Rainbow Unicode Characters
Team Reference Document
Lund University

Contents

1. Achieving AC on a solved problem

1.1. WA.

- Check that minimal input passes.
- Can an int overflow?
- Reread the problem statement.
- Start creating small test cases with python.
- Does cout print with high enough precision?
- Abstract the implementation.

1.2. TLE.

- Is the solution sanity checked?
- Use pypy instead of python.
- Rewrite in C++ or Java.
- Can we apply DP anywhere?
- To minimize penalty time you should create a worst case input (if easy) to test on.
- Binary Search over the answer?

1.3. RTE.

- Recursion limit in python?
- Arrayindex out of bounds?
- Division by 0?
- Modifying iterator while iterating over it?
- Not using a well defined operator for Collections.sort?
- If nothing makes sense and the end of the contest is approaching you can binary search over where the error is with try-except.

1.4. MLE.

- Create objects outside recursive function.
- Rewrite recursive solution to iterative with an own stack.

2. Ideas

2.1. A TLE solution is obvious.

- If doing dp, drop parameter and recover from others.
- Use a sorted data structure.
- Is there a hint in the statement saying that something more is bounded?

2.2. Try this on clueless problems.

- Try to interpret problem as a graph (D - NCPC2017).
- Can we apply maxflow, with mincost?
- How does it look for small examples, can we find a pattern?
- Binary search over solution.
- If problem is small, just brute force instead of solving it cleverly. Some times its enough to iterate over the entire domains instead of using xgcd.

3. Code Templates

3.1. .bashrc. Aliases.

```
alias p2=python2
alias p3=python3
alias nv=vim
alias o="xdg-open ."
setxkbmap -option 'nocaps:ctrl'
```

3.2. .vimrc. Tabs, line numbers, wrapping

```
set nowrap
syntax on
set tabstop=8 softtabstop=0 shiftwidth=4
set expandtab smarttab
set autoindent smartindent
set rnu number
set scrolloff=8
filetype plugin indent on
```

3.3. run.sh. Bash script to run all tests in a folder.

```
#!/bin/bash
# make executable: chmod +x run.sh
# run: ./run.sh A pypy A.py
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
```

3.4. **Java Template.** A Java template.

```
import java.util.*;
import java.io.*;
public class A {
    void solve(Kattio io) {

    }
    void run() {
        Kattio io = new Kattio(System.in, System.out);
        solve(io);
        io.flush();
    }
    public static void main(String[] args) {
        (new A()).run();
    }
    class Kattio extends PrintWriter {
        public Kattio(InputStream i) {
            super(new BufferedOutputStream(System.out));
            r = new BufferedReader(new InputStreamReader(i));
        }
        public Kattio(InputStream i, OutputStream o) {
            super(new BufferedOutputStream(o));
            r = new BufferedReader(new InputStreamReader(i));
        }

        public boolean hasMoreTokens() {
            return peekToken() != null;
        }

        public int getInt() {
            return Integer.parseInt(nextToken());
        }

        public double getDouble() {
            return Double.parseDouble(nextToken());
        }

        public long getLong() {
            return Long.parseLong(nextToken());
        }
```

```
        public String getWord() {
            return nextToken();
        }

        private BufferedReader r;
        private String line;
        private StringTokenizer st;
        private String token;

        private String peekToken() {
            if (token == null)
                try {
                    while (st == null || !st.hasMoreTokens()) {
                        line = r.readLine();
                        if (line == null) return null;
                        st = new StringTokenizer(line);
                    }
                    token = st.nextToken();
                } catch (IOException e) { }
            return token;
        }

        private String nextToken() {
            String ans = peekToken();
            token = null;
            return ans;
        }
        private String joinRemainder() {
            ArrayList<String> tokens = new ArrayList<>();
            while (st.hasMoreTokens()) {
                tokens.add(st.nextToken());
            }
            return String.join(" ", tokens);
        }
        public String remainingLine() {
            if(st != null && st.hasMoreTokens()) {
                return joinRemainder();
            }
            return nextLine();
```

```
        }
        public String nextLine() {
            try {
                return r.readLine();
            } catch(IOException e) {
                return null;
            }
        }
    }
}
```

3.5. **Python Template.** A Python template

```
#!/usr/bin/python3
from collections import *
from itertools import permutations #No repeated elements
import sys
sys.setrecursionlimit(10**5)
itr = (line for line in sys.stdin.read().split('\n'))
INP = lambda: next(itr)
def ni(): return int(INP())
def nl(): return [int(_) for _ in INP().split()]
def err(*s): print(*s, file=sys.stderr)

def main():

    return

if __name__ == '__main__':
    main()
```

3.6. **C++ Template.** A C++ template

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

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define trav(a, x) for(auto& a : x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
typedef long long ll;
ll smod(ll a, ll b){
    return (a % b + b) % b;
}
int main() {
    cout.precision(9);
    cin.sync_with_stdio(0); cin.tie(0);
    cin.exceptions(cin.failbit);
    int N;
    cin >> N;
    cout << 0 << endl;
}
```

4. Data Structures

4.1. **Fenwick Tree.** Also called a Binary indexed tree. Builds in $\mathcal{O}(n \log n)$ from an array. Querry sum from 0 to i in $\mathcal{O}(\log n)$ and updates an element in $\mathcal{O}(\log n)$.

