from __future__ import solution

Northwestern Europe Regional Contest

2018-11-25

Contents

1. Achieving AC on solved problems

1.1. WA.

- Edge cases (minimal input, etc)?
- Does test input pass?
- Is overflow possible?
- Is it possible that we solved the wrong problem? Reread the problem statement!
- Start creating small test cases.
- Does cout print with sufficient percision?
- Abstract the implementation!

1.2. TLE.

- Is the solution sanity checked? (Time complexitity)
- Use pypy / rewrite in C++ and Java
- Can DP be applied in some part?
- Try creating worst case input to see how far away the solution is
- Binary search instead of exhaustive search
- Binary search over the answer

1.3. RTE.

- Recursion limit in python?
- Out of bounds?
- Division by 0?
- Modifying something we iterate over?
- Not using well defined sorting?
- If nothing makes sense, try binary search with try-catch/except.

1.4. **MLE.**

- Create objects outside recursive function
- Rewrite recursive solution to iterative

2. Templates, etc.

2.1. **C++.**

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define rep(i,a,b) for (ll i = a; i<ll(b); i++)
//compile with g++/cc -g -Wall -Wconversion -fsanitize=address,
//undefined <filename.cpp>
int main() {
  ios::sync_with_stdio(false);
  cout << setprecision(10);
}
// Reads in an unknown number of rows with unknown number of words
string line;
string word;
while (getline(cin, line)){
  stringstream ss(line);
  while(getline(ss, word, ' ')){
    cout << word << endl;
  }
  cout << "________________________" << endl;
}
//Reads ints until end of file
int k;
while (cin >> k){
   cout << k << endl;
}
```

2.2. **Python.**

```
from collections import deque
q = deque([0])      # initiates a queue
q.popleft()         # pops the first element
q.append(0)         # pushes a element to end of queue

import sys
sys.setrecursionlimit(1000000) # default is 1000.

from itertools import permutations, combinations, product
a = 'ABCD'
premutations(a,2) == ['AB','AC','AD','BA','BC','BD',
        'CA','CB','CD','DA','DB','DC']
combinations(a,2) == ['AB','AC','AD','BC','BD','CD']
combinations_with_replacement(a,2) == \
        ['AA','AB','AC','AD','BB','BC','BD','CC','CD','DD']
product(a,2) == ['AA','AB','AC','AD','BA','BB','BC','BD',
        'CA','CB','CC','CD','DA','DB','DC','DD']

#If a specified output, o, should be outputed with x decimals:
print '\%.xf' % o
print '{0:.2f}'.format(o)
#For example
print '\%.4f' % 2.05
print '\%.4f' % 3.1415926535
print '{0:.2f}'.format(3.1415926535)
#gives us 2.0500, 3.1416
```

2.3. **Bash.** Shell script to run all samples from a folder on a problem

```
#!/bin/bash
# make exacutable: chmod +x run.sh
# run: ./run.sh A pypy A.py
# or
# ./run.sh A ./a.out
folder=$1;shift
for f in $folder/*.in; do
    echo $f
    pre=${f%.in}
    out=$pre.out
    ans=$pre.ans
    $* < $f > $out
    diff $out $ans
done
```

3. Data Structures

3.1. **Fenwick Tree.**

```python
'''
Constructs a fenwicktree of an array. Can update a bit and get the
sum up to and including i in the array.

Time Complexity: O(NlogN) for construction, O(logN) for update and query.
SpaceComplexity: O(N)
'''
def fenwicktree(arr):
    fwtree = [0]*(len(arr)+1)
    for i in range(len(arr)):
        updatebit(fwtree,len(arr),i,arr[i])
    return fwtree

def updatebit(fwtree,i,val):
    i += 1
    while i < len(fwtree):
        fwtree[i] += val
        i += i&(-i)

# get sum of [0,i] inclusive
def getsum(fwtree,i):
    s = 0
    i += 1
    while i > 0:
        s += fwtree[i]
        i -= i&(-i)
    return s
```

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

class fwtree {
public:
  vector<ll> tree;
  ll n;
  fwtree(ll N) {
    n=N;
    tree.assign(n+1,0);
  }
  void update(ll ind, ll val) {
    ind++;
    while(ind <= n){
        tree[ind] += val;
        ind += ind&(-ind);
    }
  }
  ll que(ll ind) {
    ll ret = 0;
    ind++;
    while(ind > 0){
        ret += tree[ind];
        ind -= ind&(-ind);
    }
    return ret;
  }
};
```

3.2. **Segment Tree.**

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
/*
O(n) creation, O(log n) update/query
Queries are inclusive [L,R]
*/
class sgmtree {
public:
  vector<ll> vals;
  vector<ll> tree;
  ll n;
  sgmtree(vector<ll> x) {
    vals=x;
    n=x.size();
    tree.assign(4*n+4,0);
    build(1,0,n-1);
  }
  ll que(ll L, ll R) {
    return que(1,0,n-1,L,R);
  }
  void update(ll ind, ll val) {
```

```
    vals[ind]=val;
    update(1,0,n-1,ind);
  }
private:
  ll I = 0; // I
  void build(ll node, ll l, ll r) {
    if (l==r) {tree[node]=vals[l]; return;}
    ll mid=(l+r)/2;
    build(2*node,l,mid);
    build(2*node+1,mid+1,r);
    tree[node]=tree[2*node]+tree[2*node+1]; // op
  }
  ll que(ll node, ll l, ll r, ll L, ll R) {
    if (l>R || r<L) return I; // I
    if (l>=L && r<=R) return tree[node];
    ll mid=(l+r)/2;
    return que(2*node,l,mid,L,R)+que(2*node+1,mid+1,r,L,R); // op
  }
  void update(ll node, ll l, ll r, ll ind) {
    if (l==r && l==ind) {tree[node]=vals[ind]; return;}
    if (l>ind || r<ind) return;
    ll mid=(l+r)/2;
    update(2*node,l,mid,ind);
    update(2*node+1,mid+1,r,ind);
    tree[node]=tree[2*node]+tree[2*node+1]; // Op
  }
};
```

3.3. Lazy Setting Segment Tree.

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define rep(i,a,b) for (ll i = a; i<ll(b); i++)
//This is a lazy sgmtree, but updates doesnt inc,
//update sets all values in segment
class sgmtree {
public:
  vector<ll> vals;
  vector<ll> tree;
  vector<ll> lazyupdts;
  ll n;
  sgmtree(vector<ll> x) {
    vals=x;
    n=x.size();
    tree.assign(4*n+4,0);
    lazyupdts.assign(4*n+4,-1);
    build(1,0,n-1);
  }
  ll que(ll L, ll R) {
    return que(1,0,n-1,L,R);
  }
  void update(ll L, ll R, ll val) {
    //vals[ind]=val; //Set value val for all nodes L to R
    update(1,0,n-1,L,R,val);
  }
private:
  ll I = -9999999; // I
  void build(ll node, ll l, ll r) {
    if (l==r) {tree[node]=vals[l]; return;}
    ll mid=(l+r)/2;
    build(2*node,l,mid);
    build(2*node+1,mid+1,r);
    tree[node]=max(tree[2*node],tree[2*node+1]); // op
  }
  ll que(ll node, ll l, ll r, ll L, ll R) {
    if (l>R || r<L) return I; // I
    if (l>=L && r<=R) return tree[node];
    ll mid=(l+r)/2;
    if (lazyupdts[node]!=-1) {
      update(node*2,l,mid,l,mid,lazyupdts[node]);
      update(2*node+1,mid+1,r,mid+1,r,lazyupdts[node]);
      lazyupdts[node]=-1;
    }
    return max(que(2*node,l,mid,L,R),que(2*node+1,mid+1,r,L,R)); // op
  }
  void update(ll node, ll l, ll r, ll L, ll R, ll val) {
    if (l>R || r<L) return;
    if (l>=L && r<=R) {
      //Lazy update this
      tree[node]=val; //Op
```

```
      if (l==r) {return;}
      lazyupdts[node]=val;
      return;
    }
    //if (l==r && l==ind) {tree[node]=vals; return;}
    //if (l>ind || r<ind) return;
    ll mid=(l+r)/2;
    if (lazyupdts[node]!=-1) { //propagate down current lazyvalues
      update(2*node,l,mid,l,mid,lazyupdts[node]);
      update(2*node+1,mid+1,r,mid+1,r,lazyupdts[node]);
      lazyupdts[node]=-1;
    }
    update(2*node,l,mid,L,R,val);
    update(2*node+1,mid+1,r,L,R,val);
    tree[node]=max(tree[2*node],tree[2*node+1]); // Op
  }
};
```

