj[j][k];
                if( e->f >= e->c ) continue;
                if( distances[e->b] > distances[e->a] + e->w )
                {
                    distances[e->b] = distances[e->a] + e->w;
                    parents[e->b] = e;
                    updated = true;
                }
            }
    }
    output.clear();
    if(distances[to] == MAX_DIST) return false;
    int cur = to;
    while(parents[cur])
    {
        output.push_back(parents[cur]);
        cur = parents[cur]->a;
    }
    return true;
}
```

```
int min_cost_max_flow(int source, int sink)
{
    int total_cost = 0;
    vector<Edge*> p;
    while(find_path(source, sink, p))
    {
        int flow = INT_MAX;
        for(int i = 0; i < p.size(); ++i)
            if(p[i]->c - p[i]->f < flow) flow = p[i]->c - p[i]->f;

        int cost = 0;
        for(int i = 0; i < p.size(); ++i) {
            cost += p[i]->w;
            p[i]->f += flow;
            p[i]->r->f -= flow;
        }
        cost *= flow; //cost per flow
        total_cost += cost;
    }
    return total_cost;
}

void add_edge(int a, int b, int c, int w)
{
    Edge* e = new Edge(a, b, c, 0, w);
    Edge* re = new Edge(b, a, 0, 0, -w);
    e->r = re;
    re->r = e;
    adj[a].push_back(e);
    adj[b].push_back(re);
}

void run()
{
    //node_count
    //add_edge
    cout << min_cost_max_flow(source, sink) << endl;
    for(int i = 0; i < nodecount; ++i){
        for(unsigned int j = 0; j < adj[i].size(); ++j)
            delete adj[i][j];
        adj[i].clear();
    }
}
```

Kruskal

```
// GAAT ER VANUIT DAT JE EEN CONNECTED GRAPH HEBT
// anders toepassingen op elke component
struct Edge
{
  Edge(int _from, int _to, int _w) {
    from = _from;
    to = _to;
    w = _w;
  }

  bool operator <(const Edge& b) const {
    return w < b.w;
  }

  bool operator >(const Edge& b) const {
    return b.w < w;
  }

  int from;
  int to;
  int w;
};

vector<Edge> edges;
vector<int> group(N_MAX);
vector<vector<int> > groups(N_MAX);

void kruskal()
{
  for(int i = 0; i < N_MAX; ++i) {
    groups[i].push_back(i);
    group[i] = i;
  }

  vector<Edge> mst;
  int total_length = 0;

  priority_queue<Edge, vector<Edge>, greater<Edge> > q;
  for(int i = 0; i < edges.size(); ++i)
    q.push(edges[i]);

  const Edge* e;
  while(!q.empty()) {
    e = &q.top();

    if(group[e->from] != group[e->to])
    {
      int g = group[e->from];
      int size = groups[g].size();
      for(int i = 0; i < size; ++i)
      {
        group[groups[g][i]] = group[e->to];
        groups[group[e->to]].push_back(groups[g][i]);
      }
      groups[g].empty();

      mst.push_back(*e);
      total_length += e->w;

      cout << groups[e->to].size() << endl;

   // we're done if every vertex is in one single group
      if(groups[e->to].size() == N_MAX)
        break;
    }

    q.pop();
  }

  // OUTPUT
  cout << "Length: " << total_length << endl;
  for(int i = 0; i < mst.size(); ++i)
    cout << "Edge: " << mst[i].from << " " << mst[i].to << endl;
}
```

KMP String search

```
int KMPsearch(const string& word, const string& text)
{
        vector<int> table(word.size()+1, 0);
        //NOTE: If you search for the SAME word in different texts
        //then only fill this table ONCE
        unsigned int i = 1;
        unsigned int j = 0;
        while(i < word.size())
        {
                if( word[i] == word[j] )
                {
                        ++i;
                        ++j;
                        table[i] = j;
                }
                else if( j > 0 )
                {
                        j = table[j];
                }
                else
                {
                        ++i;
                }
        }

        int matchcount = 0;
        i = 0;
        j = 0;
        while(i < text.size())
        {
                if( text[i] == word[j] )
                {
                        ++i;
                        ++j;
                        if( j == word.size() )
                        {
                                ++matchcount;
//Match is at text[i-j] till text[i-1] both inclusive
//cout << "Match " << matchcount << " at position " << (i-j) << endl;
                                j = table[j];
                        }
                }
                else if( j > 0 )
                        j = table[j];
                else
                        ++i;
        }
        return matchcount;
}
```

Geometry

