Codebook- Team bits_dont_lie
IIT Delhi, India

Ankesh Gupta, Ronak Agarwal, Anant Chhajwani

Contents

1 Format

1.1 Format c++

```
#include <bits/stdc++.h>
typedef long double ld ;
#define long long long int
using namespace std;
#define pi 3.141592653589

template<class T> ostream& operator<<(ostream &os, vector<T> V) {
 os << "[ ";
 for(auto v : V) os << v << " ";
 return os << "]";
}
template<class L, class R> ostream& operator<<(ostream &os, pair<L,R> P) {
 return os << "(" << P.first << "," << P.second << ")";
}
#define TRACE
#ifdef TRACE
 #define trace(...) __f(#__VA_ARGS__, __VA_ARGS__)
 template <typename Arg1>
 void __f(const char* name, Arg1&& arg1){
  cerr << name << " : " << arg1 << endl;
 }
 template <typename Arg1, typename... Args>
 void __f(const char* names, Arg1&& arg1, Args&&... args){
  const char* comma = strchr(names + 1, ',');
  cerr.write(names, comma - names) << " : " << arg1 <<" | ";
  __f(comma+1, args...);
 }
#else
 #define trace(...)
#endif

long GCD(long a,long b){
 while(a && b){
  a=a%b;
  if(a!=0)
   b=b%a;
 }
 return a+b;
}

long exp(long a,long n){
 long ans=1;
 a=check(a);
 while(n){
  if(n&1)
   ans=check(ans*a);
  a=check(a*a);
  n=(n>>1);
 }
 return ans;
}

/*Finding Unique Elements
*/
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

1.2 String Input c++

```
cin.ignore();
for(int j=0;j<lines;j++){
 getline(cin,x);
 stringstream check1(x);
 string tokens;
    while(getline(check1, tokens, ' ')){
     if(tokens=="import")
      continue;
    }
}
```

2 Strings

2.1 KMP

```
//Takes an array of characters and calculate
```

```
//lcp[i] where lcp[i] is the longest proper suffix of the
//string c[0..i] such that it is also a prefix of the string.
vector<int> kmp(const string &str){
 int n = str.size();
 vector<int> lcp(n,0);
 for(int i=1 ; i<n ; i++){
  int j = lcp[i-1];
  while(j!=0 && str[i]!=str[j]) j = lcp[j-1];
  if(str[i]==str[j]) j++;
  lcp[i]=j;
 }
 return lcp;
}
```

2.2 AhoCohrasick

```
int m=0;
struct Trie{
 int chd[26];
 int cnt,mcnt,d=-1,p=-1,pch;
 int sLink = -1;
 Trie(int p,int pch,int d): cnt(0), mcnt(0), d(d), p(p), pch(pch){
  for(int i=0 ; i<26 ; i++) chd[i]=-1;
 }
};
const int N = 5e5;
Trie* nds[N];
void addVal(const string &str){
 int v=0;
 for(char chr : str){
  int idx = chr-'a';
  if(nds[v]->chd[idx]==-1){
   nds[v]->chd[idx] = m;
   nds[m++] = new Trie(v,idx,(nds[v]->d)+1);
  }
  v = nds[v]->chd[idx];
  nds[v]->cnt+=nds[v]->d;
 }
}
void AhoCorasick(){
 queue<int> q;
 q.push(0);
 while(!q.empty()){
  int v = q.front();
  q.pop();
  for(int i=0 ; i<26 ; i++)
   if(nds[v]->chd[i]!=-1)
    q.push(nds[v]->chd[i]);
  if(nds[v]->p==0 || nds[v]->p==-1){
   nds[v]->sLink = 0;
   nds[v]->mcnt = nds[v]->cnt;
   continue;
  }
  int b = nds[v]->pch;
  int av = nds[nds[v]->p]->sLink;
  int nLink = 0;
  while(true){
   if(nds[av]->chd[b]!=-1){
    nLink = nds[av]->chd[b];
    break;
   }
   if(av == nds[av]->sLink) break;
   av = nds[av]->sLink;
  }
  nds[v]->sLink = nLink;
  nds[v]->mcnt = max(nds[v]->cnt,nds[nLink]->mcnt);
 }
}
```

2.3 Manacher

```
vector<int> manacher(const string &str){
 int n = str.size();
 vector<int> M(n,1);
 int R = 2; int C = 1;
 for(int i=1 ; i<n ; i++){
  int len = 0;
  if(i<R) len = min(manacher[2*C-i],R-i);
  if(i+len==R){
   while(i>=len && str[i-len]==str[i+len]){
    C = i;
    len++; R++;
   }
  }
  M[i] = len;
 }
 return M;
}
```

2.4 Suffix_Array

```
struct SuffixArray {
 const int L;
 string s;
 vector<vector<int> > P;
 vector<pair<pair<int,int>,int> > M;
 vector<int> Suf,rank,LCParr;
 // returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]
 int LongestCommonPrefix(int i,int j) {
  int len = 0;
  if(i==j) return (L-i);
  for (int k=P.size()-1 ; k>=0 && i<L && j<L; k--){
   if(P[k][i]==P[k][j]){
    i+=(1<<k); j+=(1<<k);
    len+=(1<<k);
   }
  }
  return len;
 }
 //Suf[i] denotes the suffix at i^th rank
 //Rank[i] denotes the rank of the i^th suffix
 //LCP[i] the longest common prefix of the suffixes at ith and (i+1)th rank.
 SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L), rank(L), LCParr(L-1){
  vector<int> chars(L,0);
  for(int i=0 ; i<L ; i++) chars[i] = int(s[i]);
  sort(chars.begin(), chars.end());
  map<int,int> mymap;
  int ptr=0;
  for(int elem : chars) mymap[elem] = ptr++;

  for(int i=0 ; i<L ; i++) P[0][i] = mymap[int(s[i])];
  for(int skip=1,level=1 ; skip<L ; skip*=2,level++){
   P.pb(vector<int>(L, 0));
   for(int i = 0; i < L; i++)
    M[i] = mp(mp(P[level-1][i], (i+skip)<L ? P[level-1][i+skip] : -1000), i);
   sort(M.begin(),M.end());
   for(int i = 0; i < L; i++)
    P[level][M[i].Y] = (i > 0 && M[i].X == M[i-1].X) ? P[level][M[i-1].Y] : i;
  }
  Suf = P.back();
  for(int i=0 ; i<L ; i++) rank[Suf[i]] = i;
  for(int i=0 ; i<(L-1) ; i++) LCParr[i] = LongestCommonPrefix(rank[i],rank[i+1]);
 }
};
```

2.5 Z algo