3.4. **Lazy Incrementing Segment Tree.**

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define rep(i,a,b) for (ll i = a; i<ll(b); i++)
//This is a lazy sgmtree, update query increments
//all values between L and R
class sgmtree {
public:
  vector<ll> vals;
  vector<ll> tree;
  vector<ll> lazyupdts;
  ll n;
  sgmtree(vector<ll> x) {
    vals=x;
    n=x.size();
    tree.assign(4*n+4,0);
    lazyupdts.assign(4*n+4,0);
    build(1,0,n-1);
  }
  ll que(ll L, ll R) {
    return que(1,0,n-1,L,R);
  }
  void update(ll L, ll R, ll val) {
    //Inc with val for all nodes L to R
    update(1,0,n-1,L,R,val);
  }
private:
  ll I = -9999999; // I
  void build(ll node, ll l, ll r) {
    if (l==r) {tree[node]=vals[l]; return;}
    ll mid=(l+r)/2;
    build(2*node,l,mid);
    build(2*node+1,mid+1,r);
    tree[node]=max(tree[2*node],tree[2*node+1]); // op
  }
  ll que(ll node, ll l, ll r, ll L, ll R) {
    if (l>R || r<L) return I; // I
    if (l>=L && r<=R) return tree[node];
    ll mid=(l+r)/2;
    if (lazyupdts[node]!=0) {
      update(node*2,l,mid,l,mid,lazyupdts[node]);
      update(2*node+1,mid+1,r,mid+1,r,lazyupdts[node]);
      lazyupdts[node]=0;
    }
    return max(que(2*node,l,mid,L,R),que(2*node+1,mid+1,r,L,R)); // op
  }
  void update(ll node, ll l, ll r, ll L, ll R, ll val) {
    if (l>R || r<L) return;
    if (l>=L && r<=R) {
      //Lazy update this
      tree[node]+=val; //Op
      if (l==r) {return;}
      lazyupdts[node]+=val;
      return;
    }
    //if (l==r && l==ind) {tree[node]=vals; return;}
    //if (l>ind || r<ind) return;
    ll mid=(l+r)/2;
    if (lazyupdts[node]!=0) { //propagate down current lazyvalues
      update(2*node,l,mid,l,mid,lazyupdts[node]);
      update(2*node+1,mid+1,r,mid+1,r,lazyupdts[node]);
```

```
      lazyupdts[node]=0;
    }
    update(2*node,l,mid,L,R,val);
    update(2*node+1,mid+1,r,L,R,val);
    tree[node]=max(tree[2*node],tree[2*node+1]); // Op
  }
};
```

3.5. Union Find.

```python
'''
All roots stored in roots, depth of each tree stored in depth.
Both roots and depth can be either a list or a dict.

Time Complexity: O(logN) for both find and union, where N is the
                    number of objects in the structure
Space Complexity: O(N)
'''

#Finds root in the tree containing n.
def find(n):
    if roots[n] != n: roots[n] = find(roots[n])
    return roots[n]

#Unions the trees containing n and m. Returns true if the nodes
#are in different trees, otherwise false.
def union(n,m):
    pn = find(n)
    pm = find(m)
    if pn == pm: return False
    if depth[pn] < depth[pm]: roots[pn] = pm
    elif depth[pm] < depth[pn]: roots[pm] = pn
    else:
        roots[pn] = pm
        depth[pm] += 1
    return True
```

3.6. Monotone Queue.

```python
'''
Keeps a monotone queue (always increasing or decreasing).
This is good for solving "What is the smallest (largest)
element in the window of size L in an array. This is done
by in each step calling add and remove on the monotone queue
and also looking at the smallest (largest) element which
is at position 0.

Time-Complexity: O(n), n is the size of the array.
Space-Complexity: O(n).
'''
from collections import deque
def minadd(mminque,x):
    while mminque and x < mminque[-1]:
        mminque.pop()
    mminque.append(x)

def minremove(mminque,x):
    if mminque[0] == x:
        mminque.popleft()

def maxadd(mmaxque,x):
    while mmaxque and x > mmaxque[-1]:
        mmaxque.pop()
    mmaxque.append(x)

def maxremove(mmaxque,x):
    if mmaxque[0] == x:
        mmaxque.popleft()
```

3.7. Treap.

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
constexpr ll INF = 9999999999999;

class Treap {
public:
  ll prio, val,size;
  Treap *l, *r;
  Treap(ll v) {
    val=v;
    l=NULL;
```

```
    r=NULL;
    size=1;
    prio=(ll) rand();
  }
  void update() {
    size=1;
    if (l!=NULL) size += l->size;
    if (r!=NULL) size += r->size;
  }
  void print(){
    cout << "________________" << endl;
    Hprint();
    cout << "________________" << endl;
  }
  void Hprint() {
    if (l!=NULL) l->Hprint();
    cout << val << " " << prio << endl;
    if(r != NULL) r->Hprint();
  }
};

//Split on index
pair<Treap*, Treap*> splitIndex(Treap *cur, ll i) {
  if (i > cur->size) assert(false);
  Treap *left = cur->l;
  Treap *right = cur ->r;
  ll lsize = left != NULL ? left->size : 0L;
  if (lsize == i){
    cur->l = NULL;
    cur->update();
    return {left, cur};
  }
  if (lsize +1 == i) {
    cur->r = NULL;
    cur->update();
    return {cur,right};
  }
  if (lsize > i){
    auto p = splitIndex(left, i);
    cur->l = p.second;
    cur->update();
    return {p.first, cur};
  }
  auto p = splitIndex(right, i - lsize - 1);
  cur->r = p.first;
  cur->update();
  return {cur, p.second};
}
//Split on value
pair<Treap*, Treap*> split(Treap *cur, ll val){
  Treap *left = cur->l;
  Treap *right = cur ->r;
  if (cur->val >= val){
    if (left == NULL) return {NULL, cur};
    auto p = split(left, val);
    cur->l = p.second;
    cur->update();
    return {p.first, cur};
  }
  if (cur->val < val){
    if (right == NULL) return {cur, NULL};
    auto p = split(right, val);
    cur->r = p.first;
    cur->update();
    return {cur, p.second};
  }
}

Treap* meld(Treap *a, Treap *b) { // all in b is bigger than a
  if (a==NULL) return b;
  if (b == NULL) return a;
  if (a->prio < b->prio) { //a root
    a->r = (a->r == NULL) ? b : meld(a->r, b);
    a->update();
    return a;
  }
  //b root
  b->l = (b->l == NULL) ? a : meld(a, b->l);
  b->update();
  return b;
```

```
}

Treap* insert(Treap* a, ll val){
  if (a==NULL) return new Treap(val);
  auto p = split(a, val);
  Treap *t = new Treap(val);
  return meld(p.first, meld(t, p.second));
}

Treap* del(Treap *root, ll val) {
  pair<Treap*,Treap*> saker1 = split(root,val);
  if (saker1.second == NULL) return saker1.first;
  pair<Treap*,Treap*> saker2 = split(saker1.second,val+1);
  return meld(saker1.first,saker2.second);
}

pair<bool, Treap*> exists(Treap *root, ll val) {
  pair<Treap*,Treap*> firstSplit = split(root,val);
  if (firstSplit.second == NULL) return {false, firstSplit.first};
  pair<Treap*,Treap*> secondSplit = split(firstSplit.second,val+1);
  return {secondSplit.first != NULL,meld(firstSplit.first,
          meld(secondSplit.first,secondSplit.second))};
}

ll next(Treap *root, ll val){
  if(root == NULL) return INF;
  if(val >= root->val) return next(root->r,val);
  return min(root->val,next(root->l,val));
}

ll prev(Treap *root, ll val){
  if(root == NULL) return -INF;
  if(val > root->val) return max(root->val,prev(root->r,val));
  return prev(root->l,val);
}
```