```
# Tested on: https://open.kattis.com/problems/froshweek
class FenwickTree: # zero indexed calls!
    # Give array or size!
    def __init__(self, blob):
        if type(blob) == int:
            self.sz = blob
            self.data = [0]*(blob+1)
        elif type(blob) == list:
            A = blob
            self.sz = len(A)
            self.data = [0]*(self.sz + 1)
            for i, a in enumerate(A):
                self.inc(i, a)
    # A[i] = v
    def assign(self, i, v):
        currV = self.query(i, i)
        self.inc(i, v - currV)
    # A[i] += delta
    # this method is ~3x faster than doing A[i] += delta
    def inc(self, i, delta):
```

```
        i += 1 # (to 1 indexing)
        while i <= self.sz:
            self.data[i] += delta
            i += i&-i # lowest oneBit
    # sum(A[:i+1])
    def sum(self, i):
        i += 1 # (to 1 indexing)
        S = 0
        while i > 0:
            S += self.data[i]
            i -= i&-i
        return S
    # return sum(A[lo:hi+1])
    def query(self, lo, hi):
        return self.sum(hi) - self.sum(lo-1)

    # for indexing - nice to have but not required
    def __fixslice__(self, k):
        return slice(k.start or 0, self.sz if k.stop == None else k.stop)
    def __setitem__(self, i, v):
        self.assign(i, v)
    def __getitem__(self, k):
        if type(k) == slice:
            k = self.__fixslice__(k)
            return self.query(k.start, k.stop - 1)
        elif type(k) == int:
            return self.query(k, k)
```

4.2. **Segment Tree.** More general than a Fenwick tree. Can adapt other operations than sum, e.g. min and max.

```
# Tested on: https://open.kattis.com/problems/supercomputer
class SegmentTree:
    def __init__(self, arr, func=min):
        self.sz = len(arr)
        assert self.sz > 0
        self.func = func
        sz4 = self.sz*4
        self.L, self.R = [None]*sz4, [None]*sz4
        self.value = [None]*sz4
        def setup(i, lo, hi):
            self.L[i], self.R[i] = lo, hi
            if lo == hi:
                self.value[i] = arr[lo]
                return
            mid = (lo + hi)//2
            setup(2*i, lo, mid)
            setup(2*i + 1, mid+1, hi)
            self._fix(i)
        setup(1, 0, self.sz-1)
    def _fix(self, i):
        self.value[i] = self.func(self.value[2*i], self.value[2*i+1])

    def _combine(self, a, b):
        if a is None: return b
        if b is None: return a
        return self.func(a, b)

    def query(self, lo, hi):
        assert 0 <= lo <= hi < self.sz
        return self.__query(1, lo, hi)

    def __query(self, i, lo, hi):
        l, r = self.L[i], self.R[i]
        if r < lo or hi < l:
            return None
        if lo <= l <= r <= hi:
            return self.value[i]
        return self._combine(
            self.__query(i*2, lo, hi),
            self.__query(i*2 + 1, lo, hi)
            )

    def assign(self, pos, value):
        assert 0 <= pos < self.sz
        return self.__assign(1, pos, value)

    def __assign(self, i, pos, value):
        l, r = self.L[i], self.R[i]
        if pos < l or r < pos: return
        if pos == l == r:
```

```
            self.value[i] = value
            return
        self.__assign(i*2, pos, value)
        self.__assign(i*2 + 1, pos, value)
        self._fix(i)

    def inc(self, pos, delta):
        assert 0 <= pos < self.sz
        self.__inc(1, pos, delta)

    def __inc(self, i, pos, delta):
        l, r = self.L[i], self.R[i]
        if pos < l or r < pos: return
        if pos == l == r:
            self.value[i] += delta
            return
        self.__inc(i*2, pos, delta)
        self.__inc(i*2 + 1, pos, delta)
        self._fix(i)

    # for indexing - nice to have but not required
    def __setitem__(self, i, v):
        self.assign(i, v)
    def __fixslice__(self, k):
        return slice(k.start or 0, self.sz if k.stop == None else k.stop)
    def __getitem__(self, k):
        if type(k) == slice:
            k = self.__fixslice__(k)
            return self.query(k.start, k.stop - 1)
        elif type(k) == int:
            return self.query(k, k)
```

4.3. **Lazy Segment Tree.** More general implementation of a segment tree where its possible to increase whole segments by some diff, with lazy propagation. Implemented with arrays instead of nodes, which probably has less overhead to write during a competition.

```
class LazySegmentTree {
  private int n;
  private int[] lo, hi, sum, delta;
  public LazySegmentTree(int n) {
    this.n = n;
    lo = new int[4*n + 1];
    hi = new int[4*n + 1];
    sum = new int[4*n + 1];
    delta = new int[4*n + 1];
    init();
  }
  public int sum(int a, int b) {
    return sum(1, a, b);
  }
  private int sum(int i, int a, int b) {
    if(b < lo[i] || a > hi[i]) return 0;
    if(a <= lo[i] && hi[i] <= b) return sum(i);
    prop(i);
    int l = sum(2*i, a, b);
    int r = sum(2*i+1, a, b);
    update(i);
    return l + r;

  }
  public void inc(int a, int b, int v) {
    inc(1, a, b, v);
  }
  private void inc(int i, int a, int b, int v) {
    if(b < lo[i] || a > hi[i]) return;
    if(a <= lo[i] && hi[i] <= b) {
      delta[i] += v;
      return;
    }
    prop(i);
    inc(2*i, a, b, v);
    inc(2*i+1, a, b, v);
    update(i);
  }

  private void init() {
    init(1, 0, n-1, new int[n]);
  }
  private void init(int i, int a, int b, int[] v) {
```

```
    lo[i] = a;
    hi[i] = b;
    if(a == b) {
      sum[i] = v[a];
      return;
    }
    int m = (a+b)/2;
    init(2*i, a, m, v);
    init(2*i+1, m+1, b, v);
    update(i);
  }
  private void update(int i) {
    sum[i] = sum(2*i) + sum(2*i+1);
  }
  private int range(int i) {
    return hi[i] - lo[i] + 1;
  }
  private int sum(int i) {
    return sum[i] + range(i)*delta[i];
  }
  private void prop(int i) {
    delta[2*i] += delta[i];
    delta[2*i+1] += delta[i];
    delta[i] = 0;
  }
}
```

4.4. **Union Find.** This data structure is used in various algorithms, for example Kruskal's algorithm for finding a Minimal Spanning Tree in a weighted graph. Also it can be used for backward simulation of dividing a set.