```
typedef double NUM; //use either double or long long
struct point
{
    NUM x, y;
    point(){}
    point(NUM _x, NUM _y) {x= _x; y= _y;}
    point(const point& p) {x=p.x; y=p.y;}
    point operator*(NUM scalar)       const { return point(scalar*x, scalar*y); }//scalar
    NUM   operator*(const point& rhs) const { return x*rhs.x + y*rhs.y; } //dot product
    NUM   operator^(const point& rhs) const { return x*rhs.y - y*rhs.x; } //cross product
    point operator+(const point& rhs) const { return point(x+rhs.x, y+rhs.y); }//addition
    point operator-(const point& rhs) const { return point(x-rhs.x, y-rhs.y); }//subtract
};

NUM sqDist(const point& a, const point& b) {
    return (b.x-a.x)*(b.x-a.x) + (b.y-a.y)*(b.y-a.y);
}

//--- Distance between two segments: ---
//Compute the distance both points from the first line segment to the full second segment
//BUT ALSO the distance from both points of the second line segment to the first segment
//Take the minimum of these four. (Or zero if they intersect)

//Distance SQUARED from a to line through bc
double sqDistPointLine(point a, point b, point c) {
    a = a-b;
    c = c-b;
    return (a^c)*(a^c)/((double)c*c);
}

//Distance SQUARED from point a to line segment bc
double sqDistPointSegment(point a, point b, point c)
{
    a = a-b;
    c = c-b;
    NUM dot = a*c;
    if( dot <= 0 ) return a*a;
    else
    {
        NUM len = c*c;
        if( dot >= len ) return (a-c)*(a-c);
        else return a*a - dot*dot/((double)len); //OR: (a^c)*(a^c) / ((double)len);
        //point projection = c * (dot/((double)len);
    }
}

//point a on segment bc
bool pointOnSegment(point a, point b, point c)
{
    a = a-b;
    c = c-b;
    NUM cross = a^c;
    if( cross != 0 ) return false;
    NUM dot = a*c; //a is on the line through b and c
    if( dot < 0 ) return false;
    if( dot > c*c ) return false;
    return true;
}
```

```
//Line segment a1---a2 intersects with b1---b2
bool segmentsIntersect(const point& a1, const point& a2, const point& b1, const point&
b2)
{
    point q = a2-a1;
    point r = b2-b1;
    point s = b1-a1;
    NUM cross = q^r;
    if( cross == 0 ){ //parallel
        NUM cross2 = q^s;
        if( cross2 != 0 ) return false; //no intersection
        //line segments lie in the extension of each other
        NUM v1 = s*q;
        NUM v2 = (b2-a1)*q;
        NUM v3 = q*q;
        if( v1 >= 0 && v1 <= v3 ) return true; //b1 is between a1 and a2
        if( v2 >= 0 && v2 <= v3 ) return true; //b2 is between a1 and a2
        if( v1 <= 0 && v2 >= v3 ) return true; //b1 is before a1 and b2 is after a2
        return false;
    }
    NUM c1 = s^r;
    NUM c2 = s^q;
    //We must check if 0 <= c1/cross <= 1 and 0 <= c2/cross <= 1
    if( cross > 0 ){
        if( c1 < 0 ) return false;
        if( c1 > cross ) return false;
        if( c2 < 0 ) return false;
        if( c2 > cross ) return false;
    }else{
        if( c1 > 0 ) return false;
        if( c1 < cross ) return false;
        if( c2 > 0 ) return false;
        if( c2 < cross ) return false;
    }
    //double t = (s^r) / ((double)cross);
    //double u = (s^q) / ((double)cross);
    //point intersect = a1*(1-t) + a2*t;
    //point intersect = b1*(1-u) + b2*u;
    return true;
}

//This returns TWICE the area of a polygon because then it will always be an integer if
the input is integers
NUM polygonTwiceArea(const vector<point>& polygon)
{
    //if( polygon.empty() ) return 0;
    NUM area = 0;
    point p0 = polygon[0];
    for(unsigned int i = 1; i+1 < polygon.size(); ++i) area += (polygon[i] -
p0)^(polygon[i+1]-p0);
    return (area > 0 ? area : -area); //abs(area)
}
```