4. Graph Algorithms

4.1. Distance from source to all nodes (pos weights) - Djikstra's algorithm.

```python
'''
Implementation of djikstras algorithm. Finds the shortest path from a
source, to all other nodes (non-negative weights).
adj is a list of adjacency lists and s the source node.

Time Complexity: O(M + NlogN), where N is the number of nodes, M edges.
Space Complexity: O(M + N)
'''
from heapq import heappush, heappop

INF = 10**12
def djikstra(adj,S):
    N = len(adj)
    d = [INF]*N
    vis = [False]*N
    d[S],vis[S] = 0,True
    pq = []
    heappush(pq, (d[S],S))
    while pq:
        curD, curN = heappop(pq)
        if vis[curN]: continue
        vis[curN] = True
        for ne,w in adj[curN]:
            altD = curD + w
            if altD < d[ne]:
                heappush(pq,(altD,ne))
                d[ne] = altD
    return d
```

4.2. Distance from source to all nodes (neg weights) - Bellman Ford.

```python
'''
Calculates the distance from a source to all other nodes.
Run this by putting edgs as a list of tuples (u,v,w) where
the edge goes from u to v with weight w (w might be negative).
Time Complexity: O(N*M), N #nodes, M #edges
Space Complexity: O(M)
'''
def bfs(cur):
    vis = [False]*n
    b = [cur]
```

```python
    vis[cur] = True
    while b:
        c = b.pop()
        dists[c] = '-Infinity'
        for ne in adj[c]:
            if not vis[ne]:
                vis[ne] = True
                b.append(ne)

def bellmanford(edgs,s):
    dists[s] = 0
    for i in range(n-1):
        for edg in edgs:
            u,v,w = edg
            if dists[u] + w < dists[v]: dists[v] = dists[u] + w
    for edg in edgs:
        u,v,w = edg
        if dists[v] == '-Infinity': continue
        if dists[u] + w < dists[v] and dists[v] < INF/2: bfs(v)
    for i in range(n):
        if dists[i] > INF/2 and dists[i] != '-Infinity':
            dists[i] = 'Impossible'
    return dists
```

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

ll INF = 1e18;

struct Edge{
    int from,to;
    ll d;
};

class BellmanFord{
public:
    vector<ll> dists;
    int N;
    vector<vector<int> > adj;
    vector<Edge> edgs;
    BellmanFord(int N){
        this->N = N;
        adj.assign(N,vector<int>());
        dists.assign(N,INF);
    }
    //Edges are directed.
    void addEdge(int from, int to, ll d){
        adj[from].push_back(to);
        edgs.push_back({from,to,d});
    }
    void bellmanFord(int s){
        dists[s] = 0;
        for(int i = 0; i < N-1; i++){
            for(auto e : edgs){
                int u = e.from, v = e.to; ll w = e.d;
                if(dists[u] + w < dists[v]) dists[v] = dists[u]+w;
            }
        }
        //Skip if no negative cycles are guaranteed.
        for(auto e : edgs){
            int u = e.from, v = e.to; ll w = e.d;
            if(dists[v] == -INF) continue;
            if(dists[u] + w < dists[v] && dists[v] < INF/2) bfs(v);
        }
        for(int i = 0; i < N; i++){
            if(dists[i] > INF/2) dists[i] = INF;
        }
    }
    //Skip if no negative cycles are guaranteed.
    void bfs(int cur){
        vector<bool> vis(N,false);
        queue<int> q; q.push(cur);
        vis[cur] = true;
        while(!q.empty()){
            int c = q.front(); q.pop();
            dists[c] = -INF;
            for(auto ne : adj[c]){
                if(!vis[ne]){
                    vis[ne]=true;
                    q.push(ne);
```

```
                }
            }
        }
    }
};
```

4.3. All distances in graph (neg weights) - Floyd Warshall.

```
'''
Finds all distances in the graph given by edg (negative weights
might occur) edg is a list of (u,v,w) where is an edge from u
to v with weight w.

Time Complexity: O(N^3), N is the number of nodes.
Space Complexity: O(N^2)
'''
inf = 10**15
def fw(N,edg):
    dist = [[inf]*N for _ in range(N)]
    for e in edg:
        dist[e[0]][e[1]] = min(dist[e[0]][e[1]], e[2])
    for i in range(N):
        dist[i][i] = min(0,dist[i][i])
    for k in range(N):
        for i in range(N):
            for j in range(N):
                if dist[i][k] < inf and dist[k][j] < inf:
                    dist[i][j] = min(dist[i][j],
                            dist[i][k] + dist[k][j])
    return dist
```

4.4. Bipartite graphs.

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

int A;
int B;
vector<bool> used;
vector<vector<int> > G;
vector<int> M;
vector<bool> matchedA;
bool tryKuhn(int a){
  if (used[a]) return false;
  used[a] = true;
  for (auto b : G[a]){
    if (M[b] == -1) {
      M[b] = a;
      return true;
    }
  }
  for (auto b : G[a]){
    if (tryKuhn(M[b])) {
      M[b] = a;
      return true;
    }
  }
  return false;
}

void greedyMatching(){
  M.assign(B, -1);
  matchedA.assign(A, false);
  for (int i=0;i<A;i++){
    for (auto b : G[i]){
      if (M[b] == -1){
        M[b] = i;
        matchedA[i] = true;
        break;
      }
    }
  }
  return;
}

void matching(){
  greedyMatching();
  for (int i=0;i<A;i++){
    if (matchedA[i]) continue;
    used.assign(A, false);
    if(tryKuhn(i)) matchedA[i] = true;
  }
```

```
  return;
}

int main(){
  cin >> A >> B;
  G.assign(A, vector<int>());
  for (int i=0;i<A;i++){
    while (true){
      int k;
      cin >> k;
      if (k  == 0) break;
      G[i].push_back(k-1);
    }
  }
  matching();
  int ans = 0;
  for (auto a : M){
    if (a != -1) ans++;
  }
  cout << ans << endl;
  for (int i=0;i<B;i++){
    if (M[i] != -1){
      cout << M[i]+1 << " " << i + 1 << endl;
    }
  }
  return 0;
}
```