```
class UnionFind:
    def __init__(self, N):
        self.parent = [i for i in range(N)]
        self.sz = [1]*N
    def find(self, i):
        path = []
        while i != self.parent[i]:
            path.append(i)
            i = self.parent[i]
        for u in path: self.parent[u] = i
        return i
    def union(self, u, v):
        uR, vR = map(self.find, (u, v))
        if uR == vR: return False
        if self.sz[uR] < self.sz[vR]:
            self.parent[uR] = vR
            self.sz[vR] += self.sz[uR]
        else:
            self.parent[vR] = uR
            self.sz[uR] += self.sz[vR]
        return True
```

4.5. **Monotone Queue.** Used in sliding window algorithms where one would like to find the minimum in each interval of a given length. Amortized $\mathcal{O}(n)$ to find min in each of these intervals in an array of length n. Can easily be used to find the maximum as well.

```
private static class MinMonQue {
    LinkedList<Integer> que = new LinkedList<>();
    public void add(int i) {
        while(!que.isEmpty() && que.getFirst() > i)
            que.removeFirst();
        que.addFirst(i);
    }
    public int last() {
        return que.getLast();
    }
    public void remove(int i) {
        if(que.getLast() == i) que.removeLast();
    }
}
```

4.6. **Treap.** Treap is a binary search tree that uses randomization to balance itself. It's easy to implement, and gives you access to the internal structures of a binary tree, which can be used to find the k'th element for example. Because of the randomness, the average height is about a factor 4 of a perfectly balanced tree.

```
class Treap{
  int sz;
  int v;
  double y;
  Treap L, R;
```

```
static int sz(Treap t) {
  if(t == null) return 0;
  return t.sz;
}
static void update(Treap t) {
  if(t == null) return;
  t.sz = sz(t.L) + sz(t.R) + 1;
}
static Treap merge(Treap a, Treap b) {
  if (a == null) return b;
  if(b == null) return a;
  if (a.y < b.y) {
    a.R = merge(a.R, b);
    update(a);
    return a;
  } else {
    b.L = merge(a, b.L);
    update(b);
    return b;
  }
}
//inserts middle in left half
static Treap[] split(Treap t, int x) {
  if (t == null) return new Treap[2];
  if (t.v <= x) {
    Treap[] p = split(t.R, x);
    t.R = p[0];
    p[0] = t;
    return p;
  } else {
    Treap[] p = split(t.L, x);
    t.L = p[1];
    p[1] = t;
    return p;
  }
}
//use only with split
static Treap insert(Treap t, int x) {
  Treap m = new Treap();
  m.v = x;
  m.y = Math.random();
  m.sz = 1;
  Treap[] p = splitK(t, x-1);
  return merge(merge(p[0],m), p[1]);
}


//inserts middle in left half
static Treap[] splitK(Treap t, int x) {
  if (t == null) return new Treap[2];
  if (t.sz < x) return new Treap[]{t, null};
  if (sz(t.L) >= x) {
    Treap[] p = splitK(t.L, x);
    t.L = p[1];
    p[1] = t;
    update(p[0]);
    update(p[1]);
    return p;
  } else if (sz(t.L) + 1 == x){
    Treap r = t.R;
    t.R = null;
    Treap[] p = new Treap[]{t, r};
    update(p[0]);
    update(p[1]);
    return p;
  } else {
    Treap[] p = splitK(t.R, x - sz(t.L)-1);
    t.R = p[0];
    p[0] = t;
    update(p[0]);
    update(p[1]);
    return p;
  }
}
//use only with splitK
static Treap insertK(Treap t, int w, int x) {
  Treap m = new Treap();
  m.v = x;
  m.y = Math.random();
```

```
    m.sz = 1;
    Treap[] p = splitK(t, w);
    t = merge(p[0], m);
    return merge(t, p[1]);
  }
  //use only with splitK
  static Treap deleteK(Treap t, int w, int x) {
    Treap[] p = splitK(t, w);
    Treap[] q = splitK(p[0], w-1);
    return merge(q[0], p[1]);
  }

  static Treap Left(Treap t) {
    if (t == null) return null;
    if (t.L == null) return t;
    return Left(t.L);
  }
  static Treap Right(Treap t) {
    if (t == null) return null;
    if (t.R == null) return t;
    return Right(t.R);
  }

}
```

4.7. **RMQ.** $\mathcal{O}(1)$ queries of interval min, max, gcd or lcm. $\mathcal{O}(n \log n)$ building time.

```
import math
class RMQ:
    def __init__(self, arr, func=min):
        self.sz = len(arr)
        self.func = func
        MAXN = self.sz
        LOGMAXN = int(math.ceil(math.log(MAXN + 1, 2)))
        self.data = [[0]*LOGMAXN for _ in range(MAXN)]
        for i in range(MAXN):
            self.data[i][0] = arr[i]
        for j in range(1, LOGMAXN):
            for i in range(MAXN - (1<<j)+1):
                self.data[i][j] = func(self.data[i][j-1],
                        self.data[i + (1<<(j-1))][j-1])

    def query(self, a, b):
        if a > b:
            # some default value when query is empty
            return 1
        d = b - a + 1
        k = int(math.log(d, 2))
        return self.func(self.data[a][k], self.data[b-(1<<k)+1][k])
```

5. Graph Algorithms

5.1. **Dijkstras algorithm.** Finds the shortest distance between two Nodes in a weighted graph in $\mathcal{O}(|E| \log |V|)$ time.

```
from heapq import heappop as pop, heappush as push
# adj: adj-list where edges are tuples (node_id, weight):
# (1) --2-- (0) --3-- (2) has the adj-list:
# adj = [[(1, 2), (2, 3)], [(0, 2)], [0, 3]]
def dijk(adj, S, T):
    N = len(adj)
    INF = 10**18
    dist = [INF]*N
    pq = []
    def add(i, dst):
        if dst < dist[i]:
            dist[i] = dst
            push(pq, (dst, i))
    add(S, 0)

    while pq:
        D, i = pop(pq)
        if i == T: return D
        if D != dist[i]: continue
        for j, w in adj[i]:
            add(j, D + w)

    return dist[T]
```

5.2. **Hopcroft-Karp.** The Hopcroft-Karp algorithm finds the maximal matching in a bipartite graph. Also, this matching can together with Köning's theorem be used to construct a minimal vertex-cover, which as we all know is the complement of a maximum independent set. Runs in $\mathcal{O}(|E|\sqrt{|V|})$.

