# Team notebook

November 11, 2016

## Contents



## 1 BellmanFord

```
// This function runs the Bellman-Ford algorithm for single source
// shortest paths with negative edge weights. The function returns
// false if a negative weight cycle is detected. Otherwise, the
// function returns true and dist[i] is the length of the shortest
// path from start to i.
//
// Running time: O(|V|^3)
//
```

```
//   INPUT:  start, w[i][j] = cost of edge from i to j
//   OUTPUT: dist[i] = min weight path from start to i
//           prev[i] = previous node on the best path from the
//                     start node

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

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

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

bool BellmanFord (const VVT &w, VT &dist, VI &prev, int start){
  int n = w.size();
  prev = VI(n, -1);
  dist = VT(n, 1000000000);
  dist[start] = 0;

  for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if (dist[j] > dist[i] + w[i][j]){
          if (k == n-1) return false;
          dist[j] = dist[i] + w[i][j];
          prev[j] = i;
        }
      }
    }
  }

  return true;
}
```

# 2 ConvexHull

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

// Compute the 2D convex hull of a set of points using the monotone chain
// algorithm. Eliminate redundant points from the hull if
    REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
//   INPUT:  a vector of input points, unordered.
//   OUTPUT: a vector of points in the convex hull, counterclockwise,
    starting
//           with bottommost/leftmost point

#define REMOVE_REDUNDANT

typedef double T;
const T EPS = 1e-7;
struct PT {
  T x, y;
  PT() {}
  PT(T x, T y) : x(x), y(y) {}
  bool operator<(const PT &rhs) const { return make_pair(y,x) <
      make_pair(rhs.y,rhs.x); }
  bool operator==(const PT &rhs) const { return make_pair(y,x) ==
      make_pair(rhs.y,rhs.x); }
};

T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
  return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 &&
      (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  vector<PT> up, dn;
  for (int i = 0; i < pts.size(); i++) {
```

```
    while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >=
        0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <=
        0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
#endif
}

int main()
{
  vector<PT> pts(10);

  pts[0] = (PT(0,0));
  pts[1] = (PT(1,5));
  pts[2] = (PT(-4,7));
  pts[3] = (PT(8,9));
  pts[4] = (PT(20,20));
  pts[5] = (PT(1,1));
  pts[6] = (PT(-16,4));
  pts[7] = (PT(0,-20));
  pts[8] = (PT(30,45));
  pts[9] = (PT(20,0));

  ConvexHull(pts);

  for(PT pt:pts)
  {
    cout << pt.x << " " << pt.y << endl;
  }

  /*
  should print:
  0 -20
  20 0
  30 45
  -16 4
  */


  return 0;
}
```

# 3 Dijkstra

```
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)

#include <queue>
#include <stdio.h>

using namespace std;
const int INF = 2000000000;
typedef pair<int,int> PII;

int main(){

  int N, s, t;
  scanf ("%d%d%d", &N, &s, &t);
  vector<vector<PII> > edges(N);
  for (int i = 0; i < N; i++){
    int M;
    scanf ("%d", &M);
    for (int j = 0; j < M; j++){
      int vertex, dist;
      scanf ("%d%d", &vertex, &dist);
```

```
      edges[i].push_back (make_pair (dist, vertex)); // note order of
          arguments here
    }
  }

  // use priority queue in which top element has the "smallest" priority
  priority_queue<PII, vector<PII>, greater<PII> > Q;
  vector<int> dist(N, INF), dad(N, -1);
  Q.push (make_pair (0, s));
  dist[s] = 0;
  while (!Q.empty()){
    PII p = Q.top();
    if (p.second == t) break;
    Q.pop();

    int here = p.second;
    for (vector<PII>::iterator it=edges[here].begin();
        it!=edges[here].end(); it++){
      if (dist[here] + it->first < dist[it->second]){
        dist[it->second] = dist[here] + it->first;
        dad[it->second] = here;
        Q.push (make_pair (dist[it->second], it->second));
      }
    }
  }

  printf ("%d\n", dist[t]);
  if (dist[t] < INF)
    for(int i=t;i!=-1;i=dad[i])
      printf ("%d%c", i, (i==s?'\n':' '));

  return 0;
}
```

# 4 Dinic

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source and sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain actual flow values, look at edges with capacity > 0
//       (zero capacity edges are residual edges).

#include <iostream>
#include <vector>

using namespace std;
typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
  LL rcap() { return cap - flow; }
};

struct Dinic {
  int N;
  vector<vector<Edge> > G;
  vector<vector<Edge *> > Lf;
  vector<int> layer;
  vector<int> Q;

  Dinic(int N) : N(N), G(N), Q(N) {}

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

  LL BlockingFlow(int s, int t) {
    layer.clear(); layer.resize(N, -1);
    layer[s] = 0;
    Lf.clear(); Lf.resize(N);

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) {
```

```
    int x = Q[head++];
    for (int i = 0; i < G[x].size(); i++) {
      Edge &e = G[x][i]; if (e.rcap() <= 0) continue;
      if (layer[e.to] == -1) {
        layer[e.to] = layer[e.from] + 1;
        Q[tail++] = e.to;
      }
      if (layer[e.to] > layer[e.from]) {
        Lf[e.from].push_back(&e);
      }
    }
  }
  if (layer[t] == -1) return 0;

  LL totflow = 0;
  vector<Edge *> P;
  while (!Lf[s].empty()) {
    int curr = P.empty() ? s : P.back()->to;
    if (curr == t) { // Augment
      LL amt = P.front()->rcap();
      for (int i = 0; i < P.size(); ++i) {
        amt = min(amt, P[i]->rcap());
      }
      totflow += amt;
      for (int i = P.size() - 1; i >= 0; --i) {
        P[i]->flow += amt;
        G[P[i]->to][P[i]->index].flow -= amt;
        if (P[i]->rcap() <= 0) {
          Lf[P[i]->from].pop_back();
          P.resize(i);
        }
      }
    } else if (Lf[curr].empty()) { // Retreat
      P.pop_back();
      for (int i = 0; i < N; ++i)
        for (int j = 0; j < Lf[i].size(); ++j)
          if (Lf[i][j]->to == curr)
            Lf[i].erase(Lf[i].begin() + j);
    } else { // Advance
      P.push_back(Lf[curr].back());
    }
  }
  return totflow;
}
```

```
  LL GetMaxFlow(int s, int t) {
    LL totflow = 0;
    while (LL flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow
    (FASTFLOW)

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

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

// END CUT
```