4.5. **Network flow.**

```
// C++ implementation of Dinic's Algorithm
// O(V*V*E) for generall flow-graphs. (But with a good constant)
// O(E*sqrt(V)) for bipartite matching graphs.
// O(E*min(V**(2/3),E**(1/3))) For unit-capacity graphs
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
struct Edge{
  ll v ;//to vertex
  ll flow ;
  ll C;//capacity
  ll rev;//reverse edge index
};
// Residual Graph
class Graph
{
public:
  ll V; // number of vertex
  vector<ll> level; // stores level of a node
  vector<vector<Edge>> adj; //can also be array of vector with global size
  Graph(ll V){
    adj.assign(V,vector<Edge>());
    this->V = V;
    level.assign(V,0);
  }

  void addEdge(ll u, ll v, ll C){
    Edge a{v, 0, C, (int)adj[v].size()};// Forward edge
    Edge b{u, 0, 0, (int)adj[u].size()};// Back edge
    adj[u].push_back(a);
    adj[v].push_back(b); // reverse edge
  }

  bool BFS(ll s, ll t){
    for (ll i = 0 ; i < V ; i++)
        level[i] = -1;
    level[s] = 0;  // Level of source vertex
    list< ll > q;
    q.push_back(s);
    vector<Edge>::iterator i ;
    while (!q.empty()){
      ll u = q.front();
      q.pop_front();
      for (i = adj[u].begin(); i != adj[u].end(); i++){
        Edge &e = *i;
        if (level[e.v] < 0  && e.flow < e.C){
          level[e.v] = level[u] + 1;
          q.push_back(e.v);
        }
      }
    }
```

```cpp
    return level[t] < 0 ? false : true; //can/cannot reach target
  }

  ll sendFlow(ll u, ll flow, ll t, vector<ll> &start){
    // Sink reached
    if (u == t)
        return flow;
    // Traverse all adjacent edges one -by - one.
    for (  ; start[u] < (int)adj[u].size(); start[u]++){
      Edge &e = adj[u][start[u]];
      if (level[e.v] == level[u]+1 && e.flow < e.C){
        // find minimum flow from u to t
        ll curr_flow = min(flow, e.C - e.flow);
        ll temp_flow = sendFlow(e.v, curr_flow, t, start);
        // flow is greater than zero
        if (temp_flow > 0){
          e.flow += temp_flow;//add flow
          adj[e.v][e.rev].flow -= temp_flow;//sub from reverse edge
          return temp_flow;
        }
      }
    }
    return 0;
  }
  ll DinicMaxflow(ll s, ll t){
    // Corner case
    if (s == t) return -1;
    ll total = 0;  // Initialize result
    while (BFS(s, t) == true){//while path from s to t
      // store how many edges are visited
      // from V { 0 to V }
      vector <ll> start;
      start.assign(V,0);
      // while flow is not zero in graph from S to D
      while (ll flow = sendFlow(s, 999999999, t, start))
        total += flow;// Add path flow to overall flow
    }
    return total;
  }
};
```

```python
'''
This is an algorithm for calculating max-flow.
edg is an adjacency list, where e[i] is a list of all i's neighbors.
caps is a matrix where caps[i][j] is the current capacity from i to j.
inf is some sufficiently large number (larger than max capacity).
s and t are the source and sink, respectively.
n is the number of nodes.

NOTE: DONT FORGET THE BACKWARDS EDGES WHEN CONSTRUCTING THE GRAPH

Time Complexity: O(C*N)
Space Complexity: O(N^2)
'''
def dfs(vis,df,cmf):
    cur = df.pop()
    vis[cur] = True
    if cur == t: return cmf
    for e in edg[cur]:
        if not vis[e] and caps[cur][e] > 0:
            df.append(e)
            a = dfs(vis,df,min(caps[cur][e],cmf))
            if a:
                caps[cur][e] -= a
                caps[e][cur] += a
                return a
    return 0

def cap():
    c = 0
    toAdd = dfs([False]*n,[s],inf)
    while toAdd:
        c += toAdd
        toAdd = dfs([False]*n,[s],inf)
    return c

#Example of useage.
inf = 10**15
n,m,s,t = map(int, raw_input().split())
edg = [[] for _ in range(n)]
caps = [[0]*n for _ in range(n)]
```

```python
origcaps = [[0]*n for _ in range(n)]
for _ in range(m):
    u,v,c = map(int, raw_input().split())
    edg[u].append(v)
    edg[v].append(u)
    caps[u][v] = c
    origcaps[u][v] = c
mf = cap()
out = []
for node in range(n):
    for ne in edg[node]:
        if origcaps[node][ne] and (origcaps[node][ne]-caps[node][ne]):
            out.append([node,ne,origcaps[node][ne]-caps[node][ne]])

print n, mf, len(out)
for o in out:
    print ' '.join(map(str,o))
'''
This is an algorithm for calculating max-flow.
edg is an adjacency list, where e[i] is a list of all i's neighbors.
caps is a matrix where caps[i][j] is the current capacity from i to j.
inf is some sufficiently large number (larger than max capacity).
s and t are the source and sink, respectively.
n is the number of nodes.

NOTE: DONT FORGET THE BACKWARDS EDGES WHEN CONSTRUCTING THE GRAPH

Time Complexity: O(log(c)*m^2)
Space Complexity: O(n^2)
'''
def dfs(vis,cur,cmf,treshold):
    if vis[cur]: return 0
    vis[cur] = True
    if cur == t: return cmf
    for e in edg[cur]:
        if not vis[e] and caps[cur][e] > treshold:
            a = dfs(vis,e,min(caps[cur][e],cmf),treshold)
            if a:
                caps[cur][e] -= a
                caps[e][cur] += a
                return a
    return 0

def cap():
    c = 0
    for t in range(30,-1,-1):
        toAdd = dfs([False]*n,s,inf,2**t-1)
        while toAdd:
            c += toAdd
            toAdd = dfs([False]*n,s,inf,2**t-1)
    return c

#Example of useage.
inf = 10**15
n,m,s,t = map(int, raw_input().split())
edg = [[] for _ in range(n)]
caps = [[0]*n for _ in range(n)]
origcaps = [[0]*n for _ in range(n)]
for _ in range(m):
    u,v,c = map(int, raw_input().split())
    edg[u].append(v)
    edg[v].append(u)
    caps[u][v] += c
    origcaps[u][v] += c
mf = cap()
out = []
alreadyout = set()
for node in range(n):
    for ne in edg[node]:
        if origcaps[node][ne] and (origcaps[node][ne]-caps[node][ne] > 0) \
                and not (node,ne) in alreadyout:
            out.append([node,ne,origcaps[node][ne]-caps[node][ne]])
            alreadyout.add((node,ne))

print n, mf, len(out)
for o in out:
    print ' '.join(map(str,o))
```