```python
# Hopcroft-Karp bipartite max-cardinality matching and max independent set
# David Eppstein, UC Irvine, 27 Apr 2002
# Used in https://open.kattis.com/problems/cuckoo
def bipartiteMatch(graph):
    '''Find maximum cardinality matching of a bipartite graph (U,V,E).
    The input format is a dictionary mapping members of U to a list
    of their neighbors in V.  The output is a triple (M,A,B) where M is a
    dictionary mapping members of V to their matches in U, A is the part
    of the maximum independent set in U, and B is the part of the MIS in V.
    The same object may occur in both U and V, and is treated as two
    distinct vertices if this happens.'''

    # initialize greedy matching (redundant, but faster than full search)
    matching = {}
    for u in graph:
        for v in graph[u]:
            if v not in matching:
                matching[v] = u
                break

    while 1:
        # structure residual graph into layers
        # pred[u] gives the neighbor in the previous layer for u in U
        # preds[v] gives a list of neighbors in the previous layer for v in V
        # unmatched gives a list of unmatched vertices in final layer of V,
        # and is also used as a flag value for pred[u] when u is in the first layer
        preds = {}
        unmatched = []
        pred = dict([(u,unmatched) for u in graph])
        for v in matching:
            del pred[matching[v]]
        layer = list(pred)

        # repeatedly extend layering structure by another pair of layers
        while layer and not unmatched:
            newLayer = {}
            for u in layer:
                for v in graph[u]:
                    if v not in preds:
                        newLayer.setdefault(v,[]).append(u)
            layer = []
            for v in newLayer:
                preds[v] = newLayer[v]
                if v in matching:
                    layer.append(matching[v])
                    pred[matching[v]] = v
                else:
                    unmatched.append(v)

        # did we finish layering without finding any alternating paths?
        if not unmatched:
            unlayered = {}
            for u in graph:
                for v in graph[u]:
                    if v not in preds:
                        unlayered[v] = None
            return (matching,list(pred),list(unlayered))

        # recursively search backward through layers to find alternating paths
        # recursion returns true if found path, false otherwise
        def recurse(v):
            if v in preds:
                L = preds[v]
                del preds[v]
                for u in L:
                    if u in pred:
                        pu = pred[u]
                        del pred[u]
                        if pu is unmatched or recurse(pu):
                            matching[v] = u
                            return 1
            return 0

        for v in unmatched: recurse(v)
```

5.3. **Network Flow.** Ford-Fulkerson algorithm for determining the maximum flow through a graph can be used for a lot of unexpected problems. Given a problem that can be formulated as a graph, where no ideas are found trying, it might help trying to apply network flow. The running time is $\mathcal{O}(C \cdot m)$ where C is the maximum flow and m is the amount of edges in the graph. If C is very large we can change the running time to $\mathcal{O}(\log Cm^2)$ by only studying edges with a large enough capacity in the beginning.

```python
# used in mincut @ Kattis
from collections import defaultdict
class Flow:
    def __init__(self, sz):
        self.G = [
            defaultdict(int) for _ in range(sz)
        ] # neighbourhood dict, N[u] = {v_1: cap_1, v_2: cap_2, ...}
        self.Seen = set() # redundant

    def increase_capacity(self, u, v, cap):
        """ Increases capacity on edge (u, v) with cap.
            No need to add the edge """
        self.G[u][v] += cap

    def max_flow(self, source, sink):
        def dfs(u, hi):
            G = self.G
            Seen = self.Seen
            if u in Seen: return 0
            if u == sink: return hi

            Seen.add(u)
            for v, cap in G[u].items():
                if cap >= self.min_edge:
                    f = dfs(v, min(hi, cap))
                    if f:
                        G[u][v] -= f
                        G[v][u] += f
                        return f
            return 0

        flow = 0
        self.min_edge = 2**30 # minimal edge allowed
        while self.min_edge > 0:
            self.Seen = set()
            pushed = dfs(source, float('inf'))
            if not pushed:
                self.min_edge //= 2
            flow += pushed
        return flow
```

5.4. **Dinitz Algorithm.** Faster flow algorithm.

```python
from collections import defaultdict
class Dinitz:
    def __init__(self, sz, INF=10**10):
        self.G = [defaultdict(int) for _ in range(sz)]
        self.sz = sz
        self.INF = INF

    def add_edge(self, i, j, w):
        self.G[i][j] += w

    def bfs(self, s, t):
        level = [0]*self.sz
        q = [s]
        level[s] = 1
        while q:
            q2 = []
            for u in q:
                for v, w in self.G[u].items():
                    if w and level[v] == 0:
                        level[v] = level[u] + 1
                        q2.append(v)
            q = q2
        self.level = level
        return level[t] != 0

    def dfs(self, s, t, FLOW):
        if s in self.V: return 0
        if s == t: return FLOW
        self.V.add(s)
        L = self.level[s]
        for u, w in self.G[s].items():
            if u in self.dead: continue
            if w and L+1==self.level[u]:
                F = self.dfs(u, t, min(FLOW, w))
```

```
                if F:
                    self.G[s][u] -= F
                    self.G[u][s] += F
                    return F
        self.dead.add(s)
        return 0


    def max_flow(self, s, t):
        flow = 0
        while self.bfs(s, t):
            self.dead = set()
            while True:
                self.V = set()
                pushed = self.dfs(s, t, self.INF)
                if not pushed: break
                flow += pushed
        return 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);
        }
      }
    }
    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;
  }
};
```

5.5. **Min Cost Max Flow.** Finds the minimal cost of a maximum flow through a graph. Can be used for some optimization problems where the optimal assignment needs to be a maximum flow.