```
//returns 0 outside, 1 inside, 2 on boundary
int pointInPolygon(point p, const vector<point>& polygon)
{
    //Check crossings with horizontal semi-line though p to +x
    int crosscount = 0;
    unsigned int N = polygon.size();
    for(unsigned int i = 0, j = N-1; i < N; j = i++)
    {
        if( pointOnSegment( p , polygon[j], polygon[i] ) ) return 2; //p on boundary
        //Check if it crosses the y=p.y line
        if( polygon[j].y > p.y ){
            if( polygon[i].y > p.y ) continue; //same side of line
            if( (p.x-polygon[i].x)*(polygon[j].y - polygon[i].y) < (polygon[j].x -
polygon[i].x)*(p.y - polygon[i].y) )
                ++crosscount;
        }else{
            if( !(polygon[i].y > p.y) ) continue; //same side of line
            if( (p.x-polygon[i].x)*(polygon[j].y - polygon[i].y) > (polygon[j].x -
polygon[i].x)*(p.y - polygon[i].y) )
                ++crosscount;
        }
    }
    if( crosscount % 2 == 0 ) return 0;
    else return 1;
}

//Assumes that polygon has unique points!
int pointInConvex(point p, const vector<point>& polygon)
{
    //The cross product should always have the same sign
    //when the point is inside the convex
    unsigned int N = polygon.size();
    int sign = 0;
    bool onExtendedBoundary = false;
    for(unsigned int i = 0, j = N-1; i < N; j = i++)
    {
        NUM cross = ((polygon[j] - p)^(polygon[i] - p));
        if( cross == 0 ) //epsilon when doubles
            onExtendedBoundary = true;
        else
        {
            if( sign == 0 ) sign = cross > 0 ? 1 : -1;
            else if( (sign == 1 && cross < 0) || (sign == -1 && cross > 0) ) return 0;
        }
    }
    if(onExtendedBoundary) return 2; //on boundary
    return 1; //inside convex
}
```

Convex Hull

```
struct comp //for sorting the points at the start of the scan
{
    comp(const vector<point>& p, const point& r) : points(p), reference(r) {};

    const vector<point>& points;
    const point& reference;

    bool operator() (int a, int b) const
    {
        //return true if points[a] is seen LOWER THAN points[b] when seen from reference
        //when on same line, return true if a is CLOSER to the reference
        NUM cross = ( (points[a]-reference)^(points[b]-reference) );
        if( cross > 0 ) return true;
        else if( cross == 0 ) return sqDist(reference, points[a]) < sqDist(reference,points[b]);
        return false;
    }
};
//in the output vector are the indices of the points array that belong to the hull
void convexHull(const vector<point>& points, vector<int>& output)
{
    output.clear();
    //IMPORTANT: If possible that the points array is LESS than 3 points, make this special case:
    //if( points.empty() ) return;
    //else if( points.size() == 1 ){ output.push_back(0); return; }
    //else if( points.size() == 2 ){ output.push_back(0); output.push_back(1); return; }
    unsigned int bestIndex = 0;
    NUM minX = points[0].x;
    NUM minY = points[0].y;
    for(unsigned int i = 1; i < points.size(); ++i)
        if( points[i].x < minX || (points[i].x == minX && points[i].y < minY) ){ bestIndex = i;
minX = points[i].x; minY = points[i].y; }

    vector<int> ordered; //index into points
    for(unsigned int i = 0; i < points.size(); ++i)
        if( i != bestIndex ) ordered.push_back(i);

    comp compare(points, points[bestIndex]);
    sort(ordered.begin(), ordered.end(), compare);

    output.push_back(bestIndex);
    output.push_back(ordered[0]);
    output.push_back(ordered[1]);
    for(unsigned int i = 2; i < ordered.size(); ++i)
    {
        //A = second to last element is output[output.size()-2]
        //B = last element is output.back()
        //C = next element is ordered[i]
        //We need to check wether the line ABC makes a right-turn at B, if so, delete it
        //Use the cross product on (A-B) and (C-B): delete the point if (A-B)^(C-B) > 0

        //NOTE: >  INCLUDES points on the hull-line
        //      >= EXCLUDES points on the hull-line
        while( output.size() > 1 && ((points[output[output.size()-2]] -
points[output.back()])^(points[ordered[i]] - points[output.back()])) > 0 )
            output.pop_back();

        output.push_back(ordered[i]);
    }
    return;
}
```

2-SAT

```
struct Implication
{
    Implication(int _id, char _val) : id(_id), value(_val) {};

    int id; //variable id
    char value; //0 or 1
};

struct VariableValue
{
    VariableValue() : value(-1) {}; //this constructor is important!
    char value; //-1 is unkown, 0 is false, 1 is true
};

struct VariableNode
{
    //imply[0] is the implications when this variable is false
    //imply[1] is the implications when this variable is true
    vector<Implication> imply[2];
};

typedef vector<VariableValue> ValueList;
typedef vector<VariableNode> ImplyList;
typedef vector<Implication>::const_iterator ImplyIter;