4.6. Min cost max flow.

```python
'''
Solves the min-cost-max-flow problem. This is finding a flow
of maximal capacity (or of capacity at most maxf) with a
minimal cost. Each edge has a capacity and a cost.

Time Complexity: O(min(N^2*M^2, N*M*F))
Space Complexity: O(N^2)

This solution is about 2 times slower than java.
'''

#edge = [to, cap, cost, rev, f]
INF = 10**15

def createGraph(n):
    return [[] for _ in range(n)]

def addEdge(graph, fr, to, cap, cost):
    graph[fr].append([to,cap,cost,len(graph[to]),0])
    graph[to].append([fr,0,-cost,len(graph[fr])-1,0])

#edge = [to, cap, cost, rev, f]
def bellmanFord(s):
    n = len(graph)
    for i in range(n): dist[i] = INF
    dist[s] = 0
    inqueue = [False]*n
    curflow[s] = INF
    q = [0]*n
    qt = 0
    q[qt] = s
    qt += 1
    qh = 0
    while (qh-qt)%n != 0:
        u = q[qh%n]
        inqueue[u] = False
        for i in range(len(graph[u])):
            e = graph[u][i]
            if(e[4] >= e[1]): continue
            v = e[0]
            ndist = dist[u] + e[2]
            if dist[v] > ndist:
                dist[v] = ndist
                prevnode[v] = u
                prevedge[v] = i
                curflow[v] = min(curflow[u], e[1]-e[4])
                if not inqueue[v]:
                    inqueue[v] = True
                    q[qt%n] = v
                    qt += 1
        qh += 1

#edge = [to, cap, cost, rev, f]
def minCostFlow(s, t, maxf):
    n = len(graph)

    flow = 0
    flowCost = 0
    while flow < maxf:
        bellmanFord(s)
        if dist[t] == INF: break
        df = min(curflow[t], maxf - flow)
        flow += df
        v = t
        while v != s:
            e = graph[prevnode[v]][prevedge[v]]
            graph[prevnode[v]][prevedge[v]][4] += df
            graph[v][e[3]][4] -= df
            flowCost += df*e[2]
            v = prevnode[v]
    return (flow, flowCost)

#Example of useage. MUST USE THE SAME NAMES!
N,M,S,T = map(int, raw_input().split())
graph = createGraph(N)
for i in range(M):
    U,V,C,W = map(int, raw_input().split())
    addEdge(graph, U, V, C, W)

dist = [INF]*N
```

```
curflow = [0]*N
prevedge = [0]*N
prevnode = [0]*N
flow, flowCost = minCostFlow(S, T, INF)
print flow, flowCost
```

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

ll INF = 1e18;
// Finds mincost maxflow using a queue based bellmanford
// The queue based is a lot faster than normal bellmanford

struct Edge {
    int to;
    int flow;
    ll cap; //capacity
    ll cost;
    int rev; //reverse edge index
};

class Graph {
public:
    int V;
    vector<vector<Edge> > adj;
    Graph(int V){
        this->V = V;
        adj.assign(V, vector<Edge>());
    }
    void addEdge(int from, int to, ll c, ll cost){
        Edge e = {to, 0, c, cost, adj[to].size()};
        Edge rev = {from, 0, 0, -cost, adj[from].size()};
        adj[from].push_back(e);
        adj[to].push_back(rev);
    }
    // Find augumenting path and send flow
    // Returns added flow and added cost
    pair<ll,ll> bellmanFord(int source, int sink){
        vector<ll> dist(V, INF);
        vector<int> prev(V,-1);
        vector<int> prevEdge(V,-1);
        vector<ll> curFlow(V,INF);
        dist[source] = 0;
        vector<bool> inqueue(V,false);
        queue<int> que;
        que.push(source);
        while(que.size()%V != 0){
            int u = que.front();
            que.pop();
            inqueue[u] = false;
            for (int i=0;i<adj[u].size();i++){
                Edge e = adj[u][i];
                if (e.flow >= e.cap){
                    continue;
                }
                int v = e.to;
                ll ndist = dist[u] + e.cost;
                if (dist[v] > ndist){
                    dist[v] = ndist;
                    prev[v] = u;
                    prevEdge[v] = i;
                    curFlow[v] = min(curFlow[u], e.cap - e.flow);
                    if (!inqueue[v]){
                        inqueue[v] = true;
                        que.push(v);
                    }
                }
            }
        }
        if (dist[sink] == INF) return {0,0};
        ll flow = curFlow[sink];
        int v = sink;
        while (v != source){
            adj[prev[v]][prevEdge[v]].flow += flow;
            adj[v][adj[prev[v]][prevEdge[v]].rev].flow -= flow;
            v = prev[v];
        }
        return {flow, flow * dist[sink]};
    }
    pair<ll,ll> minCostMaxFlow(int S, int T){
```

```
        ll flow = 0, cost = 0;
        pair<ll,ll> temp = bellmanFord(S,T);
        while(temp.first > 0){
            flow += temp.first;
            cost += temp.second;
            temp = bellmanFord(S,T);
        }
        return {flow,cost};
    }
};
```

4.7. Topological sorting - for example finding DAG order.

```
from collections import deque

'''
Gets the topological sorting of the graph given by the adjacency
list adj, where adj[i] is a list of all nodes which are "after"
node i. Returns a sorting, which is given by sort[i] is the
position of node i. The topological sorting is usually performed
on a DAG and is the DAG order. If an the solution is not unique,
this is returned and if contradiction (cycle) is detected False is
returned. These are easy to change to suit the problem.

Time-Complexity: O(m+n), n is the number of nodes.
Space-Complexity: O(m+n)
'''
def topsort(adj):
    N = len(adj)
    par = [0]*N
    for l in adj:
        for node in l:
            par[node] += 1
    sorting = []
    queue = deque([])
    for i in range(N):
        if par[i] == 0:
            sorting.append(i)
            queue.append(i)

    while queue:
        cur = queue.popleft()
        for child in adj[cur]:
            par[child] -= 1
            if par[child] == 0:
                queue.append(child)
                sorting.append(child)

    if len(sorting) < N: return None
    return sorting
```

4.8. 2sat.

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<ll,ll> pii;

class twosat{
public:
    //for variable i, two variables are assigned as 2*i and
    //2*i+1 in G. 2*i is i, and 2*i+1 is not i.
    //Note that this has to be taken care of when adding clauses.
    vector<vector<int> > G_forward, G_reverse;
    vector<int> x,y;
    ll N;
    twosat(ll var){
        N = var*2;
        G_forward.assign(N,vector<int>());
        G_reverse.assign(N,vector<int>());
        marked.assign(N,false);
        component.assign(N,-1);
    }
    //addClause(i,j) adds the clause from i to j. But negations have
    //to be considered in the main.
    void addClause(int i, int j){
        G_forward[i^1].push_back(j);
        G_forward[j^1].push_back(i);
        G_reverse[i].push_back(j^1);
        G_reverse[j].push_back(i^1);
        x.push_back(i); y.push_back(j);
    }
```

```
    bool solve(){
      for(int i = 0; i < N; i++)
          if(!marked[i]) dfsFirst(i);
      marked.assign(N, false);
      while(!stck.empty()){
        int v = stck.back();
        stck.pop_back();
        if (!marked[v]){
          counter++;
          dfsSecond(v);
        }
      }
      for(int i = 0; i < N; i+=2)
          if(component[i] == component[i+1]) return false;
      return true;
    }
private:
    vector<bool> marked;
    vector<int> stck,component;
    int counter = 0;
    void dfsFirst(int v){
        marked[v] = true;
        for(auto u : G_forward[v]){
            if(!marked[u]) dfsFirst(u);
        }
        stck.push_back(v);
    }
    void dfsSecond(int v){
        marked[v] = true;
        for(auto u : G_reverse[v])
            if(!marked[u]) dfsSecond(u);
        component[v] = counter;
    }
};
```