```
vector<int> Z_algo(const string &str){
 int n = str.size();
 vector<int> Z(n,0);
 int L=0,R=0;
 for(int i=1 ; i<n ; i++)
  if(i>R){
   L=i; R=i;
   while(R<n && str[R]==str[R-L]) R++;
   R--; Z[i] = (R-L+1);
  }else{
   int j = i-L;
   if(Z[j]<(R-i+1)) Z[i] = Z[j];
   else{
    L=i;
    while(R<n && str[R]==str[R-L]) R++;
    R--; Z[i] = (R-L+1);
   }
  }
 return Z;
}
```

2.6 Hashing

```
long p1=2350490027,p2=1628175011;
long p3=2911165193,p4=1040332871;
2350490027,2125898167,1628175011,1749241873,
1593209441,1524872353,1040332871,2911165193,
1387346491,2776808933
```

3 Trees

3.1 Centroid Tree

```
vector<int> graph[3*Max];
int size[3*Max];
bool usable[3*Max];
int centroid_parent[3*Max];
void calc_size(int i,int pa){
 size[i]=1;
 for(auto itr:graph[i]){
  if(itr!=pa && usable[itr]){
   calc_size(itr,i);
   size[i]+=size[itr];
  }
 }
}
int getCentroid(int i,int len,int pa){
 for(auto itr:graph[i]){
  if(itr!=pa && usable[itr]){
   if(size[itr]>(len/2))
    return getCentroid(itr,len,i);
  }
 }
 return i;
}
void build_centroid(int i,int coun){
 queue<pair<int,int> > q;
 q.push({i,-1});
 while(!q.empty()){
  auto itr=q.front();
  q.pop();
  calc_size(itr.x,-1);
  int centroid=getCentroid(itr.x,size[itr.x],-1);
  centroid_parent[centroid]=itr.y;
  for(auto itr2:graph[centroid]){
   if(usable[itr2]){
    q.push({itr2,centroid});
   }
  }
  usable[centroid]=false;
 }
}
```

3.2 Heavy Light Decomposition

```
int chainNo[Max];
int pos_in_chain[Max];
int parent_in_chain[Max];
int parent[Max];
int chain_count=0;
int total_in_chain[Max];
int pos_count=0;
vector<int> graph[Max];
int arr[Max];
int subtree_count[Max];
int max_in_subtree[Max];
int height[Max];
vector<vector<pair<int,int> > > vec;
int max_elem,max_count;
void simple_dfs(int i){
 subtree_count[i]=1;
 int max_val=0;
 int ind=-1;
 for(auto itr:graph[i]){
  height[itr]=1+height[i];
  simple_dfs(itr);
  subtree_count[i]+=subtree_count[itr];
  if(max_val<subtree_count[itr]){
   max_val=subtree_count[itr];
   ind=itr;
  }
 }
 max_in_subtree[i]=ind;
}
void dfs(int i){
 if(pos_count==0)
  parent_in_chain[chain_count]=i;
 chainNo[i]=chain_count;
 pos_in_chain[i]=++pos_count;
 total_in_chain[chain_count]++;
 if(max_in_subtree[i]!=-1){
  dfs(max_in_subtree[i]);
 }
 for(auto itr:graph[i]){
  if(itr!=max_in_subtree[i]){
   chain_count++;
   pos_count=0;
   dfs(itr);
  }
 }
}
int pos;int chain;int val;
void update(int s,int e,int n){
 if(pos>e || pos<s)
  return;
 vec[chain][n]={val,1};
 if(s==e)
  return;
```

```
 int mid=(s+e)>>1;
 update(s,mid,2*n);
 update(mid+1,e,2*n+1);
 if(vec[chain][2*n].x<vec[chain][2*n+1].x)
  vec[chain][n]=vec[chain][2*n+1];
 else if(vec[chain][2*n].x>vec[chain][2*n+1].x)
  vec[chain][n]=vec[chain][2*n];
 else{
  vec[chain][n]={vec[chain][2*n].x,vec[chain][2*n].↩
     y+vec[chain][2*n+1].y};
 }
}
int qs;int qe;
void query_tree(int s,int e,int n){
 if(s>qe || qs>e)
  return;
 if(s>=qs && e<=qe){
  if(vec[chain][n].x>max_elem){
   max_elem=vec[chain][n].x;
   max_count=vec[chain][n].y;
  }
  else if(vec[chain][n].x==max_elem){
   max_count+=vec[chain][n].y;
  }
  return;
 }
 if(vec[chain][n].x <max_elem)
  return;
 int mid=(s+e)>>1;
 query_tree(s,mid,2*n);
 query_tree(mid+1,e,2*n+1);
}
void query(int i){
 if(i==-1)
  return;
 qs=1;qe=pos_in_chain[i];chain=chainNo[i];
 query_tree(1,total_in_chain[chainNo[i]],1);
 i=parent[parent_in_chain[chainNo[i]]];
 query(i);
}
```

3.3 Heavy Light Trick

```
void dfs(int i,int pa){
 int coun=1 ;
 for(auto itr:a[i]){
  if(itr.x!=pa){
   prod[itr.x]=check(prod[i]*itr.y) ;
   dfs(itr.x,i);
   coun+=siz[itr.x] ;
  }
 }
 siz[i]=coun ;
}
long ans=0 ;
void add(int i,int pa,int x){
 coun[mapped_prod[i]]+=x ;
 for(auto itr:a[i])
  if(itr.x!=pa && !big[itr.x])
   add(itr.x,i,x) ;
}
void solve(int i,int pa){
 long temp=check(multi*inv[i]);
 int xx=m[temp];
 ans+=coun[xx];
 for(auto itr:a[i])
  if(itr.x!=pa && !big[itr.x])
   solve(itr.x,i) ;
}
void dfs2(int i,int pa,bool keep){
 int mx=-1,bigc=-1;
 for(auto itr:a[i]){
  if(itr.x!=pa){
   if(siz[itr.x]>mx)
   mx=siz[itr.x],bigc=itr.x;
  }
 }
 for(auto itr:a[i]){
  if(itr.x!=pa && itr.x!=bigc)
   dfs2(itr.x,i,0);
 }
 if(bigc!=-1){
  dfs2(bigc,i,1);
  big[bigc]=true;
 }
 multi=check(p*check(prod[i]*prod[i]));
 long temp=check(p*prod[i]);
 ans+=coun[m[temp]];
 coun[mapped_prod[i]]++;
 for(auto itr:a[i])
  if(itr.x!=pa && !big[itr.x]){
   solve(itr.x,i);
   add(itr.x,i,1);
  }
 if(bigc!=-1)
  big[bigc]=false;
 if(keep==0)
  add(i,pa,-1);
}
```

3.4 LCA