//return false if a contradiction occurred
//in both cases, the ValueList will be modified, so caller must save
//it in case the result is false, and then restore the original
bool propagate(ValueList& varlist, const ImplyList& implylist, int id, char value)
{
    if(varlist[id].value != -1) return varlist[id].value == value;
    varlist[id].value = value;

    for(ImplyIter iter = implylist[id].imply[value].begin(); iter !=
implylist[id].imply[value].end(); ++iter)
    {
        if(!propagate(varlist, implylist, iter->id, iter->value))
            return false;
    }
    return true;
}

//assumes ImplyList is already filled with (empty) entries
void addCondition(ImplyList& implylist, int id1, char value1, int id2, char value2)
{
    implylist[id1].imply[1-value1].push_back( Implication(id2, value2) );
    implylist[id2].imply[1-value2].push_back( Implication(id1, value1) );
}

int main()
{
    while(true)
    {
        int n, m; //variables, conditions
        cin >> n >> m;
        if(!cin.good()) break;

        ValueList variables(n+1); //1-based index
        ImplyList implications(n+1);
```

```
        for(int i = 0, a, b; i < m; ++i)
        {
            cin >> a >> b;
            addCondition(implications, abs(a), a < 0 ? 0 : 1, abs(b), b < 0 ? 0 : 1);
        }

        if(!propagate(variables, implications, 1, 1)) cout << "no" << endl; //Karl must
win
        else
        {
            bool success = true;
            for(int i = 2; i <= n; ++i)
            {
                if(variables[i].value != -1) continue; //already filled in
                //backup the state
                ValueList backup(variables);
                if( propagate(variables, implications, i, 1) ) continue;
                //restore state
                variables = backup;
                if( propagate(variables, implications, i, 0) ) continue;
                success = false;
                cout << "no" << endl;
                break;
            }
            if(success) cout << "yes" << endl;
        }
    }
    return 0;
}
```

Binary Indexed Tree

```
//THIS USES 1-BASED INDICES!!!! for better bitmask magic
//array of values:   f[i]
//cumulative values: c[i] = sum[ f[j] , {j,0,i} ]
//Modify values (f[i]) and read cumulative values (c[i]) both in log(n) time

const int MaxVal = 256;
int tree[MaxVal+1]; //Make sure to clear tree to zero first

int read(int idx){ //returns c[i]
    int sum = 0;
    while (idx > 0){
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}

void update(int idx ,int val){ //ADDS val to f[i]
    while (idx <= MaxVal){
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

void scale(int factor){ //multiplies all f[i] (and therefore c[i]) by factor
        for(int idx = 1; idx <= MaxVal; ++idx) tree[idx] *= factor;
}

// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// the greatest one
int find(int cumFre){
        int idx = 0;
        int bitMask = MaxVal;
        // bitMask - initialy, it is the greatest bit of MaxVal
        // bitMask store interval which should be searched
        while ((bitMask != 0) && (idx < MaxVal)){
                int tIdx = idx + bitMask;
                if (cumFre >= tree[tIdx]){
                        idx = tIdx;
                        cumFre -= tree[tIdx];
                }
                bitMask >>= 1;
        }
        if (cumFre != 0)
                return -1;
        else
                return idx;
}

//2-Dimensional
void update(int x , int y , int val){ //ADDS val to f[x,y]
        int y1;
        while (x <= max_x){
                y1 = y;
                while (y1 <= max_y){
                        tree[x][y1] += val;
                        y1 += (y1 & -y1);
                }
                x += (x & -x);
        }
}
void updatey(int x , int y , int val){
        while (y <= max_y){
                tree[x][y] += val;
                y += (y & -y);
        }
}
```

Big Integer