5. Dynamic Programming

5.1. Longest increasing subsequence.

```
'''
Returns the longest increasing of list X.

Time Complexity: O(N logN), N = len(X)
Space Complexity: O(N)
'''

def lis(X):
    L = 0
    N = len(X)
    P = [-1]*N
    M = [-1]*(N+1)

    for i in range(N):
        lo = 1
        hi = L
        while lo <= hi:
            mid = (lo+hi+1)/2
            if X[M[mid]] < X[i]:
                lo = mid + 1
            else:
                hi = mid - 1
        newL = lo
        P[i] = M[newL-1]
        M[newL] = i
        if newL > L:
            L = newL

    S = [-1]*L
    k = M[L]
    for i in range(L-1,-1,-1):
        S[i] = X[k]
        k = P[k]

    return S
```

5.2. String functions.

```
'''
Generates the z-function and boarder function for a string s.

Time Complexity: O(len(s))
Space Complexity: O(len(s))
```

```
'''
#z[i] = Length of the longest common prefix of s and s[i:], i > 0.
def zfun(s):
    n = len(s)
    z = [0]*n
    L,R = (0,0)
    for i in range(1,n):
        if i < R:
            z[i] = min(z[i-L], R-i+1)
        while z[i] + i < n and s[i+z[i]] == s[z[i]]:
            z[i] += 1
            if i + z[i] - 1 > R:
                L = i
                R = i + z[i] - 1
    return z

#b[i] = Length of longest suffix of s[:i] that is a prefix of s.
def boarders(s):
    n = len(s)
    b = [0]*n
    for i in range(1,n):
        k = b[i-1]
        while k > 0 and s[k] != s[i]: k = b[k-1]
        if s[k] == s[i]: b[i] = k + 1
    return b
```

5.3. **Josephus problem.**

```
'''
Solves the problem of counting out. Given n people and
counting out every k-th person, josephus(n,k) gives the
last person standing.

Time Complexity: O(n)
Space Complexity: O(n)
'''
def josephus(n,k):
    DP = [-1]*(n+1)
    DP[1] = 0
    for i in range(2,n+1):
        DP[i] = (DP[i-1]+k)%i
    return DP[n]
```

5.4. **Knapsack.**

```
def knapsack(w, v, W):
    n = len(w)
    DP = [[0]*(W+1) for _ in range(n+1)]
    for j in range(W+1): DP[0][j] = 0
    for i in range(1,n+1):
        for j in range(W+1):
            if w[i-1] > j: #If it is not possible to put i in the bag
                DP[i][j] = DP[i-1][j]
            else: #Otherwise we either put it or not.
                DP[i][j] = max(DP[i-1][j], DP[i-1][j-w[i-1]] + v[i-1])
    return DP
```

6. Coordinate Geometry

6.1. **Area of polygon.**

```
'''
Calculates the area of the convex polygon given by the
points in pts (that are given in the right order).

Time-Complexity: O(n), n = len(pts)
Space-Complexity: O(n)
'''
from __future__ import division

def area(pts):
    out = 0
    for i in range(-1,len(pts)-1):
        out += pts[i][0]*pts[i+1][1]-pts[i][1]*pts[i+1][0]
    return abs(out/2)
```

6.2. **General geometry operations on lines, segments and points.**

```
from __future__ import division
'''
Contains the most common geometric operations on points,
segments and lines.
Points are represented as (x,y)
Segments are represented as (x1,y1,x2,y2)
```

```
Lines are represented as (a,b,c), where ax+by+c=0 is the
equation of the line.

Contains the following operations:
    Getting a line from two points
    Getting intersection between pairs of lines or segments
    Getting the closest point on a line or segment to a point
    Getting distance from a point to a point, segment or line
    Finding out if a point is on a segment or not.

Time Complexity: O(1)
Space Complexity: O(1)
'''
#Returns a line from two points.
def two_points_to_line(x1,y1,x2,y2):
    return (y2-y1,x1-x2,x2*y1-y2*x1)

#Returns the intersection between the lines.
#Assumes the lines have either a or b different from 0.
def line_line_intersect(line1,line2):
    a1,b1,c1 = line1
    a2,b2,c2 = line2
    cp = a1*b2 - a2*b1
    if cp!=0:
        return ((b1*c2-b2*c1)/cp,(a2*c1-a1*c2)/cp)
    else:
        if a1*c2==a2*c1 and b1*c2==b2*c1:
            return line1
        return None

#Returns the intersection between two segments.
#Assumes the segments have length > 0.
#Return value is None, a point or a segment.
def seg_seg_intersect(seg1,seg2):
    line1=two_points_to_line(*seg1)
    line2=two_points_to_line(*seg2)
    p=line_line_intersect(line1,line2)
    if p == None: return None
    if len(p)==2:
        if weak_point_on_seg(seg1,p) and weak_point_on_seg(seg2,p):
            return p
        return None
    pts = [(seg1[0],seg1[1],0), (seg1[2],seg1[3],0),
            (seg2[0],seg2[1],1), (seg2[2],seg2[3],1)]
    pts.sort()
    if pts[1][0] == pts[2][0] and pts[1][1] == pts[2][1]\
            and pts[1][2] != pts[2][2]:
        return (pts[1][0],pts[1][1])
    if pts[0][2] != pts[1][2]:
        return (pts[1][0],pts[1][1],pts[2][0],pts[2][1])
    return None

#Returns the point on the segment closest to p.
def seg_point_project(seg, p):
    line = two_points_to_line(*seg)
    p2 = line_point_project(line,p)
    if weakPointInsideSegment(p2,seg):
        return p2
    else:
        if dist(p,(seg[0],seg[1])) < dist(p,(seg[2],seg[3])):
            return (seg[0],seg[1])
        else:
            return (seg[2],seg[3])

#Returns the orthogonal projection of a point onto a line.
def line_point_project(line, p):
    a,b,c=line
    x,y=p
    return ((b*(b*x-a*y)-a*c)/(a**2+b**2),
            (a*(-b*x+a*y)-b*c)/(a**2+b**2))

#Returns the euclidean distance between two points.
def dist(p1,p2):
    return ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)**0.5

#Returns the distance from a point to a segment.
def seg_point_dist(seg,p):
    p2 = seg_point_project(seg,p)
    return dist(p,p2)
```

```
#Returns the distance from a point to a line.
def line_point_dist(line,p):
    p2 = line_point_project(line, p)
    return dist(p,p2)

#Returns if point p is on segment seg.
def point_on_seg(seg,p):
    x,y = p
    x1,y1,x2,y2 = seg
    if (x-x1)*(y-y2) == (x-x2)*(y-y1):
        return (x-x1)*(x-x2) <= 0 and (y-y1)*(y-y2) <= 0
    return False

#Only checks that the order of the points is correct.
def weak_point_on_seg(seg,p):
    x,y = p
    x1,y1,x2,y2 = seg
    return (x-x1)*(x-x2) <= 0 and (y-y1)*(y-y2) <= 0
```