# 5 Euclid

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

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
```

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

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

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

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

// (a^b) mod m via successive squaring
int powermod(int a, int b, int m)
{
	int ret = 1;
	while (b)
	{
		if (b & 1) ret = mod(ret*a, m);
		a = mod(a*a, m);
		b >>= 1;
	}
	return ret;
}

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

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

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

// Chinese remainder theorem (special case): find z such that
// z % m1 = r1, z % m2 = r2. Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M). On failure, M = -1.
PII chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
	int s, t;
	int g = extended_euclid(m1, m2, s, t);
	if (r1%g != r2%g) return make_pair(0, -1);
	return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
}

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

```
// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
	if (!a && !b)
	{
		if (c) return false;
		x = 0; y = 0;
		return true;
	}
	if (!a)
	{
		if (c % b) return false;
		x = 0; y = c / b;
		return true;
	}
	if (!b)
	{
		if (c % a) return false;
		x = c / a; y = 0;
		return true;
	}
	int g = gcd(a, b);
	if (c % g) return false;
	x = c / g * mod_inverse(a / g, b / g);
	y = (c - a*x) / b;
	return true;
}

int main() {
	// expected: 2
	cout << gcd(14, 30) << endl;

	// expected: 2 -2 1
	int x, y;
	int g = extended_euclid(14, 30, x, y);
	cout << g << " " << x << " " << y << endl;

	// expected: 95 451
	VI sols = modular_linear_equation_solver(14, 30, 100);
	for (int i = 0; i < sols.size(); i++) cout << sols[i] << " ";
	cout << endl;

	// expected: 8
	cout << mod_inverse(8, 9) << endl;

	// expected: 23 105
	//           11 12
	PII ret = chinese_remainder_theorem(VI({ 3, 5, 7 }), VI({ 2, 3, 2
	    }));
	cout << ret.first << " " << ret.second << endl;
	ret = chinese_remainder_theorem(VI({ 4, 6 }), VI({ 3, 5 }));
	cout << ret.first << " " << ret.second << endl;

	// expected: 5 -15
	if (!linear_diophantine(7, 2, 5, x, y)) cout << "ERROR" << endl;
	cout << x << " " << y << endl;
	return 0;
}
```

# 6 FFT

```
// Convolution using the fast Fourier transform (FFT).
//
// INPUT:
//     a[1...n]
//     b[1...m]
//
// OUTPUT:
//     c[1...n+m-1] such that c[k] = sum_{i=0}^k a[i] b[k-i]
//
// Alternatively, you can use the DFT() routine directly, which will
// zero-pad your input to the next largest power of 2 and compute the
// DFT or inverse DFT.

#include <iostream>
#include <vector>
#include <complex>

using namespace std;

typedef long double DOUBLE;
typedef complex<DOUBLE> COMPLEX;
typedef vector<DOUBLE> VD;
typedef vector<COMPLEX> VC;

struct FFT {
```

```
VC A;
int n, L;

int ReverseBits(int k) {
  int ret = 0;
  for (int i = 0; i < L; i++) {
    ret = (ret << 1) | (k & 1);
    k >>= 1;
  }
  return ret;
}

void BitReverseCopy(VC a) {
  for (n = 1, L = 0; n < a.size(); n <<= 1, L++) ;
  A.resize(n);
  for (int k = 0; k < n; k++)
    A[ReverseBits(k)] = a[k];
}

VC DFT(VC a, bool inverse) {
  BitReverseCopy(a);
  for (int s = 1; s <= L; s++) {
    int m = 1 << s;
    COMPLEX wm = exp(COMPLEX(0, 2.0 * M_PI / m));
    if (inverse) wm = COMPLEX(1, 0) / wm;
    for (int k = 0; k < n; k += m) {
      COMPLEX w = 1;
      for (int j = 0; j < m/2; j++) {
        COMPLEX t = w * A[k + j + m/2];
        COMPLEX u = A[k + j];
        A[k + j] = u + t;
        A[k + j + m/2] = u - t;
        w = w * wm;
      }
    }
  }
  if (inverse) for (int i = 0; i < n; i++) A[i] /= n;
  return A;
}

// c[k] = sum_{i=0}^k a[i] b[k-i]
VD Convolution(VD a, VD b) {
  int L = 1;
  while ((1 << L) < a.size()) L++;
  while ((1 << L) < b.size()) L++;
    int n = 1 << (L+1);

    VC aa, bb;
    for (size_t i = 0; i < n; i++) aa.push_back(i < a.size() ?
        COMPLEX(a[i], 0) : 0);
    for (size_t i = 0; i < n; i++) bb.push_back(i < b.size() ?
        COMPLEX(b[i], 0) : 0);

    VC AA = DFT(aa, false);
    VC BB = DFT(bb, false);
    VC CC;
    for (size_t i = 0; i < AA.size(); i++) CC.push_back(AA[i] * BB[i]);
    VC cc = DFT(CC, true);

    VD c;
    for (int i = 0; i < a.size() + b.size() - 1; i++)
        c.push_back(cc[i].real());
    return c;
  }

};

int main() {
  double a[] = {1, 3, 4, 5, 7};
  double b[] = {2, 4, 6};

  FFT fft;
  VD c = fft.Convolution(VD(a, a + 5), VD(b, b + 3));

  // expected output: 2 10 26 44 58 58 42
  for (int i = 0; i < c.size(); i++) cerr << c[i] << " ";
  cerr << endl;

  return 0;
}
```

## 7 FastExpo

```
/*
Uses powers of two to exponentiate numbers and matrices. Calculates
n^k in O(log(k)) time when n is a number. If A is an n x n matrix,
calculates A^k in O(n^3*log(k)) time.
```

```
*/

#include <iostream>
#include <vector>

using namespace std;