```
class MinCostMaxFlow {
boolean found[];
int N, dad[];
long cap[][], flow[][], cost[][], dist[], pi[];

static final long INF = Long.MAX_VALUE / 2 - 1;

boolean search(int s, int t) {
Arrays.fill(found, false);
Arrays.fill(dist, INF);
dist[s] = 0;

while (s != N) {
  int best = N;
  found[s] = true;
  for (int k = 0; k < N; k++) {
    if (found[k]) continue;
    if (flow[k][s] != 0) {
      long val = dist[s] + pi[s] - pi[k] - cost[k][s];
      if (dist[k] > val) {
        dist[k] = val;
        dad[k] = s;
      }
    }
    if (flow[s][k] < cap[s][k]) {
      long val = dist[s] + pi[s] - pi[k] + cost[s][k];
      if (dist[k] > val) {
        dist[k] = val;
        dad[k] = s;
      }
    }

    if (dist[k] < dist[best]) best = k;
  }
  s = best;
}
for (int k = 0; k < N; k++)
  pi[k] = Math.min(pi[k] + dist[k], INF);
return found[t];
}

long[] mcmf(long c[][], long d[][], int s, int t) {
cap = c;
cost = d;

N = cap.length;
found = new boolean[N];
```

```
flow = new long[N][N];
dist = new long[N+1];
dad = new int[N];
pi = new long[N];

long totflow = 0, totcost = 0;
while (search(s, t)) {
  long amt = INF;
  for (int x = t; x != s; x = dad[x])
    amt = Math.min(amt, flow[x][dad[x]] != 0 ?
    flow[x][dad[x]] : cap[dad[x]][x] - flow[dad[x]][x]);
  for (int x = t; x != s; x = dad[x]) {
    if (flow[x][dad[x]] != 0) {
      flow[x][dad[x]] -= amt;
      totcost -= amt * cost[x][dad[x]];
    } else {
      flow[dad[x]][x] += amt;
      totcost += amt * cost[dad[x]][x];
    }
  }
  totflow += amt;
}

return new long[]{ totflow, totcost };
}
}
```

5.6. **2-Sat.** Solves 2sat by splitting up vertices in strongly connected components.

```
# used in sevenkingdoms, illumination
import sys
sys.setrecursionlimit(10**5)
class Sat:
    def __init__(self, no_vars):
        self.size = no_vars*2
        self.no_vars = no_vars
        self.adj = [[] for _ in range(self.size)]
        self.back = [[] for _ in range(self.size)]
    def add_imply(self, i, j):
        self.adj[i].append(j)
        self.back[j].append(i)
    def add_or(self, i, j):
        self.add_imply(i^1, j)
        self.add_imply(j^1, i)
    def add_xor(self, i, j):
        self.add_or(i, j)
        self.add_or(i^1, j^1)
    def add_eq(self, i, j):
        self.add_xor(i, j^1)

    def dfs1(self, i):
        if i in self.marked: return
        self.marked.add(i)
        for j in self.adj[i]:
            self.dfs1(j)
        self.stack.append(i)

    def dfs2(self, i):
        if i in self.marked: return
        self.marked.add(i)
        for j in self.back[i]:
            self.dfs2(j)
        self.comp[i] = self.no_c

    def is_sat(self):
        self.marked = set()
        self.stack = []
        for i in range(self.size):
            self.dfs1(i)
        self.marked = set()
        self.no_c = 0
        self.comp = [0]*self.size
        while self.stack:
            i = self.stack.pop()
            if i not in self.marked:
                self.no_c += 1
                self.dfs2(i)
        for i in range(self.no_vars):
            if self.comp[i*2] == self.comp[i*2+1]:
                return False
        return True
```

```
    # assumes is_sat.
    # If ¬xi is after xi in topological sort,
    # xi should be FALSE. It should be TRUE otherwise.
    # https://codeforces.com/blog/entry/16205
    def solution(self):
        V = []
        for i in range(self.no_vars):
            V.append(self.comp[i*2] > self.comp[i*2^1])
        return V

if __name__ == '__main__':
    S = Sat(1)
    S.add_or(0, 0)
    print(S.is_sat())
    print(S.solution())
```

5.7. **Hungarian - Min Cost Max Bipartite Matching.** The Hungarian algorithm runs in $\mathcal{O}(n^3)$ with a low constant, giving us the minimum cost matching. If the maximum cost is wanted you can just negate the weights.

```
# used on https://open.kattis.com/problems/arboriculture
# G is Bipartite graph N x M (N <= M) where [i][j] is cost to match L[i] and R[j]
# Ported from: https://raw.githubusercontent.com/kth-competitive-programming/kactl/main/content/graph/WeightedMatching.h
# Description: Given a weighted bipartite graph, matches every node on
# the left with a node on the right such that no
# nodes are in two matchings and the sum of the edge weights is minimal. Takes
# cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and
# Returns: (min cost, match), where L[i] is matched with R[match[i]].
# Negate costs for max cost.
# Time: O(N^2M)
#
def hungarian(G):
    INF = 10**18
    if len(G) == 0:
        return 0, []

    n, m = len(G) + 1, len(G[0]) + 1
    u, v, p = [0]*n, [0]*m, [0]*m
    ans = [0]*(n-1)
    for i in range(1, n):
        p[0], j0 = i, 0
        dist, pre = [INF]*m, [-1]*m
        done = [False]*(m+1)
        while True:
            done[j0] = True
            i0, j1, delta = p[j0], 0, INF
            for j in range(1, m):
                if done[j]: continue
                cur = G[i0 - 1][j-1] - u[i0] - v[j]
                if cur < dist[j]:
                    dist[j], pre[j] = cur, j0
                if dist[j] < delta:
                    delta, j1 = dist[j], j
            for j in range(0, m):
                if done[j]:
                    u[p[j]] += delta
                    v[j] -= delta
                else:
                    dist[j] -= delta
            j0 = j1
            if p[j0] == 0: break
        while j0:
            j1 = pre[j0]
            p[j0] = p[j1]
            j0 = j1
    return -v[0], ans
```

6. Dynamic Programming

6.1. **Longest Increasing Subsequence.** Finds the longest increasing subsequence in an array in $\mathcal{O}(n \log n)$ time. Can easily be transformed to longest decreasing/non decreasing/non increasing subsequence.