6.3. **Pick's theorem.** Pick's theorem states that the area, A, of a polygon with lattice coordinates for its corners is given by

$$A = I + \frac{B}{2} - 1,$$

where B is the number of boundary lattice points and I is the number of interior lattice points. This can often be used to find the number of interior points of a polygon since the area is easily computed, see 6.1, and the number of boundary lattice points is calculated as follows:

```
'''
Calculates the number of lattice boundary points of the
polygon given by pts (including the points in pts). pts has
to be sorted either in clockwise or counter clockwise order.

Time Complexity: O(nlogn), where n = len(pts).
Space Complexity: O(n)
'''
def gcd(a,b):
    if a < b: return gcd(a,b)
    if b == 0: return a
    return gcd(b,a%b)

def boundarypoints(pts):
    n = len(pts)
    out = 0
    for i in range(-1,n-1):
        dx = abs(pts[i][0]-pts[i+1][0])
        dy = abs(pts[i][1]-pts[i+1][1])
        out += gcd(dx,dy)
    return out
```

6.4. **Convex Hull.**

```
'''
Returns the convex hull in counter-clockwise order of the points
in pts. A point is represented by (x,y).

Time Complexity: O(nlogn), n is the number of points.
Space Complexity: O(n)
'''

def ccw(p1,p2,p3):
    return (p2[0]-p1[0])*(p3[1]-p1[1])-(p2[1]-p1[1])*(p3[0]-p1[0])

#Returns hull in counter-clockwise order.
#pts is a list of tuples, each tuple is (x,y).
def hull(pts):
    n = len(pts)
    pts.sort()
    U = []
    L = []
    for i in range(n):
        while len(L)>1 and ccw(L[-2],L[-1],pts[i]) <= 0: L.pop()
        L.append(pts[i])
    for i in range(n-1,-1,-1):
        while len(U)>1 and ccw(U[-2],U[-1],pts[i]) <= 0: U.pop()
        U.append(pts[i])
    L.pop()
    U.pop()
    if len(L) == len(U) == 1 and L[0] == U[0]: return L
    return L+U
```

7. Math

7.1. System of equations.

```
from __future__ import division
'''
Solves Ax=b. A has size n*n, b has size n*1
Returns x if unique solution exists, otherwise
'multiple' or 'inconsistent'.

Time Complexity: O(n^3)
Space Complexity: O(n^2)
'''
def gaussianelimination(A,b):
    h = 0
    k = 0
    n = len(A)
    while h < n and k < n:
        imax = h
        for i in range(h+1,n):
            if abs(A[i][k]) > abs(A[imax][k]): imax = i
        if A[imax][k] == 0: k += 1
        else:
            temp = A[h]
            A[h] = A[imax]
            A[imax] = temp
            temp2 = b[h]
            b[h] = b[imax]
            b[imax] = temp2
            for i in range(h+1,n):
                f = A[i][k] / A[h][k]
                A[i][k] = 0
                for j in range(k+1,n):
                    A[i][j] -= A[h][j]*f
                b[i] -= b[h]*f
            h += 1
            k += 1
    x = [-1]*n
    if A[n-1][n-1] == 0 and b[n-1] == 0: return 'multiple'
    elif A[n-1][n-1] == 0 and b[n-1] != 0: return 'inconsistent'
    else: x[n-1] = b[n-1]/A[n-1][n-1]
    for i in range(n-2,-1,-1):
        s = 0
        for j in range(i+1,n): s += A[i][j]*x[j]
        x[i] = (b[i]-s)/A[i][i]
    return x
```

7.2. Number Theory.

```
'''
Returns gcd for two numbers, or for all numbers in a list.
Also returns Bezout's identity.

Time Complexity: O(N) (if b == 1), O(logN) for random numbers,
                N = a+b.
Space Complexity: O(1)
TODO: Do it iteratively.
'''
def gcd(a,b):
    if a < b: return gcd(b,a)
    if b == 0: return a
    return gcd(b,a%b)

def listgcd(l):
    if len(l) == 1: return l[0]
    else: return listgcd(l[:-2]+[gcd(l[-2],l[-1])])

#Returns (u,v) such that au+bv = gcd(a,b)
def bezout(a,b):
    if a < b:
        v,u = bezout(b,a)
        return (u,v)
    if b == 0: return (1,0)
    u1,v1 = bezout(b,a%b)
    return (v1,u1-a//b*v1)
```

7.3. Chinese remainder theorem.

```
'''
Implementation of the chineese remainder theorem.
The equation is x = a_i mod b_i for a_i in a, b_i in b.

Time Complexity: O(n^2), n = len(a)=len(b).
```

```
Space complexity: O(n)
'''

def gcd(a,b):
    if a < b: return gcd(b,a)
    if b == 0: return a
    return gcd(b,a%b)

#Returns (u,v) such that au+bv = gcd(a,b)
def bezout(a,b):
    if a < b:
        v,u = bezout(b,a)
        return (u,v)
    if b == 0: return (1,0)
    u1,v1 = bezout(b,a%b)
    return (v1,u1-a//b*v1)

#Solves x = a_i mod b_i for a_i in a, b_i in b.
def crt(a,b):
    if len(a) == 1: return (a[0],b[0])
    c1,c2,m1,m2 = (a[-2],a[-1],b[-2],b[-1])
    k = gcd(m1,m2)
    if c1%k != c2%k: return (False, False)
    r = c1%k
    u,v = bezout(m1/k,m2/k)
    x = ((((c1//k)*v*(m2//k) + \
           (c2//k)*u*(m1//k))%(m1*m2/k/k))*k + r) % (m1*m2/k)
    return crt(a[:-2]+[x], b[:-2]+[m1*m2/k])
```

7.4. Finding large fibonacci numbers.

```
'''
Returns the x-th fibonacci number modulo MOD,
f(0)=f(1)=1.
Time Complexity: O(log(x)*log(x))
Space Complexity: O(log(x)*log(x))
'''
DP = {}
def fib(x):
    if x < 2: return 1
    if x in DP: return DP[x]
    st = 1
    while x >= 2**(st+1): st += 1
    out = steps[st][0]*fib(x-2**st+1) + steps[st][1]*fib(x-2**st)
    out %= MOD
    DP[x] = out
    return out
MOD = 10**9
MAX_2pot = 64
steps = [(0,0),(1,1)]
for _ in range(2,MAX_2pot+1):
    a,b = steps[-1]
    a2 = (a*a+2*a*b)%MOD
    b2 = (a*a+b*b)%MOD
    steps.append((a2,b2))
```

7.5. Finding primitive root.

```
def primefactors(n):
    out = set()
    for i in range(2,int(n**0.5)+3):
        if n % i == 0:
            out2 = primefactors(n/i)
            out.add(i)
            for o in out2: out.add(o)
            return out
    out.add(n)
    return out

def primroot(p):
    ps = primefactors(p-1)
    for i in range(2,p-2):
        suc = True
        for pp in ps:
            if pow(i,(p-1)/pp,p) == 1:
                suc = False
                break
        if suc: return i
    return False
```

7.6. Baby-step-giant-step algorithm.

```python
'''
Solves a^x=b mod P, where a is a number, P is a prime
and b is an arbitrary number. Here a is usually a
primitive root with respect to the prime P.

Time-complexity: O(sqrt(P))
Space-complexity: O(sqrt(P))
'''
def babystepgiantstep(a,b,P):
    m = int(P**0.5) + 1
    aminv = pow(pow(a,m,P),P-2,P)
    vals = {}
    for j in range(m):
        val = pow(a,j,P)
        if val not in vals: vals[val] = j
    for i in range(m):
        if b in vals:
            return i*m+vals[b]
        b *= aminv
        b %= P
    return -1
```