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

T power(T x, int k) {
  T ret = 1;

  while(k) {
    if(k & 1) ret *= x;
    k >>= 1; x *= x;
  }
  return ret;
}

VVT multiply(VVT& A, VVT& B) {
  int n = A.size(), m = A[0].size(), k = B[0].size();
  VVT C(n, VT(k, 0));

  for(int i = 0; i < n; i++)
    for(int j = 0; j < k; j++)
      for(int l = 0; l < m; l++)
        C[i][j] += A[i][l] * B[l][j];

  return C;
}

VVT power(VVT& A, int k) {
  int n = A.size();
  VVT ret(n, VT(n)), B = A;
  for(int i = 0; i < n; i++) ret[i][i]=1;

  while(k) {
    if(k & 1) ret = multiply(ret, B);
    k >>= 1; B = multiply(B, B);
  }
  return ret;
}

int main()
{
  /* Expected Output:
     2.37^48 = 9.72569e+17

     376 264 285 220 265
     550 376 529 285 484
     484 265 376 264 285
     285 220 265 156 264
     529 285 484 265 376 */
  double n = 2.37;
  int k = 48;

  cout << n << "^" << k << " = " << power(n, k) << endl;

  double At[5][5] = {
    { 0, 0, 1, 0, 0 },
    { 1, 0, 0, 1, 0 },
    { 0, 0, 0, 0, 1 },
    { 1, 0, 0, 0, 0 },
    { 0, 1, 0, 0, 0 } };

  vector <vector <double> > A(5, vector <double>(5));
  for(int i = 0; i < 5; i++)
    for(int j = 0; j < 5; j++)
      A[i][j] = At[i][j];

  vector <vector <double> > Ap = power(A, k);

  cout << endl;
  for(int i = 0; i < 5; i++) {
    for(int j = 0; j < 5; j++)
      cout << Ap[i][j] << " ";
    cout << endl;
  }
}
```

## 8 FenwickTree

```
class FenwickTree {
private: vi ft;
        // recall that vi is: typedef vector<int> vi;
```

```
public:
        FenwickTree(long n) { ft.assign(n + 1, 0); }
        // init n + 1 zeroes
        long LSOne(long n){return (n & (-n));}
        long long rsq(long b)
        {
                // returns RSQ(1, b)
                long long sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
                return sum;
        }
        long long rsq(long a, long b)
        {
                return rsq(b) - (a == 1 ? 0 : rsq(a - 1));
        }
        // adjusts value of the k-th element by v (v can be +ve/inc or
            -ve/dec)
        void adjust(long k, long v)
        {
                // note: n = ft.size() - 1
                for (; k < (long)ft.size(); k += LSOne(k)) ft[k] += v;
        }
};
```

# 9 FenwickTree2D

```
long tree[max_xy][max_xy];

void update(long x , long y , long val){
    long y1;
    while (x <= max_xy){
        y1 = y;
        while (y1 <= max_xy){
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}

long long read(long x,long y){ // return sum from 1,1 to x,y.
   long long sum= 0;
     while(x){
          long y1 = y;
         while(y1){
             sum += tree[x][y1];
               y1 -= y1 & -y1;
         }
         x -= x & -x;
    }
        return sum;
}

int main()
{
        long long suma = read(x2+1,y2+1) + read(x1,y1) - read(x2+1,y1) -
            read(x1,y2+1); //suma de rango entre x1,y1 y x2,y2
        update(x+1,y+1, num ); //aumenta en num
}
```

# 10 FloydWar

```
//ASSP!!!
// This function runs the Floyd-Warshall algorithm for all-pairs
// shortest paths. Also handles negative edge weights. Returns true
// if a negative weight cycle is found.
//
// Running time: O(|V|^3)
//
//   INPUT:  w[i][j] = weight of edge from i to j
//   OUTPUT: w[i][j] = shortest path from i to j
//           prev[i][j] = node before j on the best path starting at i

bool FloydWarshall (VVT &w, VVI &prev){
  int n = w.size();
  prev = VVI (n, VI(n, -1));

  for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if (w[i][j] > w[i][k] + w[k][j]){
          w[i][j] = w[i][k] + w[k][j];
          prev[i][j] = k;
        }
      }
```

```
    }
  }

  // check for negative weight cycles
  for(int i=0;i<n;i++)
    if (w[i][i] < 0) return false;
  return true;
}
```

# 11 Geom3D

```
public class Geom3D {
  // distance from point (x, y, z) to plane aX + bY + cZ + d = 0
  public static double ptPlaneDist(double x, double y, double z,
      double a, double b, double c, double d) {
    return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a + b*b + c*c);
  }

  // distance between parallel planes aX + bY + cZ + d1 = 0 and
  // aX + bY + cZ + d2 = 0
  public static double planePlaneDist(double a, double b, double c,
      double d1, double d2) {
    return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b + c*c);
  }

  // distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
  // (or ray, or segment; in the case of the ray, the endpoint is the
  // first point)
  public static final int LINE = 0;
  public static final int SEGMENT = 1;
  public static final int RAY = 2;
  public static double ptLineDistSq(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);

    double x, y, z;
    if (pd2 == 0) {
      x = x1;
      y = y1;
      z = z1;
    } else {
      double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) /
          pd2;
      x = x1 + u * (x2 - x1);
      y = y1 + u * (y2 - y1);
      z = z1 + u * (z2 - z1);
      if (type != LINE && u < 0) {
        x = x1;
        y = y1;
        z = z1;
      }
      if (type == SEGMENT && u > 1.0) {
        x = x2;
        y = y2;
        z = z2;
      }
    }

    return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);
  }

  public static double ptLineDist(double x1, double y1, double z1,
      double x2, double y2, double z2, double px, double py, double pz,
      int type) {
    return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz,
        type));
  }
}
```

# 12 Geometry

```
// C++ routines for computational geometry.