```
def lis(X):
    N = len(X)
    P = [0]*N
    M = [0]*(N+1)
    L = 0
    for i in range(N):
```

```
        lo, hi = 1, L + 1
        while lo < hi:
            mid = (lo + hi) >> 1
            if X[M[mid]] < X[i]:
                lo = mid + 1
            else:
                hi = mid
        newL = lo
        P[i] = M[newL - 1]
        M[newL] = i
        L = max(L, newL)
    S = [0]*L
    k = M[L]
    for i in range(L-1, -1, -1):
        S[i] = X[k]
        k = P[k]
    return S
```

6.2. **String functions.** The z-function computes the longest common prefix of t and $t[i:]$ for each i in $\mathcal{O}(|t|)$. The border function computes the longest common proper (smaller than whole string) prefix and suffix of string $t[:i]$.

```
def zfun(t):
    z = [0]*len(t)
    n = len(t)
    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 t[i+z[i]] == t[z[i]]:
            z[i]+=1
        if i + z[i] - 1 > r:
            l = i
            r = i + z[i] - 1
    return z

def matches(t, p):
    s = p + '#' + t
    return filter(lambda x: x[1] == len(p),
            enumerate(zfun(s)))
```

```
def boarders(s):
    b = [0]*len(s)
    for i in range(1, len(s)):
        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
```

6.3. **Josephus problem.** Who is the last one to get removed from a circle if the k'th element is continuously removed?

```
# Rewritten from J(n, k) = (J(n-1, k) + k)%n
def J(n, k):
    r = 0
    for i in range(2, n+1):
        r = (r + k)%i
    return r
```

6.4. **Floyd Warshall.** Constructs a matrix with the distance between all pairs of nodes in $\mathcal{O}(n^3)$ time. Works for negative edge weights, but not if there exists negative cycles. The nxt matrix is used to reconstruct a path. Can be skipped if we don't care about the path.

```
# Computes distance matrix and next matrix given an edgelist
def FloydWarshall(n, edges):
    INF = 10**9
    dist = [[INF]*n for _ in range(n)]
    nxt = [[None]*n for _ in range(n)]
    for e in edges:
        dist[e[0]][e[1]] = e[2]
        nxt[e[0]][e[1]] = e[1]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    nxt[i][j] = nxt[i][k]
    return dist, nxt

# Computes the path from i to j given a nextmatrix
```

```python
def path(i, j, nxt):
    if nxt[i][j] == None: return []
    path = [i]
    while i != j:
        i = nxt[i][j]
        path.append(i)
    return path
```

7. Etc

7.1. System of Equations. Solves the system of equations $\boldsymbol{Ax} = \boldsymbol{b}$ by Gaussian elimination. This can for example be used to determine the expected value of each node in a markov chain. Runns in $\mathcal{O}(N^3)$.

```python
# monoid needs to implement
# __add__, __mul__, __sub__, __div__ and isZ
def gauss(A, b, monoid=None):
    def Z(v): return abs(v) < 1e-6 if not monoid else v.isZ()

    N = len(A[0])
    for i in range(N):
        try:
            m = next(j for j in range(i, N) if Z(A[j][i]) == False)
        except:
            return None #A is not independent!
        if i != m:
            A[i], A[m] = A[m], A[i]
            b[i], b[m] = b[m], b[i]
        for j in range(i+1, N):
            sub = A[j][i]/A[i][i]
            b[j] -= sub*b[i]
            for k in range(N):
                A[j][k] -= sub*A[i][k]

    for i in range(N-1, -1, -1):
        for j in range(N-1, i, -1):
            sub = A[i][j]/A[j][j]
            b[i] -= sub*b[j]
        b[i], A[i][i] = b[i]/A[i][i], A[i][i]/A[i][i]
    return b
```

7.2. Convex Hull. From a collection of points in the plane the convex hull is often used to compute the largest distance or the area covered, or the length of a rope that encloses the points. It can be found in $\mathcal{O}(N \log N)$ time by sorting the points on angle and the sweeping over all of them.

```python
def convex_hull(pts):
    pts = sorted(set(pts))

    if len(pts) <= 2:
        return pts

    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    lo = []
    for p in pts:
        while len(lo) >= 2 and cross(lo[-2], lo[-1], p) <= 0:
            lo.pop()
        lo.append(p)

    hi = []
    for p in reversed(pts):
        while len(hi) >= 2 and cross(hi[-2], hi[-1], p) <= 0:
            hi.pop()
        hi.append(p)

    return lo[:-1] + hi[:-1]
```

7.3. Number Theory.

```python
import math

# Evaluates to n! / (k! * (n - k)!) when k <= n and evaluates to zero when k > n.
# math.comb(n, k) #introduced in python3.8

# math.gcd(a, b)
def gcd(a, b):
    return b if a%b == 0 else gcd(b, a%b)

# returns b where (a*b)%MOD == 1
def inv(a, MOD):
    return pow(a, -1, MOD)
```

```
# returns g = gcd(a, b), x0, y0,
# where g = x0*a + y0*b
def xgcd(a, b):
    x0, x1, y0, y1 = 1, 0, 0, 1
    while b != 0:
        q, a, b = (a // b, b, a % b)
        x0, x1 = (x1, x0 - q * x1)
        y0, y1 = (y1, y0 - q * y1)
    return (a, x0, y0)

def crt(la, ln):
    assert len(la) == len(ln)
    for i in range(len(la)):
        assert 0 <= la[i] < ln[i]
    prod = 1
    for n in ln:
        assert gcd(prod, n) == 1
        prod *= n
    lN = []
    for n in ln:
        lN.append(prod//n)
    x = 0
    for i, a in enumerate(la):
        print(lN[i], ln[i])
        _, Mi, mi = xgcd(lN[i], ln[i])
        x += a*Mi*lN[i]
    return x % prod

# finds x^e mod m
# Or just pow(x, e, m)
def modpow(x, m, e):
    res = 1
    while e:
        if e%2 == 1:
            res = (res*x) % m
        x = (x*x) % m
        e = e//2
    return res
```

```
# Divides a list of digits with an int.
# A lot faster than using bigint-division.
def div(L, d):
    r = [0]*(len(L) + 1)
    q = [0]*len(L)
    for i in range(len(L)):
        x = int(L[i]) + r[i]*10
        q[i] = x//d
        r[i+1] = x-q[i]*d
    s = []
    for i in range(len(L) - 1, 0, -1):
        s.append(q[i]%10)
        q[i-1] += q[i]//10

    while q[0]:
        s.append(q[0]%10)
        q[0] = q[0]//10
    s = s[::-1]
    i = 0
    while s[i] == 0:
        i += 1
    return s[i:]