```
int pa[21][3*N], level[3*N];
int lca(int u,int v){
 if(level[u]>level[v])return lca(v,u);
 for(long i=19;i>=0 && level[v]!=level[u];i--){
  if(level[v]>=level[u]+(1<<i))
   v=pa[i][v];
 }
 if(u==v)return u;
 for(long i=19;i>=0;i--){
  if(pa[i][u]!=pa[i][v]){
   u=pa[i][u];v=pa[i][v];
  }
 }
 return pa[0][u];
}
```

4 Graph and Matching, Flows

4.1 Euler Walk

```
vector<pair<int,int> > graph[202];
bool visited[202];
vector<int> odd;
bool used_edges[41000];
stack<int> s;
int tot_edges;
int counter[202];

void dfs(int i)
{
  visited[i]=true;

  int len=graph[i].size();
  if(len&1)
    odd.pb(i);

  for(auto itr:graph[i])
    if(!visited[itr.x])
      dfs(itr.x);
}
void euler_tour(int i)
{
    visited[i]=true;
    s.push(i);

    int x=graph[i].size();
    while(counter[i]<x)
    {
      auto itr=graph[i][counter[i]];
      counter[i]++;

      if(!used_edges[itr.y])
      {
        used_edges[itr.y]=true;
        if(itr.y<=tot_edges)
          cout<<i<<" "<<itr.x<<"\n";
        euler_tour(itr.x);
      }
    }
    s.pop();
}
```

4.2 Articulation Point Pseudo

```
ArtPt(v){
 color[v] = gray;
 Low[v] = d[v] = ++time;
 for all w in Adj(v) do {
  if (color[w] == white){
   pred[w] = v;
   ArtPt(w) ;
   if (pred [v] == NULL) {
    if ('w' is v''s second child) output v;
   }
   else if (Low[w] >= d[v]) output v;
   Low[v] = min(Low[v], Low[w]);
  }
  else if (w != pred[v]){
   Low[v] = min(Low[v], d[w]);
  }
 }
 color[v] = black;
}
```

4.3 Ford Fulkerson

```
const int N=250;
const int M=210*26*2;

int n,m;
vector<pair<int,int> > graph[N];
int edge_count=0;
int visited_from[N];
int edge_entering[N];
int reverse_no[M];
int capacity[M];
int max_flow_dfs[N];

void addEdge(int x,int y,int cap)
{
 ++edge_count;
 capacity[edge_count]=cap;
 graph[x].pb({y,edge_count});
 ++edge_count;
 capacity[edge_count]=0;
 graph[y].pb({x,edge_count});
 reverse_no[edge_count]=edge_count-1;
 reverse_no[edge_count-1]=edge_count;
}

void dfs(int source)
{
 // cout<<source<<endl;
 for(auto itr:graph[source])
 {
  if(visited_from[itr.x]==-1 && capacity[itr.y])
  {
   edge_entering[itr.x]=itr.y;
   visited_from[itr.x]=source;
   max_flow_dfs[itr.x]=min(capacity[itr.y],↩
      max_flow_dfs[source]);
   dfs(itr.x);
  }
 }
 // cout<<source<<endl;
}

void reverse_edge(int i,int flow)
{
 while(visited_from[i]!=0)
 {
  capacity[edge_entering[i]]-=flow;
  capacity[reverse_no[edge_entering[i]]]+=flow;
  i=visited_from[i];
 }
}

int ford_faulkerson(int source,int sink,int n)
{
 int ans=0;
 // cout<<n<<endl;
 while(true)
 {
  for(int i=1;i<=n;i++)
   visited_from[i]=-1;
  visited_from[source]=0;
  max_flow_dfs[source]=1e9;

  dfs(source);
  if(visited_from[sink]==-1)
   break;
  ans+=max_flow_dfs[sink];
  reverse_edge(sink,max_flow_dfs[sink]);
 }
 return ans;
}
```

4.4 Max Bipartite Matching $O(EV)$

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in↩
   practice
//
//   INPUT: w[i][j] = edge between row node i and ↩
   column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 ↩
   if unassigned
//           mc[j] = assignment for column node j, ↩
   -1 if unassigned
//           function returns number of matches ↩
   made

#include <vector>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc,↩
    VI &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, ↩
         seen)) {
        mr[i] = j;
```

```
            mc[j] = i;
            return true;
        }
      }
    }
    return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc)↩
    {
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);

    int ct = 0;
    for (int i = 0; i < w.size(); i++) {
      VI seen(w[0].size());
      if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
    return ct;
}
```

4.5 Dinic- Maximum Flow $O(EV^2)$

```
struct Edge {
    int a, b, cap, flow;
};
struct MaxFlow {
    int n, s, t;
    vector<int> d, ptr, q;
    vector< Edge > e;
    vector< vector<int> > g;
    int i,j;
    MaxFlow(int n) : n(n), d(n), ptr(n), q(n), g(n)↩
        {
        e.clear();
        for(int i=0;i<n;i++) {
            g[i].clear();
            ptr[i] = 0;
        }
    }
    void addEdge(int a, int b, int cap) {
        Edge e1 = { a, b, cap, 0 };
        Edge e2 = { b, a, 0, 0 };
        g[a].push_back( (int) e.size() );
        e.push_back(e1);
        g[b].push_back( (int) e.size() );
        e.push_back(e2);
    }
    int getMaxFlow(int _s, int _t) {
        s = _s; t = _t;
        int flow = 0;
        for (;;) {
            if (!bfs()) break;
            for(int i=0;i<n;i++) ptr[i] = 0;
            while (int pushed = dfs(s, INF))
                flow += pushed;
        }
        return flow;
    }
private:
    bool bfs() {
        int qh = 0, qt = 0;
        q[qt++] = s;
        for(int i=0;i<n;i++) d[i] = -1;
        d[s] = 0;

        while (qh < qt && d[t] == -1) {
            int v = q[qh++];
            int gv_sz=g[v].size();
            for(int i=0;i<gv_sz;i++) {
                int id = g[v][i], to = e[id].b;
                if (d[to] == -1 && e[id].flow < e[↩
                   id].cap) {
                    q[qt++] = to;
                    d[to] = d[v] + 1;
                }
            }
        }
        return d[t] != -1;
    }
    int dfs (int v, int flow) {
        if (!flow) return 0;
        if (v == t) return flow;
        for (; ptr[v] < (int)g[v].size(); ++ptr[v])↩
            {
            int id = g[v][ptr[v]],
                to = e[id].b;
            if (d[to] != d[v] + 1) continue;
            int pushed = dfs(to, min(flow, e[id].↩
               cap - e[id].flow));
            if (pushed) {
                e[id].flow += pushed;
                e[id^1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
};
```