```
// base and base_digits must be consistent
const int base = 1000000000;
const int base_digits = 9;

struct bigint {
    vector<int> a;
    int sign;

    bigint() : sign(1) { }
    bigint(long long v) { *this = v;
    bigint(const string &s) { read(s); }
    void operator=(const bigint &v) { sign = v.sign; a = v.a; }

    void operator=(long long v) {
        sign = 1;
        if (v < 0)
            sign = -1, v = -v;
        for (; v > 0; v = v / base)
            a.push_back(v % base);
    }

    bigint operator+(const bigint &v) const {
        if (sign == v.sign) {
            bigint res = v;
            for (int i = 0, carry = 0; i < (int) max(a.size(), v.a.size()) || carry; ++i) {
                if (i == (int) res.a.size())
                    res.a.push_back(0);
                res.a[i] += carry + (i < (int) a.size() ? a[i] : 0);
                carry = res.a[i] >= base;
                if (carry)
                    res.a[i] -= base;
            }
            return res;
        }
        return *this - (-v);
    }

    bigint operator-(const bigint &v) const {
        if (sign == v.sign) {
            if (abs() >= v.abs()) {
                bigint res = *this;
                for (int i = 0, carry = 0; i < (int) v.a.size() || carry; ++i) {
                    res.a[i] -= carry + (i < (int) v.a.size() ? v.a[i] : 0);
                    carry = res.a[i] < 0;
                    if (carry)
                        res.a[i] += base;
                }
                res.trim();
                return res;
            }
            return -(v - *this);
        }
        return *this + (-v);
    }

    void operator*=(int v) {
        if (v < 0)
            sign = -sign, v = -v;
        for (int i = 0, carry = 0; i < (int) a.size() || carry; ++i) {
            if (i == (int) a.size())
                a.push_back(0);
            long long cur = a[i] * (long long) v + carry;
            carry = (int) (cur / base);
            a[i] = (int) (cur % base);
            //asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) : "A"(cur), "c"(base));
        }
        trim();
    }

    bigint operator*(int v) const {
        bigint res = *this;
        res *= v;
        return res;
    }

    friend pair<bigint, bigint> divmod(const bigint &a1, const bigint &b1) {
        int norm = base / (b1.a.back() + 1);
        bigint a = a1.abs() * norm;
        bigint b = b1.abs() * norm;
        bigint q, r;
        q.a.resize(a.a.size());

        for (int i = a.a.size() - 1; i >= 0; i--) {
            r *= base;
```

```
        r += a.a[i];
        int s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
        int s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
        int d = ((long long) base * s1 + s2) / b.a.back();
        r -= b * d;
        while (r < 0)
            r += b, --d;
        q.a[i] = d;
    }

    q.sign = a1.sign * b1.sign;
    r.sign = a1.sign;
    q.trim();
    r.trim();
    return make_pair(q, r / norm);
}

bigint operator/(const bigint &v) const {
    return divmod(*this, v).first;
}
bigint operator%(const bigint &v) const {
    return divmod(*this, v).second;
}

void operator/=(int v) {
    if (v < 0)
        sign = -sign, v = -v;
    for (int i = (int) a.size() - 1, rem = 0; i >= 0; --i) {
        long long cur = a[i] + rem * (long long) base;
        a[i] = (int) (cur / v);
        rem = (int) (cur % v);
    }
    trim();
}

bigint operator/(int v) const {
    bigint res = *this;
    res /= v;
    return res;
}

int operator%(int v) const {
    if (v < 0)
        v = -v;
    int m = 0;
    for (int i = a.size() - 1; i >= 0; --i)
        m = (a[i] + m * (long long) base) % v;
    return m * sign;
}

bool operator<(const bigint &v) const {
    if (sign != v.sign)
        return sign < v.sign;
    if (a.size() != v.a.size())
        return a.size() * sign < v.a.size() * v.sign;
    for (int i = a.size() - 1; i >= 0; i--)
        if (a[i] != v.a[i])
            return a[i] * sign < v.a[i] * sign;
    return false;
}

bool operator>(const bigint &v) const {
    return v < *this;
}
bool operator<=(const bigint &v) const {
    return !(v < *this);
}
bool operator>=(const bigint &v) const {
    return !(*this < v);
}
bool operator==(const bigint &v) const {
    return !(*this < v) && !(v < *this);
}
bool operator!=(const bigint &v) const {
    return *this < v || v < *this;
}

void trim() {
    while (!a.empty() && !a.back())
        a.pop_back();
    if (a.empty())
        sign = 1;
}

bool isZero() const {
    return a.empty() || (a.size() == 1 && !a[0]);
}
```

```
bigint operator-() const {
    bigint res = *this;
    res.sign = -sign;
    return res;
}