# Multiplies a list of digits with an int.
# A lot faster than using bigint-multiplication.
def mul(L, d):
    r = [d*x for x in L]
    s = []
    for i in range(len(r) - 1, 0, -1):
        s.append(r[i]%10)
        r[i-1] += r[i]//10
    while r[0]:
        s.append(r[0]%10)
        r[0] = r[0]//10
    return s[::-1]


large_primes = [
5915587277,
1500450271,
```

```
3267000013,
5754853343,
4093082899,
9576890767,
3628273133,
2860486313,
5463458053,
3367900313,
10000000000000061,
10**16 + 61,
10**17 + 3
]

def getPrimesBelow(N):
    primes = []
    soll = [1]*N
    for p in range(2, N):
        if soll[p]:
            primes.append(p)
            for k in range(p*p, N, p):
                soll[k] = 0
    return primes

def isPrime(N):
    if N < 2: return False
    if N%2 == 0: return N == 2
    mx = min(int(N**.5) + 2, N)
    for i in range(3, mx, 2):
        if N % i == 0: return False
    return True

def genPrimesFrom(N):
    while True:
        if isPrime(N):
            yield N
        N += 1

def getPrimesFrom(N, cnt):
    itr = genPrimesFrom(N)
    return [next(itr) for _ in range(cnt)]
```

7.4. **FFT.** FFT can be used to calculate the product of two polynomials of length N in $\mathcal{O}(N \log N)$ time. The FFT function requires a power of 2 sized array of size at least 2N to store the results as complex numbers.

```
import cmath
# A has to be of length a power of 2.

def FFT(A, inverse=False):
    N = len(A)
    if N <= 1:
        return A
    if inverse:
        D = FFT(A) # d_0/N, d_{N-1}/N, d_{N-2}/N, ...
        return map(lambda x: x/N, [D[0]] + D[:0:-1])
    evn = FFT(A[0::2])
    odd = FFT(A[1::2])
    Nh = N//2
    return [evn[k%Nh]+cmath.exp(2j*cmath.pi*k/N)*odd[k%Nh]
            for k in range(N)]

# A has to be of length a power of 2.
def FFT2(a, inverse=False):
    N = len(a)
    j = 0
    for i in range(1, N):
        bit = N>>1
        while j&bit:
            j ^= bit
            bit >>= 1
        j^= bit
        if i < j:
            a[i], a[j] = a[j], a[i]

    L = 2
    MUL = -1 if inverse else 1
    while L <= N:
        ang = 2j*cmath.pi/L * MUL
        wlen = cmath.exp(ang)
        for i in range(0, N, L):
            w = 1
            for j in range(L//2):
```

```
                u = a[i+j]
                v = a[i+j+L//2] * w
                a[i+j] = u + v
                a[i+j+L//2] = u - v
                w *= wlen
            L *= 2
    if inverse:
        for i in range(N):
            a[i] /= N
    return a

def uP(n):
    while n != (n&-n):
        n += n&-n
    return n

# C[x] = sum_{i=0..N}(A[x-i]*B[i])
def polymul(A, B):
    sz = 2*max(uP(len(A)), uP(len(B)))
    A = A + [0]*(sz - len(A))
    B = B + [0]*(sz - len(B))
    fA = FFT(A)
    fB = FFT(B)
    fAB = [a*b for a, b in zip(fA, fB)]
    C = [x.real for x in FFT(fAB, True)]
    return C
```

8. NP tricks

8.1. **MaxClique.** The max clique problem is one of Karp's 21 NP-complete problems. The problem is to find the largest subset of an undirected graph that forms a clique - a complete graph. There is an obvious algorithm that just inspects every subset of the graph and determines if this subset is a clique. This algorithm runs in $\mathcal{O}(n^2 2^n)$. However one can use the meet in the middle trick (one step divide and conquer) and reduce the complexity to $\mathcal{O}(n^2 2^{\frac{n}{2}})$.

```
static int max_clique(int n, int[][] adj) {
  int fst = n/2;
  int snd = n - fst;
  int[] maxc = new int[1<<fst];
  int max = 1;
  for(int i = 0; i<(1<<fst); i++) {
    for(int a = 0; a<fst; a++) {
      if((i&1<<a) != 0)
        maxc[i] = Math.max(maxc[i], maxc[i^(1<<a)]);
    }
    boolean ok = true;
    for(int a = 0; a<fst; a++) if((i&1<<a) != 0) {
      for(int b = a+1; b<fst; b++) {
          if((i&1<<b) != 0 && adj[a][b] == 0)
              ok = false;
      }
    }
    if(ok) {
      maxc[i] = Integer.bitCount(i);
      max = Math.max(max, maxc[i]);
    }
  }
  for(int i = 0; i<(1<<snd); i++) {
    boolean ok = true;
    for(int a = 0; a<snd; a++) if((i&1<<a) != 0) {
      for(int b = a+1; b<snd; b++) {
        if((i&1<<b) != 0)
          if(adj[a+fst][b+fst] == 0)
            ok = false;
      }
    }
    if(!ok) continue;
    int mask = 0;
    for(int a = 0; a<fst; a++) {
      ok = true;
      for(int b = 0; b<snd; b++) {
        if((i&1<<b) != 0) {
          if(adj[a][b+fst] == 0) ok = false;
        }
      }
      if(ok) mask |= (1<<a);
    }
```