4.6 Minimum Cost Bipartite Matching $O(V^3)$

```
// Min cost bipartite matching via shortest ↩
   augmenting path
// This is an O(n^3) implementation of a shortest ↩
   augmenting path
// algorithm for finding min cost perfect matchings↩
    in dense
// graphs. In practice, it solves 1000x1000 ↩
   problems in around 1 second.
//  cost[i][j] = cost for pairing left node i with ↩
   right node j
//  Lmate[i] = index of right node that left node i↩
    pairs with
//  Rmate[j] = index of left node that right node j↩
    pairs with
// The values in cost[i][j] may be positive or ↩
   negative.To perform
// maximization, simply negate the cost[][] matrix.
typedef vector<long> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
long MinCostMatching(const VVD &cost, VI &Lmate, VI↩
    &Rmate) {
 int n = int(cost.size());
 // construct dual feasible solution
 VD u(n); VD v(n) ;
 for (int i = 0; i < n; i++) {
  u[i] = cost[i][0];
  for (int j = 1; j < n; j++) u[i] = min(u[i], cost↩
     [i][j]) ;
 }
 for (int j = 0; j < n; j++) {
  v[j] = cost[0][j] - u[0];
  for (int i = 1; i < n; i++) v[j] = min(v[j], cost↩
     [i][j] - u[i]) ;
 } // construct primal solution satisfying ↩
    complementary slackness
 Lmate = VI(n, -1);
 Rmate = VI(n, -1);
 int mated = 0;
 for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
   if (Rmate[j] != -1) continue;
   if ((cost[i][j] - u[i] - v[j])==0){
    Lmate[i] = j;
    Rmate[j] = i;
    mated++;
    break;
   }
  }
 }
 VD dist(n); VI dad(n); VI seen(n);
 // repeat until primal solution is feasible
 while (mated < n) {   // find an unmatched left ↩
    node
  int s = 0;
  while (Lmate[s] != -1) s++;   // initialize ↩
     Dijkstra
  fill(dad.begin(), dad.end(), -1);
  fill(seen.begin(), seen.end(), 0);
  for (int k = 0; k < n; k++)
   dist[k] = cost[s][k] - u[s] - v[k];
  int j = 0;
  while (true){    // find closest
   j = -1;
   for (int k = 0; k < n; k++) {
    if (seen[k]) continue ;
    if (j == -1 || dist[k] < dist[j]) j = k;
   }
   seen[j] = 1 ;    // termination condition
   if (Rmate[j] == -1) break ;    // relax ↩
     neighbors
   const int i = Rmate[j] ;
   for (int k = 0; k < n; k++) {
    if (seen[k]) continue;
    const long new_dist = dist[j] + cost[i][k] - u[↩
       i] - v[k];
    if (dist[k] > new_dist) {
     dist[k] = new_dist;
     dad[k] = j;
    }
   }
  }   // update dual variables
  for (int k = 0; k < n; k++) {
   if (k == j || !seen[k]) continue;
   const int i = Rmate[k];
   v[k] += dist[k] - dist[j];
   u[i] -= dist[k] - dist[j];
  }
  u[s] += dist[j] ;   // augment along path
  while (dad[j] >= 0) {
   const int d = dad[j];
   Rmate[j] = Rmate[d];
   Lmate[Rmate[j]] = j;
   j = d;
  }
  Rmate[j] = s; Lmate[s] = j;
  mated++;
 }
 long value = 0;
 for (int i = 0; i < n; i++)
  value += cost[i][Lmate[i]];
 return value;
```

```
}
VVD cost;
cost.resize(n+m-1);
VI Lmate,Rmate;
MinCostMatching(cost,Lmate,Rmate)
```

4.7 Minimum Cost Maximum Flow

```
struct Edge {
    int u, v;
    long long cap, cost;
    Edge(int _u, int _v, long long _cap, long long _cost) {
        u = _u; v = _v; cap = _cap; cost = _cost;
    }
};
struct MinimumCostMaximumFlow{
    int n, s, t;
    long long flow, cost;
    vector<vector<int> > graph;
    vector<Edge> e;
    vector<long long> dist;
    vector<int> parent;
    MinimumCostMaximumFlow(int _n){
        // 0-based indexing
        n = _n;
        graph.assign(n, vector<int> ());
    }
    void add(int u, int v, long long cap, long long cost, bool directed = true){
        graph[u].push_back(e.size());
        e.push_back(Edge(u, v, cap, cost));
        graph[v].push_back(e.size());
        e.push_back(Edge(v, u, 0, -cost));
        if(!directed)
            add(v, u, cap, cost, true);
    }
    pair<long long, long long> getMinCostFlow(int _s, int _t){
        s = _s; t = _t;
        flow = 0, cost = 0;
        while(SPFA()){
            flow += sendFlow(t, 1LL<<62);
        }
        return make_pair(flow, cost);
    }
    bool SPFA(){
        parent.assign(n, -1);
        dist.assign(n, 1LL<<62);        dist[s] = 0;
        vector<int> queuetime(n, 0);    queuetime[s] = 1;
        vector<bool> inqueue(n, 0);     inqueue[s] = true;
        queue<int> q;                   q.push(s);
        bool negativecycle = false;
        while(!q.empty() && !negativecycle){
            int u = q.front(); q.pop(); inqueue[u] = false;
            for(int i = 0; i < graph[u].size(); i++){
                int eIdx = graph[u][i];
                int v = e[eIdx].v, w = e[eIdx].cost, cap = e[eIdx].cap;
                if(dist[u] + w < dist[v] && cap > 0){
                    dist[v] = dist[u] + w;
                    parent[v] = eIdx;
                    if(!inqueue[v]){
                        q.push(v);
                        queuetime[v]++;
                        inqueue[v] = true;
                        if(queuetime[v] == n+2){
                            negativecycle = true;
                            break;
                        }
                    }
                }
            }
        }
        return dist[t] != (1LL<<62);
    }
    long long sendFlow(int v, long long curFlow){
        if(parent[v] == -1)
            return curFlow;
        int eIdx = parent[v];
        int u = e[eIdx].u, w = e[eIdx].cost;
        long long f = sendFlow(u, min(curFlow, e[eIdx].cap));
        cost += f*w;
        e[eIdx].cap -= f;
        e[eIdx^1].cap += f;
        return f;
    }
};
```

```
int source=2*n+1;
int sink=2*n+2;
MinimumCostMaximumFlow mcmf(id+10);
mcmf.add(source,i,1,k);
cout<<mcmf.getMinCostFlow(source,sink).second<<endl;
```