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

using namespace std;

double INF = 1e100;
double EPS = 1e-12;
```

```
struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y) {}
  PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)  const { return PT(x*c, y*c ); }
  PT operator / (double c)  const { return PT(x/c, y/c ); }
};

double dot(PT p, PT q)  { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p)  { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
```

```cpp
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c,
      c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y -
          p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion. Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
```

```
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

int main() {

  // expected: (-5,2)
  cerr << RotateCCW90(PT(2,5)) << endl;

  // expected: (5,-2)
  cerr << RotateCW90(PT(2,5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 0 0 1
  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 1 1 1 0
  cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

  // expected: (1,2)
  cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) <<
       endl;

  // expected: (1,1)
  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

  vector<PT> v;
  v.push_back(PT(0,0));
  v.push_back(PT(5,0));
  v.push_back(PT(5,5));
  v.push_back(PT(0,5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2,2)) << " "
       << PointInPolygon(v, PT(2,0)) << " "
       << PointInPolygon(v, PT(0,2)) << " "
       << PointInPolygon(v, PT(5,2)) << " "
       << PointInPolygon(v, PT(2,5)) << endl;

  // expected: 0 1 1 1 1
  cerr << PointOnPolygon(v, PT(2,2)) << " "
       << PointOnPolygon(v, PT(2,0)) << " "
       << PointOnPolygon(v, PT(0,2)) << " "
       << PointOnPolygon(v, PT(5,2)) << " "
       << PointOnPolygon(v, PT(2,5)) << endl;

  // expected: (1,6)
  //           (5,4) (4,5)
  //           blank line
  //           (4,5) (5,4)
  //           blank line
  //           (4,5) (5,4)
  vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
```

```
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

  // area should be 5.0
  // centroid should be (1.1666666, 1.166666)
  PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
  vector<PT> p(pa, pa+4);
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

## 13 HLD

```
/*The code is probably too simple to explain it, but not well tested yet.
Graph can be represented for example as vector<vector<int>>.
I use single segment tree for all the chains. Creating separate
trees makes code harder to read and also slower (at least in my
experiment). For a good segment tree implementation you can look
here: http://codeforces.com/blog/entry/18051.
*/

#define VALUES_IN_VERTICES false

template <class T, int V>
class HeavyLight {
  int parent[V], heavy[V], depth[V];
  int root[V], treePos[V];
  SegmentTree<T> tree; // <---- importante implementar!

  template <class G>
  int dfs(const G& graph, int v) {
    int size = 1, maxSubtree = 0;
    for (int u : graph[v]) if (u != parent[v]) {
      parent[u] = v;
      depth[u] = depth[v] + 1;
      int subtree = dfs(graph, u);
      if (subtree > maxSubtree) heavy[v] = u, maxSubtree = subtree;
      size += subtree;
    }
    return size;
  }

  template <class BinaryOperation>
  void processPath(int u, int v, BinaryOperation op) {
    for (; root[u] != root[v]; v = parent[root[v]]) {
      if (depth[root[u]] > depth[root[v]]) swap(u, v);
      op(treePos[root[v]], treePos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(treePos[u] + (VALUES_IN_VERTICES ? 0 : 1), treePos[v] + 1);
  }

public:
  template <class G>
  void init(const G& graph) {
    int n = graph.size();
    fill_n(heavy, n, -1);
    parent[0] = -1;
    depth[0] = 0;
    dfs(graph, 0);
    for (int i = 0, currentPos = 0; i < n; ++i)
      if (parent[i] == -1 || heavy[parent[i]] != i)
        for (int j = i; j != -1; j = heavy[j]) {
          root[j] = i;
          treePos[j] = currentPos++;
        }
    tree.init(n);
  }

  void set(int v, const T& value) {
    tree.set(treePos[v], value);
  }

  void modifyPath(int u, int v, const T& value) {
```

```
    processPath(u, v, [this, &value](int l, int r) { tree.modify(l, r,
        value); });
  }

  T queryPath(int u, int v) {
    T res = T();
    processPath(u, v, [this, &res](int l, int r) { res.add(tree.query(l,
        r)); });
    return res;
  }
};
```

# 14 Kadane1DMaxSum

```
#include <bits/stdc++.h>

using namespace std;

int max1DSum(int* arr,int size)
{
	int max_ending_here,max_so_far;
	max_ending_here = max_so_far = arr[0];
	for (int i = 1; i < size; ++i)
	{
		max_ending_here = max(arr[i],max_ending_here + arr[i]);
		max_so_far = max(max_so_far, max_ending_here);
	}
	return max_so_far;
}

int main()
{
	int arr[9] = {5,3, -16,7,-8,9,10,-14,33};
	//print: max 1D sum = 38
	cout << "max 1D sum = " << max1DSum(arr,9) << endl;
	return 0;
}
```

# 15 Kruskal

```
/*
Uses Kruskal's Algorithm to calculate the weight of the minimum spanning
forest (union of minimum spanning trees of each connected component) of
a possibly disjoint graph, given in the form of a matrix of edge weights
(-1 if no edge exists). Returns the weight of the minimum spanning
forest (also calculates the actual edges - stored in T). Note: uses a
disjoint-set data structure with amortized (effectively) constant time per
union/find. Runs in O(E*log(E)) time.
*/

#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

typedef int T;

struct edge
{
  int u, v;
  T d;
};

struct edgeCmp
{
  int operator()(const edge& a, const edge& b) { return a.d > b.d; }
};

int find(vector <int>& C, int x) { return (C[x] == x) ? x : C[x] =
    find(C, C[x]); }

T Kruskal(vector <vector <T> >& w)
{
  int n = w.size();
  T weight = 0;

  vector <int> C(n), R(n);
  for(int i=0; i<n; i++) { C[i] = i; R[i] = 0; }

  vector <edge> T;
  priority_queue <edge, vector <edge>, edgeCmp> E;
```

```
  for(int i=0; i<n; i++)
    for(int j=i+1; j<n; j++)
      if(w[i][j] >= 0)
      {
        edge e;
        e.u = i; e.v = j; e.d = w[i][j];
        E.push(e);
      }

  while(T.size() < n-1 && !E.empty())
  {
    edge cur = E.top(); E.pop();

    int uc = find(C, cur.u), vc = find(C, cur.v);
    if(uc != vc)
    {
      T.push_back(cur); weight += cur.d;

      if(R[uc] > R[vc]) C[vc] = uc;
      else if(R[vc] > R[uc]) C[uc] = vc;
      else { C[vc] = uc; R[uc]++; }
    }
  }

  return weight;
}

int main()
{
  int wa[6][6] = {
    { 0, -1, 2, -1, 7, -1 },
    { -1, 0, -1, 2, -1, -1 },
    { 2, -1, 0, -1, 8, 6 },
    { -1, 2, -1, 0, -1, -1 },
    { 7, -1, 8, -1, 0, 4 },
    { -1, -1, 6, -1, 4, 0 } };

  vector <vector <int> > w(6, vector <int>(6));

  for(int i=0; i<6; i++)
    for(int j=0; j<6; j++)
      w[i][j] = wa[i][j];

  cout << Kruskal(w) << endl;
  cin >> wa[0][0];
}
```