```java
    max = Math.max(Integer.bitCount(i) + maxc[mask],
            max);
  }
  return max;
}
```

9. Coordinate Geometry

9.1. Area of a nonintersecting polygon. The signed area of a polygon with n vertices is given by

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$

9.2. Intersection of two lines. Two lines defined by

$$a_1 x + b_1 y + c_1 = 0$$
$$a_2 x + b_2 y + c_2 = 0$$

Intersects in the point

$$P = (\frac{b_1 c_2 - b_2 c_1}{w}, \frac{a_2 c_1 - a_1 c_2}{w}),$$

where $w = a_1 b_2 - a_2 b_1$. If $w = 0$ the lines are parallel.

9.3. Distance between line segment and point. Given a line segment between point P, Q, the distance D to point R is given by:

$$a = Q_y - P_y$$
$$b = Q_x - P_x$$
$$c = P_x Q_y - P_y Q_x$$
$$R_P = (\frac{b(bR_x - aR_y) - ac}{a^2 + b^2}, \frac{a(aR_y - bR_x) - bc}{a^2 + b^2})$$
$$D = \begin{cases} \frac{|aR_x + bR_y + c|}{\sqrt{a^2+b^2}} & \text{if } (R_{P_x} - P_x)(R_{P_x} - Q_x) < 0, \\ \min |P - R|, |Q - R| & \text{otherwise} \end{cases}$$

9.4. Picks theorem. Find the amount of internal integer coordinates i inside a polygon with picks theorem $A = \frac{b}{2} + i - 1$, where A is the area of the polygon and b is the amount of coordinates on the boundary.

9.5. Trigonometry. Sine-rule

$$\frac{\sin(\alpha)}{a} = \frac{\sin(\beta)}{b} = \frac{\sin(\gamma)}{c}$$

Cosine-rule

$$a^2 = b^2 + c^2 - 2bc \cdot \cos(\alpha)$$

Area-rule

$$A = \frac{a \cdot b \cdot \sin(\gamma)}{2}$$

Rotation Matrix, rotate a 2D-vector θ radians by multiplying with the following matrix.

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

9.6. Implementations.

```python
import math

# Distance between two points
def dist(p, q):
    return math.hypot(p[0]-q[0], p[1] - q[1])

# Square distance between two points
def d2(p, q):
    return (p[0] - q[0])**2 + (p[1] - q[1])**2

# Converts two points to a line (a, b, c),
# ax + by + c = 0
# if p == q, a = b = c = 0
def pts2line(p, q):
    return (-q[1] + p[1],
          q[0] - p[0],
          p[0]*q[1] - p[1]*q[0])

# Distance from a point to a line,
# given that a != 0 or b != 0
def distl(l, p):
    return (abs(l[0]*p[0] + l[1]*p[1] + l[2])
      /math.hypot(l[0], l[1]))

# intersects two lines.
# if parallell, returnes False.
```

```python
# lines on format (a, b, c) where ax + by + c == 0
def line_intersection(l1, l2):
    a1,b1,c1 = l1
    a2,b2,c2 = l2
    cp = a1*b2 - a2*b1
    if cp != 0:
        return float(b1*c2 - b2*c1)/cp, float(a2*c1 - a1*c2)/cp
    else:
        return False

# projects a point on a line
def project(l, p):
    a, b, c = l
    return ((b*(b*p[0] - a*p[1]) - a*c)/(a*a + b*b),
        (a*(a*p[1] - b*p[0]) - b*c)/(a*a + b*b))

# Intersections between circles
def circle_intersection(c1, c2):
    if c1[2] > c2[2]:
        c1, c2 = c2, c1
    x1, y1, r1 = c1
    x2, y2, r2 = c2
    if x1 == x2 and y1 == y2 and r1 == r2:
        return False

    dist2 = (x1 - x2)*(x1-x2) + (y1 - y2)*(y1 - y2)
    rsq = (r1 + r2)*(r1 + r2)
    if dist2 > rsq or dist2 < (r1-r2)*(r1-r2):
        return []
    elif dist2 == rsq:
        cx = x1 + (x2-x1)*r1/(r1+r2)
        cy = y1 + (y2-y1)*r1/(r1+r2)
        return [(cx, cy)]
    elif dist2 == (r1-r2)*(r1-r2):
        cx = x1 - (x2-x1)*r1/(r2-r1)
        cy = y1 - (y2-y1)*r1/(r2-r1)
        return [(cx, cy)]

    d = math.sqrt(dist2)
    f = (r1*r1 - r2*r2 + dist2)/(2*dist2)
    xf = x1 + f*(x2-x1)
    yf = y1 + f*(y2-y1)
    dx = xf-x1
    dy = yf-y1
    h = math.sqrt(r1*r1 - dx*dx - dy*dy)
    norm = abs(math.hypot(dx, dy))
    p1 = (xf + h*(-dy)/norm, yf + h*(dx)/norm)
    p2 = (xf + h*(dy)/norm, yf + h*(-dx)/norm)
    return sorted([p1, p2])

# Finds the bisector through origo
# between two points by normalizing.
def bisector(p1, p2):
    d1 = math.hypot(p1[0], p2[1])
    d2 = math.hypot(p2[0], p2[1])
    return ((p1[0]/d1 + p2[0]/d2),
          (p1[1]/d1 + p2[1]/d2))

# Distance from P to origo
def norm(P):
    return (P[0]**2 + P[1]**2 + P[2]**2)**(0.5)

# Finds ditance between point p
# and line A + t*u in 3D
def dist3D(A, u, p):
    AP = tuple(A[i] - p[i] for i in range(3))
    cross = tuple(AP[i]*u[(i+1)%3] - AP[(i+1)%3]*u[i]
        for i in range(3))
    return norm(cross)/norm(u)

def vec(p1, p2):
    return p2[0]-p1[0], p2[1] - p1[1]

def sign(x):
    if x < 0: return -1
    return 1 if x > 0 else 0

def cross(u, v):
    return u[0] * v[1] - u[1] * v[0]
```

```python
# s1: (Point, Point)
# s2: (Point, Point)
# Point : (x, y)
# returns true if intersecting s1 & s2 shares at least 1 point.
def is_segment_intersection(s1, s2):
    u = vec(*s1)
    v = vec(*s2)
    p1, p2 = s1
    q1, q2 = s2
    d1 = cross(u, vec(p1, q1))
    d2 = cross(u, vec(p1, q2))
    d3 = cross(v, vec(q1, p1))
    d4 = cross(v, vec(q1, p2))
    if d1 * d2 * d3 * d4 == 0:
        return True
    return sign(d1) != sign(d2) and sign(d3) != sign(d4)
```

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 clarification.
- Print a file.
- Directory with test cases.