4.8 Push Relabel Max Flow($O(V^3)$ vs $O(V^2\sqrt{E})$)

```
// Running time:
//      O(|V|^3)
// INPUT:
//      - graph, constructed using AddEdge()
//      - source
//      - sink
// OUTPUT:
//      - maximum flow value
//      - To obtain the actual flow values, look at ↩
   all edges with
//        capacity > 0 (zero capacity edges are ↩
   residual edges).

typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int ↩
     index) :
    from(from), to(to), cap(cap), flow(flow), index↩
       (index) {}
};

struct PushRelabel {
  int N;
  vector<vector<Edge> > G;
  vector<LL> excess;
  vector<int> dist, active, count;
  queue<int> Q;

  PushRelabel(int N) : N(N), G(N), excess(N), dist(↩
     N), active(N), count(2*N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].↩
       size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].↩
       size() - 1));
  }

  void Enqueue(int v) {
    if (!active[v] && excess[v] > 0) { active[v] = ↩
       true; Q.push(v); }
  }

  void Push(Edge &e) {
    int amt = int(min(excess[e.from], LL(e.cap - e.↩
       flow)));
    if (dist[e.from] <= dist[e.to] || amt == 0) ↩
       return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
  }

  void Gap(int k) {
    for (int v = 0; v < N; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--;
      dist[v] = max(dist[v], N+1);
      count[dist[v]]++;
      Enqueue(v);
    }
  }

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

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

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

    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
```

```
      active[v] = false;
      Discharge(v);
    }

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

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

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

4.9 General Unweighted Maximum Matching (Edmonds' algorithm)

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

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

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

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

// Call this after constructing G
void Solve()
{
  for (int i = 1; i <= V; i++)
    if (Mate[i] == 0)
      {
```

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

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

4.10 König's Theorem (Text)

In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. To exhibit the vertex cover:

1. Find a maximum matching
2. Change each edge **used** in the matching into a directed edge from **right to left**
3. Change each edge **not used** in the matching into a directed edge from **left to right**
4. Compute the set T of all vertices reachable from unmatched vertices on the left (including themselves)
5. The vertex cover consists of all vertices on the right that are **in** T, and all vertices on the left that are **not in** T

4.11 Minimum Edge Cover (Text)

If a minimum edge cover contains C edges, and a maximum matching contains M edges, then $C + M = |V|$. To obtain the edge cover, start with a maximum matching, and then, for every vertex not matched, just select some edge incident upon it and add it to the edge set.

5 Data Structures

5.1 BIT- Range Update + Range Sum

```
// BIT with range updates, inspired by Petr ↩
   Mitrichev
struct BIT {
    int n;
    vector<int> slope;
    vector<int> intercept;
    // BIT can be thought of as having entries f↩
       [1], ..., f[n]
    // which are 0-initialized
    BIT(int n): n(n), slope(n+1), intercept(n+1) {}
    // returns f[1] + ... + f[idx-1]
    // precondition idx <= n+1
    int query(int idx) {
        int m = 0, b = 0;
        for (int i = idx-1; i > 0; i -= i&-i) {
```

```
            m += slope[i];
            b += intercept[i];
        }
        return m*idx + b;
    }
    // adds amt to f[i] for i in [idx1, idx2)
    // precondition 1 <= idx1 <= idx2 <= n+1 (you can't update element 0)
    void update(int idx1, int idx2, int amt) {
        for (int i = idx1; i <= n; i += i&-i) {
            slope[i] += amt;
            intercept[i] -= idx1*amt;
        }
        for (int i = idx2; i <= n; i += i&-i) {
            slope[i] -= amt;
            intercept[i] += idx2*amt;
        }
    }
};
update(ft, p, v):
  for (; p <= N; p += p&(-p))
    ft[p] += v

# Add v to A[a...b]
update(a, b, v):
  update(B1, a, v)
  update(B1, b + 1, -v)
  update(B2, a, v * (a-1))
  update(B2, b + 1, -v * b)

query(ft, b):
  sum = 0
  for(; b > 0; b -= b&(-b))
    sum += ft[b]
  return sum

# Return sum A[1...b]
query(b):
  return query(B1, b) * b - query(B2, b)

# Return sum A[a...b]
query(a, b):
return query(b) - query(a-1)
```

5.2 BIT- 2D

```
void update(int x , int y , int val){
    while (x <= max_x){
        updatey(x , y , val);
        // this function should update array tree[x]
        x += (x & -x);
    }
}
void updatey(int x , int y , int val){
    while (y <= max_y){
        tree[x][y] += val;
        y += (y & -y);
    }
}
void update(int x , int y , int val){
    int y1;
    while (x <= max_x){
        y1 = y;
        while (y1 <= max_y){
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}

int getSum(int BIT[][N+1], int x, int y)
{
    int sum = 0;

    for(; x > 0; x -= x&-x)
    {
        // This loop sum through all the 1D BIT
        // inside the array of 1D BIT = BIT[x]
        for(; y > 0; y -= y&-y)
        {
            sum += BIT[x][y];
        }
    }
    return sum;
}
```

5.3 Ordered Statistics

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template <typename T> using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
```

```
// typedef
// tree<
//   pair<int,int>,
//   null_type,
//   less<pair<int,int>>,
//   rb_tree_tag,
//   tree_order_statistics_node_update>
// ordered_set;

ordered_set t;
int x,y;
for(int i=0;i<n;i++)
{
    cin>>x>>y;
    ans[t.order_of_key({x,++sz})]++;
    t.insert({x,sz});
}
// If we want to get map but not the set, as the second argument type must be used mapped type. Apparently, the tree supports the same operations as the set (at least I haven't any problems with them before), but also there are two new features     it is find_by_order() and order_of_key(). The first returns an iterator to the k-th largest element (counting from zero), the second     the number of items in a set that are strictly smaller than our item. Example of use:
//      ordered_set X;
//      X.insert(1);
//      X.insert(2);
//      X.insert(4);
//      X.insert(8);
//      X.insert(16);
//      cout<<*X.find_by_order(1)<<endl; // 2
//      cout<<*X.find_by_order(2)<<endl; // 4
//      cout<<*X.find_by_order(4)<<endl; // 16
//      cout<<(end(X)==X.find_by_order(6))<<endl; // true
//      cout<<X.order_of_key(-5)<<endl;  // 0
//      cout<<X.order_of_key(1)<<endl;   // 0
//      cout<<X.order_of_key(3)<<endl;   // 2
//      cout<<X.order_of_key(4)<<endl;   // 2
//      cout<<X.order_of_key(400)<<endl; // 5
```

5.4 Persistent Tree

