Grafi

Ordinamento topologico (DAG)

- per ogni nodo esploro tutti i figli e dopo inserisco il risultato in uno stack
- il risultato è lo stack ribaltato

SCC, Kosaraju, Strongly Connected Components (grafo diretto)

- calcolo l'ordinamento topologico
- calcolo il grafo trasposto (con i vertici al contrario)
- esploro il grafo trasposto procedendo nell'ordine dell'ordinamento topologico. Ogni esplorazione è una componente fortemente connessa (eventualmente di 1 nodo).

```
struct Node {
   bool vis=false;
   int condensed=-1;
   vector<int> to, from;
};
struct CNode {
   vector<int> nodes;
   vector<int> to, from;
};

vector<int> ordinati;
function<void(int)> toposort = [&](int i){
    if(nodes[i].vis) return;
    nodes[i].vis=true;
    for(auto e:nodes[i].to) toposort(e);
    ordinati.push_back(i);
};
for(int n=0;n<N;++n) toposort(n);

int c=0;
vector<CNode> condensed;
function<void(int)> condense = [&](int i){
    if(nodes[i].condensed!=-1) {
        if(nodes[i].condensed!=c) {
            condensed[c].from.push_back(nodes[i].condensed);
            condensed[nodes[i].condensed].to.push_back(c);
        }
        return;
    }
    nodes[i].condensed=c;
    condensed[c].nodes.push_back(i);
    for(auto e:nodes[i].from) condense(e);
```

```
};

for(int n=0;n<N;++n){
  if(nodes[ordinati[N-n-1]].condensed==-1) {
     condensed.emplace_back();
     condense(ordinati[N-n-1]);
     ++c;
  }
}
```

Max flow

- in BFS -> complessità `V*E^2`
- in DFS -> complessità `flow*(V+E)`
- Utile per risolvere bipartite matching -> attaccare sorgente (source) a tutti nodi a sinistra, pozzo (sink) a tutti i nodi a destra, tutti gli archi con flow = 1

```
vector<int> parent(N);
auto bfsAugmentingPath = [&]() -> int {
   fill(parent.begin(), parent.end(), -1);
   parent[S] = -2; // prevent passing through S
   queue<pair<int, int>> q;
   q.push({S, numeric_limits<int>::max()});

   while (!q.empty()) {
       auto [i, flow] = q.front();
       q.pop();

       for (int e : adj[i]) {
          if (capacity[i][e] > 0 && parent[e] == -1) {
             parent[e] = i;
             if (e == T) return min(flow, capacity[i][e]);
             q.push({e, min(flow, capacity[i][e])});
          }
       }
   }
   return 0;
};

int flow=0;
while(1) {
    int partialFlow = bfsAugmentingPath();
    if (partialFlow == 0) break;
    flow += partialFlow;
```

```
    int last=T;
    while(last!=S){
       capacity[parent[last]][last] -= partialFlow;
       capacity[last][parent[last]] += partialFlow;
       last = parent[last];
    }
 }
```

Tarjan, Articulation points and bridges (grafo non diretto)

- Inizialmente settare `t=0`
- Fare dfs incrementando `t` ogni volta che si attraversa un arco in avanti, cioè ogni volta che si vede un nuovo nodo
- Ogni nodo ha un `tEntrata` e `tMin`
  - `tEntrata` è il `t` della prima volta in cui quel nodo è stato visto
  - `tMin` è il min tra `tEntrata` e tutti i `tMin` dei nodi adiacenti eccetto il padre nella dfs
- Il nodo `b` è un **articulation point** se esiste un nodo adiacente `a` tale per cui `a.tMin >= b.tEntrata`
- L'arco che connette due nodi `a` e `b` è un **bridge** se `a.tMin > b.tEntrata`

Bipartite graph / bicoloring

```
struct Node {
  int color=-1;
  vector<int> conn;
};

int32_t main() {
  vector<Node> nodes(N);
  queue<pair<int,int>> q;
  q.push({0,0});
  bool bicolorable=true;

  while (!q.empty()) {
    auto [i,c] = q.front();
    q.pop();

    if (nodes[i].color == -1) {
      nodes[i].color=c;
      for(auto&& con : nodes[i].conn) {
        q.push({con, (c+1)%2});
      }
    } else {
      if (nodes[i].color!=c) {
        bicolorable=false;
```

```
        break;
      }
    }
  }
}
```