# 16 LCA

```
const int max_nodes, log_max_nodes;
int num_nodes, log_num_nodes, root;

vector<int> children[max_nodes];    // children[i] contains the children
    of node i
int A[max_nodes][log_max_nodes+1];  // A[i][j] is the 2^j-th ancestor of
    node i, or -1 if that ancestor does not exist
int L[max_nodes];                   // L[i] is the distance between node
    i and the root

// floor of the binary logarithm of n
int lb(unsigned int n)
{
    if(n==0)
        return -1;
    int p = 0;
    if (n >= 1<<16) { n >>= 16; p += 16; }
    if (n >= 1<< 8) { n >>= 8; p += 8; }
    if (n >= 1<< 4) { n >>= 4; p += 4; }
    if (n >= 1<< 2) { n >>= 2; p += 2; }
    if (n >= 1<< 1) {       p += 1; }
    return p;
}

void DFS(int i, int l)
{
    L[i] = l;
    for(int j = 0; j < children[i].size(); j++)
        DFS(children[i][j], l+1);
}

int LCA(int p, int q)
{
    // ensure node p is at least as deep as node q
    if(L[p] < L[q])
        swap(p, q);
```

```
    // "binary search" for the ancestor of node p situated on the same
        level as q
    for(int i = log_num_nodes; i >= 0; i--)
        if(L[p] - (1<<i) >= L[q])
            p = A[p][i];

    if(p == q)
        return p;

    // "binary search" for the LCA
    for(int i = log_num_nodes; i >= 0; i--)
        if(A[p][i] != -1 && A[p][i] != A[q][i])
        {
            p = A[p][i];
            q = A[q][i];
        }

    return A[p][0];
}

int main(int argc,char* argv[])
{
    // read num_nodes, the total number of nodes
    log_num_nodes=lb(num_nodes);

    for(int i = 0; i < num_nodes; i++)
    {
        int p;
        // read p, the parent of node i or -1 if node i is the root

        A[i][0] = p;
        if(p != -1)
            children[p].push_back(i);
        else
            root = i;
    }

    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
        for(int i = 0; i < num_nodes; i++)
            if(A[i][j-1] != -1)
                A[i][j] = A[A[i][j-1]][j-1];
            else
                A[i][j] = -1;

    // precompute L
    DFS(root, 0);


    return 0;
}
```

# 17 LCS

```
/*
Calculates the length of the longest common subsequence of two vectors.
Backtracks to find a single subsequence or all subsequences. Runs in
O(m*n) time except for finding all longest common subsequences, which
may be slow depending on how many there are.
*/

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;

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

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

void backtrack(VVI& dp, VT& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) return;
  if(A[i-1] == B[j-1]) { res.push_back(A[i-1]); backtrack(dp, res, A, B,
      i-1, j-1); }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrack(dp, res, A, B, i, j-1);
    else backtrack(dp, res, A, B, i-1, j);
  }
}
```

```
void backtrackall(VVI& dp, set<VT>& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) { res.insert(VI()); return; }
  if(A[i-1] == B[j-1])
  {
    set<VT> tempres;
    backtrackall(dp, tempres, A, B, i-1, j-1);
    for(set<VT>::iterator it=tempres.begin(); it!=tempres.end(); it++)
    {
      VT temp = *it;
      temp.push_back(A[i-1]);
      res.insert(temp);
    }
  }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrackall(dp, res, A, B, i, j-1);
    if(dp[i][j-1] <= dp[i-1][j]) backtrackall(dp, res, A, B, i-1, j);
  }
}

VT LCS(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);

  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }

  VT res;
  backtrack(dp, res, A, B, n, m);
  reverse(res.begin(), res.end());
  return res;
}

set<VT> LCSall(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);
  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
  set<VT> res;
  backtrackall(dp, res, A, B, n, m);
  return res;
}

int main()
{
  int a[] = { 0, 5, 5, 2, 1, 4, 2, 3 }, b[] = { 5, 2, 4, 3, 2, 1, 2, 1, 3
     };
  VI A = VI(a, a+8), B = VI(b, b+9);
  VI C = LCS(A, B);

  for(int i=0; i<C.size(); i++) cout << C[i] << " ";
  cout << endl << endl;

  set <VI> D = LCSall(A, B);
  for(set<VI>::iterator it = D.begin(); it != D.end(); it++)
  {
    for(int i=0; i<(*it).size(); i++) cout << (*it)[i] << " ";
    cout << endl;
  }
}
```

# 18 LongestIncreasingSubsequence

```
// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest increasing subsequence

using namespace std;
```

```
typedef vector<int> vi;

vi LIS(vi v)
{
        vi p(v.size(),-1); //parents
        vi m(v.size()+1,0); //increasing ordered list
        int l = 0;

        for (int i = 0; i < v.size(); ++i)
        {
                int lo=1,hi=l;

                while(lo <= hi)
                {
                        int mid = ceil((lo+hi)/2);
                        if( v[m[mid]] < v[i]) lo = mid+1;
                        else hi = mid-1;
                }
                int newL = lo;
                p[i] = m[newL-1];
                m[newL] = i;

                l = max(newL,l);
        }
        vi s(l,0);
        int k = m[l];
        for (int i = l-1; i >=0; --i)
        {
                s[i] = v[k];
                k = p[k];
        }

    return s; //return longest increasing subsequence
}
```