```
struct node{
 int coun;
 node *l,*r ;
 node(int coun,node *l,node *r):
  coun(coun),l(l),r(r){}
  node *inser(int l,int r,int pos) ;
};
node* node::inser(int l,int r,int pos){
 if(l<=pos && pos<=r){
  if(l==r){
   return new node(this->coun+1,NULL,NULL);
  }
  int mid=(l+r)>>1;
  return new node(this->coun+1,this->l->inser(l,mid,pos),this->r->inser(mid+1,r,pos));
 }
 return this;
}
int query(node *lef,node *rig,int cc,int s,int e){
 if(s==e)
   return s;
 int co=rig->l->coun-lef->l->coun;
 int mid=(s+e)>>1;
 if(co>=cc)
  return query(lef->l,rig->l,cc,s,mid);
 return query(lef->r,rig->r,cc-co,mid+1,e);
}
node *null=new node(0,NULL,NULL);
node *root[100100];
null->l=null->r=null;
root[0]=null;
for(int i=1;i<=n;i++)
 root[i]=root[i-1]->inser(0,maxy,m[arr[i]]);
while(mmm-->0){
 int i,j,k;cin>>i>>j>>k;
 cout<<mm[query(root[i-1],root[j],k,0,maxy)]<<"\n";
```

6 Math

6.1 Convex Hull

```
struct point{
 int x, y;
 point(int _x = 0, int _y = 0){
  x = _x, y = _y;
 }
```

```cpp
 friend bool operator < (point a, point b){
  return (a.x == b.x) ? (a.y < b.y) : (a.x < b.x);
 }
};
point pt[2*Max], hull[2*Max];
//Here idx is the new length of the hull
int idx=0,cur;
inline long area(point a, point b, point c){
 return (b.x - a.x) * 1LL * (c.y - a.y) - (b.y - a.y) * 1LL * (c.x - a.x);
}
inline long dist(point a, point b){
 return (a.x - b.x) * 1LL * (a.x - b.x) + (a.y - b.y) * 1LL * (a.y - b.y);
}
inline bool is_right(point a, point b){
 int dx = (b.x - a.x);
 int dy = (b.y - a.y);
 return (dx > 0) || (dx == 0 && dy > 0);
}
inline bool compare(point b, point c){
 long det = area(pt[1], b, c);
 if(det == 0){

  if(is_right(pt[1], b) != is_right(pt[1], c))
   return is_right(pt[1], b);
  return (dist(pt[1], b) < dist(pt[1], c));
 }
 return (det > 0);
}
void convexHull(){
 int min_x = pt[1].x, min_y = pt[1].y, min_idx = 1;
 for(int i = 2; i <= cur; i++){
  if(pt[i].y < min_y || (pt[i].y == min_y && pt[i].x < min_x)){
   min_x = pt[i].x;
   min_y = pt[i].y;
   min_idx = i;
  }
 }
 swap(pt[1], pt[min_idx]);
 sort(pt + 2, pt + 1 + cur, compare);
 idx = 2;
 hull[1] = pt[1], hull[2] = pt[2];
 for(int i = 3 ; i <= cur ; i++){
  while(idx>=2 && (area(hull[idx - 1], hull[idx], pt[i]) <= 0)) idx-- ;
  hull[++idx] = pt[i] ;
 }
}
```

6.2 FFT

```cpp
const long mod = 5 * (1 << 25) + 1;
long root = 243;
long root_1 = 114609789;
const long root_pw = 1 << 25;

inline void fft (vector < long > & a, bool invert) {
    int n = (int) a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j >= bit; bit >>= 1) {
            j -= bit;
        }
        j += bit;
        if (i < j) {
          swap(a[i], a[j]);
        }
    }
    for (int len = 2; len <= n; len <<= 1) {
        long wlen = invert ? root_1 : root;
        for (long i = len; i < root_pw; i <<= 1)
            wlen = (long) (wlen * 1ll * wlen % mod);
        for (int i = 0; i < n; i += len) {
            long w = 1;
            for (int j = 0; j < len / 2; j++) {
                long u = a[i + j];
                long v = (long) (a[i + j + len / 2] * 1ll * w % mod);
                a[i + j] = u + v < mod ? u + v : u + v - mod;
                a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + mod;
                w = (long) (w * 1ll * wlen % mod);
            }
        }
    }
    if (invert) {
        long nrev = exp(n, mod-2);
        for (int i = 0; i < n; i++)
          a[i] = (long) (a[i] * 1ll * nrev % mod);
    }
}
```

6.3 FFT_Complex

```
// Instructions for using this: Nothing it is very ↩
   obvious to use

typedef complex<long double> Cld;
class FFT{
public:
 static const ld PI;
 static void cfft (vector<Cld> &L,int invert) {
  int n = (int) L.size();
  for(int i=1,j=0 ; i<n ; i++){
   int bit = n>>1 ;
   for( ; j>=bit ; bit>>=1) j-=bit ;
   j+=bit ;
   if(i<j) swap(L[i],L[j]);
  }
  for(int len=2 ; len<=n ; len<<=1){
   int l2 = (len/2);
   ld theta = (PI/l2);
   Cld wlen = polar(1.0L,(invert ? -1 : 1)*theta);
   for(int i=0 ; i<n ; i+=len){
    Cld w(1.0,0.0);
    for(int j=0 ; j<l2 ; j++, w=(w*wlen)){
     Cld u = L[i+j]; Cld v = w*L[i+j+l2];
     L[i+j] = (u+v); L[i+j+l2] = (u-v);
    }
   }
  }
  if(invert)
   for(int i=0 ; i<n ; i++) L[i] = L[i]/((ld) n);
 }
};
const ld FFT::PI = acos(-1.0);
```

6.4 Find Primitive Root