Alberi

UFDS, Union Find Disjoint Set

- si può ottimizzare ammortizzando di più mettendo `return nodes[i].parent = parent(nodes[i].parent);` in `parent()`

```
struct Node {
    int parent=-1, rank=0;
};
function<int(int)> parent = [&](int i) {
    if (nodes[i].parent == -1) return i;
    return parent(nodes[i].parent);
};
auto connect = [&](int a, int b) {
    int pa = parent(a);
    int pb = parent(b);
    if (pa==pb) return false;
    if (nodes[pa].depth > nodes[pb].depth) swap(pa, pb);
    nodes[pa].parent = pb;
    nodes[pb].depth = max(nodes[pb].depth, 1+nodes[pa].depth);
    return true;
};
```

LCA, Lowest Common Ancestor

- Salvarsi per ogni nodo la profondità dalla radice
- Trovare con binary lifting per ogni nodo l'array `antenati[20]` (e se serve anche anche `dist[20]` o `minarco[20]`) dove `antenati[e]` indica l'antenato risalendo di `2^e` nodi. Basta prima impostare gli `antenati[0]` e poi fare un `for(0<e<20) for(nodo in albero) nodo.antenati[e] = albero[nodo.antenati[e-1]].antenati[e-1]`:

```
for(int e=1;e<20;++e) {
   for(int i=0; i<N; i++) {
      int half = albero[i].antenati[e-1];
      albero[i].antenati[e] = albero[half].antenati[e-1];
      albero[i].dist[e] = albero[i].dist[e-1] + albero[half].dist[e-1];
      albero[i].minarco[e] = min(albero[i].minarco[e-1], albero[half].minarco[e-1]);
   }
}
```

- Funzione `lift()`:

```
int lift(int v, int h) {
   for(int e=20; e>=0; e--) {
      if(h & (1<<e)) {
         v = albero[v].antenati[e];
      }
   }
   return v;
}
```

- Funzione `lca()` (si cicla `e` al contrario per scomporre in potenze di 2 decrescenti):

```
int lca(int u, int v) {
   int hu=albero[u].altezza, hv=albero[v].altezza;
   if (hu>hv) {
      u=lift(u, hu-hv);
   } else if (hv>hu) {
      v=lift(v, hv-hu);
   }

   if(u==v) {
      return u;
   }

   for(int e=19; e>=0; e--) {
      if(albero[u].antenati[e]!=albero[v].antenati[e]) {
         u=albero[u].antenati[e];
         v=albero[v].antenati[e];
      }
   }
   return albero[u].antenati[0];
}
```

Oppure si può fare anche in `O(n)`: - dfs dalla radice salvando quando nodi vengono aperti e chiusi in array - fare Range Minimum Query con una Sparse Table (la costruzione richiede `O(n*logn)`)

MST, Minimum spanning tree

- Sortare gli archi per il peso (`O(E*logE)=O(V^2*logV)`)
- Partire dagli archi più piccoli ed aggiungerli all'albero, ma solo se questo non lo rende non più un albero (`O(E)=O(V^2)`)
- Usare Union Find per capire se un arco unirebbe due nodi già collegati e romperebbe l'albero
- Se serve trovare il *maximum* spanning tree basta scegliere gli archi più grossi invece che più piccoli

- Se serve trovare il *second* minimum spanning tree, si può:
  - Per ogni percorso che connette due nodi nel MST, trovare l'arco massimo nel percorso con `V` dfs (`O(V^2)`)
  - Per ogni arco `a-b` che non è già nel MST, calcolare di quanto aumenterebbe il peso totale del MST se si aggiungesse quell'arco e si togliesse però l'arco massimo nel percorso `a-b`.
  - Il minimo dei pesi totali trovati sopra corrisponde al second minimum spanning tree