# 19 MaxFlow

```
vector<vector<long> > res; //adj
vector<vi> edgeflows; //edge flow
long mf, f, s, t; // global variables
vi p; // p stores the BFS spanning tree from s

void augment(long v, long minEdge)
{ // traverse BFS spanning tree from s->t
        if (v == s) { f = minEdge; return; } // record minEdge in a global
            var f
        else if (p[v] != -1)
        {
                augment(p[v], min(minEdge,edgeflows[p[v]][v]));
                edgeflows[p[v]][v] -= f; edgeflows[v][p[v]] += f;
        }
}

int main()// inside int main(): set up res ,   s   , and   t   with
    appropriate values
{

        while (1)
        {
                f = 0;

                vi dist(MAX_V, INF); dist[s] = 0; queue<long> q; q.push(s);
                p.assign(MAX_V, -1); // record the BFS spanning tree, from
                    s to t!
                while (!q.empty())
                {

                        long u = q.front(); q.pop();
                        if (u == t) break; // immediately stop BFS if we
                            already reach sink t
                        for (long v:res[u]) // note: this part is slow
                                if (edgeflows[u][v] > 0 && dist[v] == INF)
                                        dist[v] = dist[u] + 1, q.push(v),
                                            p[v] = u; // 3 lines in 1!
                }

                augment(t, INF); // find the min edge weight  f  in this
                    path, if any
                if (f == 0) break; // we cannot send any more flow (f =
                    0), terminate
                mf += f; // we can still send a flow, increase the max
                    flow!
        }
        printf("%ld\n", mf);
```

```
}
```

# 20 Prim

```
// This function runs Prim's algorithm for constructing minimum
// weight spanning trees.
//
// Running time: O(|V|^2)
//
//   INPUT:  w[i][j] = cost of edge from i to j
//
//           NOTE: Make sure that w[i][j] is nonnegative and
//           symmetric. Missing edges should be given -1
//           weight.
//
//   OUTPUT: edges = list of pair<int,int> in minimum spanning tree
//           return total weight of tree

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

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

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

T Prim (const VVT &w, VPII &edges){
  int n = w.size();
  VI found (n);
  VI prev (n, -1);
  VT dist (n, 1000000000);
  int here = 0;
  dist[here] = 0;

  while (here != -1){
    found[here] = true;
    int best = -1;
    for (int k = 0; k < n; k++) if (!found[k]){
      if (w[here][k] != -1 && dist[k] > w[here][k]){
        dist[k] = w[here][k];
        prev[k] = here;
      }
      if (best == -1 || dist[k] < dist[best]) best = k;
    }
    here = best;
  }

  T tot_weight = 0;
  for (int i = 0; i < n; i++) if (prev[i] != -1){
    edges.push_back (make_pair (prev[i], i));
    tot_weight += w[prev[i]][i];
  }
  return tot_weight;
}

int main(){
  int ww[5][5] = {
    {0, 400, 400, 300, 600},
    {400, 0, 3, -1, 7},
    {400, 3, 0, 2, 0},
    {300, -1, 2, 0, 5},
    {600, 7, 0, 5, 0}
  };
  VVT w(5, VT(5));
  for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
      w[i][j] = ww[i][j];

  // expected: 305
  //           2 1
  //           3 2
  //           0 3
  //           2 4

  VPII edges;
  cout << Prim (w, edges) << endl;
  for (int i = 0; i < edges.size(); i++)
    cout << edges[i].first << " " << edges[i].second << endl;
}
```

## 21 Primes

```
// O(sqrt(x)) Exhaustive Primality Test
#include <cmath>
#define EPS 1e-7
typedef long long LL;
bool IsPrimeSlow (LL x)
{
  if(x<=1) return false;
  if(x<=3) return true;
  if (!(x%2) || !(x%3)) return false;
  LL s=(LL)(sqrt((double)(x))+EPS);
  for(LL i=5;i<=s;i+=6)
  {
    if (!(x%i) || !(x%(i+2))) return false;
  }
  return true;
}
// Primes less than 1000:
//      2     3     5     7    11    13    17    19    23    29    31    37
//     41    43    47    53    59    61    67    71    73    79    83    89
//     97   101   103   107   109   113   127   131   137   139   149   151
//    157   163   167   173   179   181   191   193   197   199   211   223
//    227   229   233   239   241   251   257   263   269   271   277   281
//    283   293   307   311   313   317   331   337   347   349   353   359
//    367   373   379   383   389   397   401   409   419   421   431   433
//    439   443   449   457   461   463   467   479   487   491   499   503
//    509   521   523   541   547   557   563   569   571   577   587   593
//    599   601   607   613   617   619   631   641   643   647   653   659
//    661   673   677   683   691   701   709   719   727   733   739   743
//    751   757   761   769   773   787   797   809   811   821   823   827
//    829   839   853   857   859   863   877   881   883   887   907   911
//    919   929   937   941   947   953   967   971   977   983   991   997

// Other primes:
//    The largest prime smaller than 10 is 7.
//    The largest prime smaller than 100 is 97.
//    The largest prime smaller than 1000 is 997.
//    The largest prime smaller than 10000 is 9973.
//    The largest prime smaller than 100000 is 99991.
//    The largest prime smaller than 1000000 is 999983.
//    The largest prime smaller than 10000000 is 9999991.
//    The largest prime smaller than 100000000 is 99999989.
//    The largest prime smaller than 1000000000 is 999999937.
//    The largest prime smaller than 10000000000 is 9999999967.
//    The largest prime smaller than 100000000000 is 99999999977.
//    The largest prime smaller than 1000000000000 is 999999999989.
//    The largest prime smaller than 10000000000000 is 9999999999971.
//    The largest prime smaller than 100000000000000 is 99999999999973.
//    The largest prime smaller than 1000000000000000 is 999999999999989.
//    The largest prime smaller than 10000000000000000 is
    9999999999999937.
//    The largest prime smaller than 100000000000000000 is
    99999999999999997.
//    The largest prime smaller than 1000000000000000000 is
    999999999999999989.
```

## 22 RollingHash

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

struct RollingHash
{
        string s;
        const long L;
        long long hashValue;
        long ini;
        long end;
        long long base;