```
vector < lli > factorize(lli x) {
    // Returns prime factors of x
    vector < lli > primes;
    for (lli i = 2; i * i <= x; i++) {
        if (x % i == 0) {
            primes.push_back(i);
            while (x % i == 0) {
                x /= i;
            }
        }
    }
    if (x != 1) {
        primes.push_back(x);
    }
    return primes;
}

inline bool test_primitive_root(lli a, lli m) {
    // Is 'a' a primitive root of modulus 'm'?
    // m must be of the form 2^k * x + 1
    lli exp = m - 1;
    lli val = power(a, exp, m);
    if (val != 1) {
        return false;
    }
    vector < lli > factors = factorize(exp);
    for (lli f: factors) {
        lli cur = exp / f;
        val = power(a, cur, m);
        if (val == 1) {
            return false;
        }
    }
    return true;
}

inline lli find_primitive_root(lli m) {
    // Find primitive root of the modulus 'm'.
    // m must be of the form 2^k * x + 1
    for (lli i = 2; ; i++) {
        if (test_primitive_root(i, m)) {
            return i;
        }
    }
}
```

6.5 Convex Hull Trick

```
mylist hull(mylist pts){
 int n = pts.size() ;
 if(n<2) return pts ;
 Collections.sort(pts,new Comparator<pair>(){
  public int compare(pair p1,pair p2){
   if(p1.x!=p2.x) return Double.compare(p1.x,p2.x) ↩
   ;
   return Double.compare(p2.y,p1.y) ;
  }
 }) ;
 mylist h = new mylist() ;
 h.add(pts.get(0)) ; h.add(pts.get(1)) ;
```

```
 int idx=1 ;
 for(int i=2 ; i<n ; i++){
  pair p = pts.get(i) ;
  while(idx>0){
   if(isOriented(h.get(idx-1),h.get(idx),p))
    break ;
   else
    h.remove(idx--) ;
  }
  h.add(p) ;
  idx++ ;
 }
 while(idx>0 && h.get(idx).x==h.get(idx-1).x) h.remove(idx--) ;
 Collections.reverse(h) ;
 return h ;
}
public boolean isOriented(pair p1,pair p2,pair p3){
 double val = ((p2.y-p1.y)*(p3.x-p2.x))-((p2.x-p1.x)*(p3.y-p2.y)) ;
 return val>=0 ;
}
```

6.6 Miscellaneous Geometry

```
const ld EPS = 1e-12;
struct PT{
 ld x,y,z;
 PT(ld x=0,ld y=0,ld z=0): x(x),y(y),z(z){}
 bool operator<(const PT &t){ return make_tuple(x,y,z)<make_tuple(t.x,t.y,t.z); }
 bool operator==(const PT &t){ return make_tuple(x,y,z)==make_tuple(t.x,t.y,t.z); }
 PT operator+(const PT &t){ return PT(x+t.x,y+t.y,z+t.z); }
 PT operator-(const PT &t){ return PT(x-t.x,y-t.y,z-t.z); }
 PT operator*(const ld &d){ return PT(x*d,y*d,z*d); }
 PT operator/(const ld &d){ return PT(x/d,y/d,z/d); }
 ld norm2(){ return (x*x + y*y + z*z); }
 ld norm(){ return sqrtl(norm2()); }
};
PT cross(const PT &p,const PT &q){
 return PT(p.y*q.z - p.z*q.y, p.z*q.x - p.x*q.z, p.x*q.y - p.y*q.x);
}
ld dot(const PT &p, const PT &q){
 return (p.x*q.x + p.y*q.y + p.z*q.z);
}
// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p){ return PT(-p.y,p.x); }
PT RotateCW90(PT p){ return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
 return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}
// project point c onto line segment through a and b
// if the projection doesn't lie on the segment, returns closest vertex
PT ProjectPointSegment(PT a, PT b, PT c) {
 double r = dot(b-a,b-a);
 if(fabs(r)<EPS) return a;
 r = dot(c-a,b-a)/r;
 if(r<0) return a;
 if(r>1) return b;
 return a+(b-a)*r;
}
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
 return a + (b-a)*dot(c-a,b-a)/dot(b-a, b-a);
}
// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
 return fabs(cross(b-a, c-d)) < EPS;
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
 return LinesParallel(a, b, c, d)
 && fabs(cross(a-b, a-c)) < EPS
    && fabs(cross(c-d, c-a)) < EPS;
}
// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
 if (LinesCollinear(a, b, c, d)) {
  if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
   dist2(b, c) < EPS || dist2(b, d) < EPS)
   return true;
  if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
   return false;
  return true;
 }
 if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
```

```
 if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return ↩
    false;
 return true;
}
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) ↩
    {
 b=b-a; d=c-d; c=c-a;
 assert(b.norm() > EPS && d.norm() > EPS);
 return (a + b*cross(c, d)/cross(b, d));
}
// determine if c and d are on same side of line ↩
    passing through a and b
bool OnSameSide(PT a, PT b, PT c, PT d) {
  return (cross(c-a, c-b)*cross(d-a, d-b))>0;
}
PT ComputeCircleCenter(PT a, PT b, PT c) {
 b=(a+b)/2;
 c=(a+c)/2;
 return ComputeLineIntersection(b, b+RotateCW90(a-b↩
    ), c, c+RotateCW90(a-c));
}
vector<PT> CircleCircleIntersection(PT a, PT b, ld ↩
    r, ld R) {
 vector<PT> ret;
 ld d = (a-b).norm();
 if (d>(r+R) || d+min(r,R) < max(r,R)) return ret;
 ld x = (d*d-R*R+r*r)/(2*d);
 ld y = sqrtl(r*r-x*x);
 PT v = (b-a)/d;
 ret.push_back(a+v*x + RotateCCW90(v)*y);
 if(y>0) ret.push_back(a+v*x - RotateCCW90(v)*y);
 return ret;
}
ld ComputeSignedArea(const vector<PT> &p) {
 ld area = 0;
 int n = p.size();
 for(int i=0 ; i<n ; i++)
  area += cross(p[i],p[(i+1)%n]);
 return area/2.0;
}
ld ComputeArea(const vector<PT> &p) {
 return fabs(ComputeSignedArea(p));
}
bool IsSimple(const vector<PT> &p) {
 for (int i = 0; i < p.size(); i++) {
  for (int k = i+1; k < p.size(); k++) {
   int j = (i+1) % p.size(); int l = (k+1) % p.size↩
      ();
   if (i == l || j == k) continue;
   if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
    return false;
  }
 }
 return true;
}
// determine if point is in a possibly non-convex ↩
   polygon (by William
// Randolph Franklin); returns 1 for strictly ↩
   interior points, 0 for
// strictly exterior points, and 0 or 1 for the ↩
   remaining points.
// Note that it is possible to convert this into an↩
    *exact* test using
// integer arithmetic by taking care of the ↩
   division appropriately
// (making sure to deal with signs properly) and ↩
   then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
 bool c = false;
 for (int i = 0; i < p.size(); i++){
 int j = (i+1)%p.size();
 bool test1 = (p[i].y <= q.y && q.y < p[j].y || p[j↩
    ].y <= q.y && q.y < p[i].y);
 bool test2 = q.x < (p[i].x + (p[j].x - p[i].x)*((q↩
    .y - p[i].y)/(p[j].y - p[i].y)));
 if(test1 && test2) c = !c;
 return c;
}
// determine if point is on the boundary of a ↩
   polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
 for (int i = 0; i < p.size(); i++)
  if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.↩
     size()], q), q) < EPS)
   return true;
 return false;
}
struct Line{
 ld a,b,c;
 Line(ld a=0,ld b=0,ld c=0): a(a),b(b),c(c){}
};
pdd LineIntersection(const Line &l1,const Line &l2)↩
   {
 ld a1 = l1.a; ld b1 = l1.b; ld c1 = l1.c;
 ld a2 = l2.a; ld b2 = l2.b; ld c2 = l2.c;
 ld det = (a1*b2 - a2*b1);
 assert(abs(det)>eps);
 ld x = (b1*c2 - b2*c1)/det;
 ld y = (c1*a2 - c2*a1)/det;
 return mp(x,y);
}
```

6.7 Gaussian elimination for square matrices of full rank; finds inverses and determinants

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

const double EPS = 1e-10;

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

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

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

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

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

  return det;
}
```