bigint abs() const {
    bigint res = *this;
    res.sign *= res.sign;
    return res;
}

long long longValue() const {
    long long res = 0;
    for (int i = a.size() - 1; i >= 0; i--)
        res = res * base + a[i];
    return res * sign;
}

friend bigint gcd(const bigint &a, const bigint &b) {
    return b.isZero() ? a : gcd(b, a % b);
}
friend bigint lcm(const bigint &a, const bigint &b) {
    return a / gcd(a, b) * b;
}

void read(const string &s) {
    sign = 1;
    a.clear();
    int pos = 0;
    while (pos < (int) s.size() && (s[pos] == '-' || s[pos] == '+')) {
        if (s[pos] == '-')
            sign = -sign;
        ++pos;
    }
    for (int i = s.size() - 1; i >= pos; i -= base_digits) {
        int x = 0;
        for (int j = max(pos, i - base_digits + 1); j <= i; j++)
            x = x * 10 + s[j] - '0';
        a.push_back(x);
    }
    trim();
}

friend ostream& operator<<(ostream &stream, const bigint &v) {
    if (v.sign == -1)
        stream << '-';
    stream << (v.a.empty() ? 0 : v.a.back());
    for (int i = (int) v.a.size() - 2; i >= 0; --i)
        stream << setw(base_digits) << setfill('0') << v.a[i];
    return stream;
}

static vector<int> convert_base(const vector<int> &a, int old_digits, int new_digits) {
    vector<long long> p(max(old_digits, new_digits) + 1);
    p[0] = 1;
    for (int i = 1; i < (int) p.size(); i++)
        p[i] = p[i - 1] * 10;
    vector<int> res;
    long long cur = 0;
    int cur_digits = 0;
    for (int i = 0; i < (int) a.size(); i++) {
        cur += a[i] * p[cur_digits];
        cur_digits += old_digits;
        while (cur_digits >= new_digits) {
            res.push_back(int(cur % p[new_digits]));
            cur /= p[new_digits];
            cur_digits -= new_digits;
        }
    }
    res.push_back((int) cur);
    while (!res.empty() && !res.back())
        res.pop_back();
    return res;
}

typedef vector<long long> vll;

static vll karatsubaMultiply(const vll &a, const vll &b) {
    int n = a.size();
    vll res(n + n);
    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                res[i + j] += a[i] * b[j];
        return res;
```

```
        }

        int k = n >> 1;
        vll a1(a.begin(), a.begin() + k);
        vll a2(a.begin() + k, a.end());
        vll b1(b.begin(), b.begin() + k);
        vll b2(b.begin() + k, b.end());

        vll a1b1 = karatsubaMultiply(a1, b1);
        vll a2b2 = karatsubaMultiply(a2, b2);

        for (int i = 0; i < k; i++)
            a2[i] += a1[i];
        for (int i = 0; i < k; i++)
            b2[i] += b1[i];

        vll r = karatsubaMultiply(a2, b2);
        for (int i = 0; i < (int) a1b1.size(); i++)
            r[i] -= a1b1[i];
        for (int i = 0; i < (int) a2b2.size(); i++)
            r[i] -= a2b2[i];

        for (int i = 0; i < (int) r.size(); i++)
            res[i + k] += r[i];
        for (int i = 0; i < (int) a1b1.size(); i++)
            res[i] += a1b1[i];
        for (int i = 0; i < (int) a2b2.size(); i++)
            res[i + n] += a2b2[i];
        return res;
    }

    bigint operator*(const bigint &v) const {
        vector<int> a6 = convert_base(this->a, base_digits, 6);
        vector<int> b6 = convert_base(v.a, base_digits, 6);
        vll a(a6.begin(), a6.end());
        vll b(b6.begin(), b6.end());
        while (a.size() < b.size())
            a.push_back(0);
        while (b.size() < a.size())
            b.push_back(0);
        while (a.size() & (a.size() - 1))
            a.push_back(0), b.push_back(0);
        vll c = karatsubaMultiply(a, b);
        bigint res;
        res.sign = sign * v.sign;
        for (int i = 0, carry = 0; i < (int) c.size(); i++) {
            long long cur = c[i] + carry;
            res.a.push_back((int) (cur % 1000000));
            carry = (int) (cur / 1000000);
        }
        res.a = convert_base(res.a, 6, base_digits);
        res.trim();
        return res;
    }
};
```