        RollingHash(const string &s,long
            hlength):L(hlength),s(s),ini(0),end(L-1),hashValue(0)
        {
                base = 1;
                for (long i = L-1; i >= 0; --i)
                {
                        //hashValue += (s[i])<<((L-1) - i);
                        hashValue += s[i] * base;
                        if(i!=0)base*=127;
                }
        }

        long long getHashValue(){return hashValue;}
```

```
    //returns the next hash value. If the last value is reached, next
        calls will return the last value.
    long long nextHashValue()
    {
        if(end == s.length()) return hashValue;
        hashValue -= (s[ini]) * base;//<< (L-1);
        hashValue*=127;
        hashValue+= (s[end+1]);
        ini++;
        end++;
        return hashValue;
    }

    void reset()
    {
        hashValue = 0;
        for (long i = 0; i < L; ++i)
        {
            hashValue += (s[i])<<((L-1) - i);
        }
        end = end - ini;
        ini = 0;
    }

    bool finished()
    {
        return end==s.length();
    }
};

int main()
{
    string s1 = "aggg";
    string s2 = "aggtmreaggaggtmreagg";

    RollingHash RH(s1,s1.length());
    long long h1 = RH.getHashValue();

    RollingHash RH2(s2,s1.length());

    long long count = RH2.getHashValue()==h1?1:0;
    while(!RH2.finished())
    {
        if(RH2.nextHashValue()==h1)count++;
    }
    //ocurrences of "agg" in "aggtmreaggaggtmreagg" are 4
    cout << "ocurrences of \""<< s1 << "\" in \""<< s2<< "\" are "<<
        count << endl;

}
```

# 23 SCC

```
#define INF INT_MAX;

vector<vii> AdjList;
vi dfs_num, dfs_low, S, visited;
int dfsNumberCounter,numSCC,UNVISITED=-1,VISITED=1;

void tarjanSCC(int u)
{
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <=
        dfs_num[u]
    S.push_back(u);
    // stores u in a vector based on order of visitation
    visited[u] = 1;
    for (int j = 0; j < (int)AdjList[u].size(); j++)
    {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == UNVISITED)
        tarjanSCC(v.first);
        if (visited[v.first])
        // condition for update
        dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
    }
    if (dfs_low[u] == dfs_num[u])
    {
        bool agg = false;
        while (1)
        {
            int v = S.back(); S.pop_back(); visited[v] = 0;
            //aca se ouede imprimir la visita si es necesario
            if (u == v) break;
        }
    }
}
```

# 24 SegmentTree

```
#include <bits/stdc++.h>

using namespace std;

typedef vector<long long> vi;

#define MAX (1+(1<<6)) // Why? :D
#define inf 0x7fffffff //9223372036854775807 max llong

//st para max queries
class SegmentTree
{
private:
    int N;
    vi arr;
    long long tree[MAX];
    long long lazy[MAX];

    /**
     * Build and init tree
     */
    void build_tree(long long node, long long a, long long b) {
        if(a > b) return; // Out of range

        if(a == b) { // Leaf node
            tree[node] = arr[a]; // Init value
            return;
        }

        build_tree(node*2, a, (a+b)/2); // Init left child
        build_tree(node*2+1, 1+(a+b)/2, b); // Init right child

        tree[node] = max(tree[node*2], tree[node*2+1]); // Init root value
    }

    /**
     * Increment elements within range [i, j] with value value
     */
    void update_tree(long long node, long long a, long long b, long long i, long long j, long long value) {

        if(lazy[node] != 0) { // This node needs to be updated
            tree[node] += lazy[node]; // Update it

            if(a != b) {
                lazy[node*2] += lazy[node]; // Mark child as lazy
                lazy[node*2+1] += lazy[node]; // Mark child as lazy
            }

            lazy[node] = 0; // Reset it
        }

        if(a > b || a > j || b < i) // Current segment is not within range [i, j]
            return;

        if(a >= i && b <= j) { // Segment is fully within range
            tree[node] += value;

            if(a != b) { // Not leaf node
                lazy[node*2] += value;
                lazy[node*2+1] += value;
            }

            return;
        }

        update_tree(node*2, a, (a+b)/2, i, j, value); // Updating left child
        update_tree(1+node*2, 1+(a+b)/2, b, i, j, value); // Updating right child

        tree[node] = max(tree[node*2], tree[node*2+1]); // Updating root with max value
    }

    /**
     * Query tree to get max element value within range [i, j]
     */
    long long query_tree(long long node, long long a, long long b, long long i, long long j) {

        if(a > b || a > j || b < i) return -inf; // Out of range
```

```
        if(lazy[node] != 0) { // This node needs to be updated
            tree[node] += lazy[node]; // Update it

            if(a != b) {
                lazy[node*2] += lazy[node]; // Mark child as
                    lazy
                lazy[node*2+1] += lazy[node]; // Mark child
                    as lazy
            }

            lazy[node] = 0; // Reset it
        }

        if(a >= i && b <= j) // Current segment is totally within
            range [i, j]
            return tree[node];

        long long q1 = query_tree(node*2, a, (a+b)/2, i, j); //
            Query left child
        long long q2 = query_tree(1+node*2, 1+(a+b)/2, b, i, j);
            // Query right child

        long long res = max(q1, q2); // Return final result

        return res;
    }
public:

    SegmentTree(long long N):N(N)
    {
        arr.assign(N,0);
        build_tree(1, 0, N-1);
        memset(lazy, 0, sizeof lazy);
    }

    void update(long long i,long long j, long long v)
    {
        update_tree(1, 0, N-1, i, j, v);
    }

    long long rmq(long long i,long long j)
    {
        query_tree(1, 0, N-1, i, j);
    }
};
```

```
//st para sum queries
class SegmentTreeSum
{
private:
    int N;
    vi arr;
    long long tree[MAX];
    long long lazy[MAX];