7 Number Theory Reference

7.1 Modular arithmetic and linear Diophantine solver

```
// This is a collection of useful code for solving ↩
   problems that
// involve modular linear equations.  Note that all↩
    of the
// algorithms described here work on nonnegative ↩
   integers.

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

// return a % b (positive value)
int mod(int a, int b) {
  return ((a%b)+b)%b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
```

```
}

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

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

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

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

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

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

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

7.2 Polynomial Coefficients (Text)

$(x_1 + x_2 + ... + x_k)^n = \sum_{c_1+c_2+...+c_k=n} \frac{n!}{c_1!c_2!...c_k!} x_1^{c_1} x_2^{c_2} ... x_k^{c_k}$

7.3 Möbius Function (Text)

$\mu(n) = \begin{cases} 0 & n \text{ not squarefree} \\ 1 & n \text{ squarefree w/ even no. of prime factors} \\ 1 & n \text{ squarefree w/ odd no. of prime factors} \end{cases}$ Note that $\mu(a)\mu(b) = \mu(ab)$ for a, b relatively prime Also $\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$

Möbius Inversion If $g(n) = \sum_{d|n} f(d)$ for all $n \geq 1$, then $f(n) = \sum_{d|n} \mu(d) g(n/d)$ for all $n \geq 1$.

7.4 Burnside's Lemma (Text)

The number of orbits of a set X under the group action G equals the average number of elements of X fixed by the elements of G.

Here's an example. Consider a square of $2n$ times $2n$ cells. How many ways are there to color it into X colors, up to rotations and/or reflections? Here, the group has only 8 elements (rotations by 0, 90, 180 and 270 degrees, reflections over two diagonals, over a vertical line and over a horizontal line). Every coloring stays itself after rotating by 0 degrees, so that rotation has X^{4n^2} fixed points. Rotation by 180 degrees and reflections over a horizonal/vertical line split all cells in pairs that must be of the same color for a coloring to be unaffected by such rotation/reflection, thus there exist X^{2n^2} such colorings for each of them. Rotations by 90 and 270 degrees split cells in groups of four, thus yielding X^{n^2} fixed colorings. Reflections over diagonals split cells into $2n$ groups of 1 (the diagonal itself) and $2n^2 - n$ groups of 2 (all remaining cells), thus yielding $X^{2n^2-n+2n} = X^{2n^2+n}$ unaffected colorings. So, the answer is $(X^{4n^2} + 3X^{2n^2} + 2X^{n^2} + 2X^{2n^2+n})/8$.

8 Miscellaneous

8.1 2-SAT

```
// 2-SAT solver based on Kosaraju's algorithm.
// Variables are 0-based. Positive variables are stored in vertices 2n, corresponding negative variables in 2n+1
// TODO: This is quite slow (3x-4x slower than Gabow's algorithm)
struct TwoSat {
 int n;
 vector<vector<int> > adj, radj, scc;
 vector<int> sid, vis, val;
 stack<int> stk;
 int scnt;

 // n: number of variables, including negations
 TwoSat(int n): n(n), adj(n), radj(n), sid(n), vis(n), val(n, -1) {}

 // adds an implication
 void impl(int x, int y) { adj[x].push_back(y); radj[y].push_back(x); }
 // adds a disjunction
 void vee(int x, int y) { impl(x^1, y); impl(y^1, x); }
 // forces variables to be equal
 void eq(int x, int y) { impl(x, y); impl(y, x); impl(x^1, y^1); impl(y^1, x^1); }
 // forces variable to be true
 void tru(int x) { impl(x^1, x); }

 void dfs1(int x) {
  if (vis[x]++) return;
  for (int i = 0; i < adj[x].size(); i++) {
   dfs1(adj[x][i]);
  }
  stk.push(x);
 }

 void dfs2(int x) {
  if (!vis[x]) return; vis[x] = 0;
  sid[x] = scnt; scc.back().push_back(x);
  for (int i = 0; i < radj[x].size(); i++) {
   dfs2(radj[x][i]);
  }
 }

 // returns true if satisfiable, false otherwise
```

```
  // on completion, val[x] is the assigned value of ↩
     variable x
  // note, val[x] = 0 implies val[x^1] = 1
  bool two_sat() {
   scnt = 0;
   for (int i = 0; i < n; i++) {
    dfs1(i);
   }
   while (!stk.empty()) {
    int v = stk.top(); stk.pop();
    if (vis[v]) {
     scc.push_back(vector<int>());
     dfs2(v);
     scnt++;
    }
   }
   for (int i = 0; i < n; i += 2) {
    if (sid[i] == sid[i+1]) return false;
   }
   vector<int> must(scnt);
   for (int i = 0; i < scnt; i++) {
    for (int j = 0; j < scc[i].size(); j++) {
     val[scc[i][j]] = must[i];
     must[sid[scc[i][j]^1]] = !must[i];
    }
   }
   return true;
  }
};
```