8. Other things

8.1. Fast Fourier Transform.

```cpp
#include <bits/stdc++.h>
#include <math.h>
using namespace std;
typedef long long ll;
ll mod=998244353;
ll generator=5; //Not used but need to find this
ll modomega1=961777435; //assuming n=2^23

vector<complex<double> > omega;
vector<ll> modomega;
vector<ll> r;
ll n;
ll logN;
double pi;
vector<complex<double> > fft(vector<complex<double> > inp){
    vector<complex<double> > ret;
    for(ll i = 0; i < (ll) inp.size(); i++) ret.push_back(inp[r[i]]);
    for(ll k = 1; k < n; k = k*2){
        for(ll i = 0; i < n; i = i + 2*k){
            for(ll j = 0; j < k; j++){
                complex<double> z = omega[j*n/(2*k)] * ret[i + j + k];
                ret[i + j + k] = ret[i + j] - z;
                ret[i + j] = ret[i + j] + z;
            }
        }
    }
    return ret;
}
vector<ll> modfft(vector<ll> inp) {
  vector<ll> ret;
  for(ll i = 0; i < (ll) inp.size(); i++) ret.push_back(inp[r[i]]);
  for(ll k = 1; k < n; k = k*2){
      for(ll i = 0; i < n; i = i + 2*k){
          for(ll j = 0; j < k; j++){
              ll z = (modomega[j*n/(2*k)] * ret[i + j + k])%mod;
              ret[i + j + k] = (ret[i + j] - z + mod)%mod;
              ret[i + j] = (ret[i + j] + z)%mod;
          }
      }
  }
  return ret;
}

void init() {
  r.push_back(0);
  for(ll i = 1; i < n; i++)
      r.push_back(r[i/2]/2 + ((i&1) << (logN-1)));
  for(ll i = 0; i < n; i++)
      omega.push_back({cos(2*i*pi/n),sin(2*i*pi/n)});
  modomega.push_back(1);
  for(ll i = 1; i < n; i++)
      modomega.push_back((modomega[i-1]*modomega1)%mod);
}
//needs to be tweaked for modfft
```

```
vector<complex<double> > ifft(vector<complex<double> > inp){
    vector<complex<double> > temp;
    temp.push_back(inp[0]);
    for(ll i = n-1; i > 0; i--) temp.push_back(inp[i]);
    temp = fft(temp);
    for(ll i = 0; i < n; i++) temp[i] /= n;
    return temp;
}

int main(){
    pi = atan(1)*4;
    ll T, deg1, deg2; cin >> T >> deg1;
    vector<complex<double> > a1,a2;
    for(int i = 0; i <= deg1; i++){double c; cin >> c;
        a1.push_back({c,0});}
    cin >> deg2;
    for(int i = 0; i <= deg2; i++){double c; cin >> c;
        a2.push_back({c,0});}
    n = 2; ll counter = 1;
    while (n <= deg1 + deg2){n *= 2; counter++;}
    while ((ll) a1.size() < n) a1.push_back({0,0});
    while ((ll) a2.size() < n) a2.push_back({0,0});
    logN=counter;
    init();
    vector<complex<double> > b1, b2;
    b1 = fft(a1); b2 = fft(a2);
    vector<complex<double> > c;
    for(ll i = 0; i < n; i++) c.push_back(b1[i]*b2[i]);

    vector<complex<double> > out = ifft(c);
    vector<ll> outs;
    for(ll i = 0; i <= deg1 + deg2; i++)
        outs.push_back(round(out[i].real()));

    cout << deg1 + deg2 << endl;
    for(ll i = 0; i < (ll) outs.size(); i++) cout << outs[i] << " ";
    cout << endl;
    return 0;
}
```

8.2. Large Primes.

- 133469857
- 1519262429
- 17024073439
- 3435975962563
- 22732918586849
- 22734054029887
- $10^9 + 7$
- $10^9 + 9$
- $13631489 = 2^{20} \cdot 13 + 1$
- $120586241 = 2^{20} \cdot 5 \cdot 23 + 1$
- $998244353 = 2^{23} \cdot 7 \cdot 17 + 1$

8.3. Scheduling.

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

bool comp(pair<ll,ll> p1, pair<ll,ll> p2){
    if(p1.second != p2.second) return p1.second < p2.second;
    return p1.first < p2.first;
}

/**
 *  Returns the number of jobs that can be completed with k
 *  working stations. jobs is a vector of pairs, that contain
 *  start and end time.
 *
 *  Time-complexity: O(n log k), where n is the number of jobs
 *  Space-complexity: O(n + k)
 */
int schedule(vector<pair<ll,ll> > jobs, int k){
    int no_scheduled = 0;
    sort(jobs.begin(),jobs.end(),comp);
    set<pair<ll,int> > stations;
    for(int i = 0; i < k; i++) stations.insert({0,i});
    for(auto job : jobs){
        auto it = stations.lower_bound({job.first,k});
        if(it == stations.begin()) continue;
        pair<ll,int> toins = {job.second,(--it)->second};
```

```
        stations.erase(it);
        stations.insert(toins);
        no_scheduled++;
    }
    return no_scheduled;
}

int main(){
    int n,k; cin >> n >> k;
    vector<pair<ll,ll> > jobs;
    for(int i = 0; i < n; i++){
        ll s,t; cin >> s >> t; jobs.push_back({s,t});
    }
    cout << schedule(jobs,k) << endl;
}
```

9. Methods and ideas

Use some characteristics of the problem (i.e. bounds)

- $N \leq 10$: Exhaustive search $N!$
- $N \leq 20$: Exponential, bitmask-DP?
- $N \leq 10^4$: Quadratic
- $N \leq 10^6$: Has to be NlogN

Greedy

- Invariants
- Scheduling

BFS/DFS
DP

- Bitmask
- Recursively, storing answers

Binary search

- Over the answer
- To find something in sorted structure

Flow

- Min-cost-max-flow
- Run the flow and look at min cut
- Regular flow
- Matching

View the problem as a graph
Color the graph
When there is an obvious TLE solution

- Use some sorted data structure
- In DP, drop one parameter and recover from others
- Is something bounded by the statement?
- In DP, use FFT to reduce one N to logN

Divide and conquer - find interesting points in NlogN
Square-root tricks

- Periodic rebuilding: every $\sqrt{n}$, rebuild static structure.
- Range queries: split array into segments, store something for each segment.
- Small and large: do something for small(with low degree) nodes and something else for large nodes.
- If the sum of some parameters is small, then the number of different sized parameters is bounded by roughly $\sqrt{n}$.

Hall's marriage theorem
Combinatorics / Number theory / Maths

- Inclusion/Exclusion
- Fermat's little theorem / Euler's theorem
- NIM

Randomization

- Finding if 3 points are on the same line
- Checking matrix equality by randomizing vector and multiply

Geometry

- Cross product - to check order of points / area
- Scalar product

10. Practice Contest Checklist

- Operations per second in py2
- Operations per second in py3
- Operations per second in java
- Operations per second in c++
- Operations per second on local machine
- Is MLE called MLE or RTE?
- What happens if extra output is added? What about one extra new line or space?
- Look at documentation on judge.
- Submit a clar.
- Print a file.
- Directory with test cases.
- Check how to change keyboard layout (english, swedish)
- Check that bash script works