    /**
     * Build and init tree
     */
    void build_tree(long long node, long long a, long long b) {
        if(a > b) return; // Out of range

        if(a == b) { // Leaf node
            tree[node] = arr[a]; // Init value
            return;
        }

        build_tree(node*2, a, (a+b)/2); // Init left child
        build_tree(node*2+1, 1+(a+b)/2, b); // Init right child

        tree[node] = tree[node*2] + tree[node*2+1]; // Init root
            value
    }

    /**
     * Increment elements within range [i, j] with value value
     */
    void update_tree(long long node, long long a, long long b, long
        long i, long long j, long long value) {

        if(lazy[node] != 0) { // This node needs to be updated
            tree[node] += (b-a+1)*lazy[node]; // Update it

            if(a != b) {
                lazy[node*2] += lazy[node]; // Mark child as
                    lazy
                lazy[node*2+1] += lazy[node]; // Mark child
                    as lazy
            }

            lazy[node] = 0; // Reset it
```

```
        }

        if(a > b || a > j || b < i) // Current segment is not
            within range [i, j]
            return;

        if(a >= i && b <= j) { // Segment is fully within range
            tree[node] += (b-a+1)*value;

            if(a != b) { // Not leaf node
                lazy[node*2] += value;
                lazy[node*2+1] += value;
            }

            return;
        }

        update_tree(node*2, a, (a+b)/2, i, j, value); // Updating
            left child
        update_tree(1+node*2, 1+(a+b)/2, b, i, j, value); //
            Updating right child

        tree[node] = (tree[node*2] + tree[node*2+1]); // Updating
            root with max value
}

/**
 * Query tree to get max element value within range [i, j]
 */
long long query_tree(long long node, long long a, long long b,
    long long i, long long j) {

        if(a > b || a > j || b < i) return 0; // Out of range

        if(lazy[node] != 0) { // This node needs to be updated
            tree[node] += (b-a+1)*lazy[node]; // Update it

            if(a != b) {
                lazy[node*2] += lazy[node]; // Mark child as
                    lazy
                lazy[node*2+1] += lazy[node]; // Mark child
                    as lazy
            }

            lazy[node] = 0; // Reset it
        }

        if(a >= i && b <= j) // Current segment is totally within
            range [i, j]
            return tree[node];

        long long q1 = query_tree(node*2, a, (a+b)/2, i, j); //
            Query left child
        long long q2 = query_tree(1+node*2, 1+(a+b)/2, b, i, j);
            // Query right child

        long long res = (q1 + q2); // Return final result

        return res;
    }
public:

    SegmentTreeSum(long long N):N(N)
    {
        arr.assign(N,0);
        build_tree(1, 0, N-1);
        memset(lazy, 0, sizeof lazy);
    }

    void update(long long i,long long j, long long v)
    {
        update_tree(1, 0, N-1, i, j, v);
    }

    long long rsq(long long i,long long j)
    {
        return query_tree(1, 0, N-1, i, j);
    }
};

int main()
{
    int N = 20;
    SegmentTree st(N);
    st.update(0, 6, 5); // Increment range [0, 6] by 5. here 0, N-1
        represent the current range.
    st.update(7, 10, 12); // Incremenet range [7, 10] by 12. here 0,
        N-1 represent the current range.
    st.update(11, N-1, 100); // Increment range [10, N-1] by 100. here
        0, N-1 represent the current range.
```

```
    cout << st.rmq(0, N-1) << endl; // Get max element in range [0,
        N-1]
    return 0;
}
```

# 25 SuffixArray

```
// Suffix array construction in O(L log^2 L) time. Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:  string s
//
// OUTPUT: array suffix[] such that suffix[i] = index (from 0 to L-1)
//         of substring s[i...L-1] in the list of sorted suffixes.
//         That is, if we take the inverse of the permutation suffix[],
//         we get the actual suffix array.

#include <bits/stdc++.h>

using namespace std;

struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L,
      0)), M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector<int>(L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ?
            P[level-1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ?
            P[level][M[i-1].second] : i;
    }
  }

  vector<int> GetSuffixArray() { return P.back(); }

  // returns the length of the longest common prefix of s[i...L-1] and
      s[j...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};

int main() {

    // bobocel is the 0'th suffix
    //  obocel is the 5'th suffix
    //   bocel is the 1'st suffix
    //    ocel is the 6'th suffix
    //     cel is the 2'nd suffix
    //      el is the 3'rd suffix
    //       l is the 4'th suffix
        string s = "amandamandaamandamanda";
    SuffixArray suffix(s);
    vector<int> v = suffix.GetSuffixArray();

    // Expected output: 0 5 1 6 2 3 4
    //                    2
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
        cout << endl;
    vector<int> v2(v.size(),0);
    for (int i = 0; i < v.size(); i++)
    {
         v2[v[i]] = i;
    }
    for(int n:v2)
    {
         cout << s.substr(n) << endl;
```

```
    }
}
```

# 26 TopoSort

```
// This function uses performs a non-recursive topological sort.
//
// Running time: O(|V|^2). If you use adjacency lists (vector<map<int> >),
//               the running time is reduced to O(|E|).
//
//   INPUT:   w[i][j] = 1 if i should come before j, 0 otherwise
//   OUTPUT:  a permutation of 0,...,n-1 (stored in a vector)
//            which represents an ordering of the nodes which
//            is consistent with w
//
// If no ordering is possible, false is returned.
// Puede usarse para saber si el grafo es un DAG!

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

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

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

bool TopologicalSort (const VVI &w, VI &order){
  int n = w.size();
  VI parents (n);
  queue<int> q;
  order.clear();

  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++)
      if (w[j][i]) parents[i]++;
      if (parents[i] == 0) q.push (i);
  }

  while (q.size() > 0){
    int i = q.front();
    q.pop();
    order.push_back (i);
    for (int j = 0; j < n; j++) if (w[i][j]){
      parents[j]--;
      if (parents[j] == 0) q.push (j);
    }
  }

  return (order.size() == n);
}
```

# 27 UnionFind

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

//map implementation, requires c++11

class UnionFind
{
        private: unordered_map<int,int> p, rank; // remember: vi is
            vector<int>
        public:
                UnionFind(int N)
                {
                        //rank.assign(N, 0);
                        //p.assign(N, 0);
                        //for (int i = 0; i < N; i++) p[i] = i;
                }
                int findSet(int i) { if(p.count(i)==0)p[i]=i; return (p[i]
                    == i) ? i : (p[i] = findSet(p[i])); }
                bool isSameSet(int i, int j) { return findSet(i) ==
                    findSet(j); }
                void unionSet(int i, int j)
                {
                        if (!isSameSet(i, j))
                        { // if from different set
```

```
                int x = findSet(i), y = findSet(j);
                if (rank[x] > rank[y]) p[y] = x; // rank
                    keeps the tree short
                else
                {
                        p[x] = y;
                        if (rank[x] == rank[y]) rank[y] =
                            rank[y]+1;
                }
        }
    }
};
```