Strutture dati

Segment tree base

```
int higherPowerOf2(int x) {
    int res = 1;
    while (res < x) res *= 2;
    return res;
}

struct SegmentTree {
    vector<int> data;
    SegmentTree(int n) : data(2 * higherPowerOf2(n), 0) {}

    int query(int i, int a, int b, int x, int y) {
        if (b <= x || a >= y) return 0;
        if (b <= y && a >= x) return data[i];

        return query(i*2,   a, (a+b)/2, x, y)
             + query(i*2+1, (a+b)/2, b, x, y);
    }

    int update(int i, int a, int b, int x, int v) {
        if (x < a || x >= b) return data[i];
        if (a == b-1) {
            assert(a == x);
            return data[i] = v;
        }

        return data[i] = update(i*2,   a, (a+b)/2, x, v)
                       + update(i*2+1, (a+b)/2, b, x, v);
    }

    int query(int x, int y) {
        assert(x <= y);
        return query(1, 0, data.size()/2, x, y);
```

```
    }

    void update(int x, int v) {
        update(1, 0, data.size()/2, x, v);
    }
};
```

Segment tree con lazy propagation

```
enum class Mode : char { none, add, set };
struct Node {
   ll min = numeric_limits<ll>::max();
   ll sum = 0;

   ll update = 0;
   Mode mode = Mode::none;
};
void setup(const vector<ll>& v, int a, int b, int i) {
   if (b-a == 1) {
      if (a < (ll)v.size()) {
         dat[i].min = v[a];
         dat[i].sum = v[a];
      }
      return;
   }
   setup(v, a, (a+b)/2, i*2);
   setup(v, (a+b)/2, b, i*2+1);
   setup(i);
}
void setup(int i) {
   if (i2 >= (ll)dat.size()) return;
   dat[i].min = min(dat[i*2].min, dat[i*2+1].min);
   dat[i].sum = dat[i*2].sum + dat[i*2+1].sum;
}
void lazyPropStep(int a, int b, int i, ll update, Mode mode) {
   if(mode == Mode::none) {
      return;
   } else if (mode == Mode::add) {
      if (dat[i].mode == Mode::none) {
         dat[i].update = 0; // just in case
      }
      dat[i].min += update;
      dat[i].sum += (b-a)*update;
      dat[i].update += update;
      if (dat[i].mode == Mode::none) {
         dat[i].mode = Mode::add; // do not change Mode::set
```

```
      }
   } else /* mode == Mode::set */ {
      dat[i].min = update;
      dat[i].sum = (b-a)*update;
      dat[i].update = update;
      dat[i].mode = Mode::set;
   }
}
void lazyProp(int a, int b, int i) {
   if (i*2 >= (ll)dat.size()) return;
   lazyPropStep(a, (a+b)/2, i*2,   dat[i].update, dat[i].mode);
   lazyPropStep((a+b)/2, b, i*2+1, dat[i].update, dat[i].mode);
   dat[i].update = 0;
   dat[i].mode = Mode::none;
}
ll queryMin(int l, int r, int a, int b, int i) {
   if (a>=r || b<=l) return numeric_limits<int>::max();
   if (a>=l && b<=r) return dat[i].min;
   lazyProp(a, b, i);
   return min(queryMin(l, r, a, (a+b)/2, i*2),
              queryMin(l, r, (a+b)/2, b, i*2+1));
}
ll querySum(int l, int r, int a, int b, int i) {
   if (a>=r || b<=l) return 0;
   if (a>=l && b<=r) return dat[i].sum;
   lazyProp(a, b, i);
   return querySum(l, r, a, (a+b)/2, i*2)
        + querySum(l, r, (a+b)/2, b, i*2+1);
}
void lazyAdd(int l, int r, ll x, int a, int b, int i) {
   if (a>=r || b<=l) return;
   lazyProp(a, b, i);
   if (a>=l && b<=r) {
      dat[i].min += x;
      dat[i].sum += (b-a)*x;
      dat[i].update = x;
      dat[i].mode = Mode::add;
      return;
   }
   lazyAdd(l, r, x, a, (a+b)/2, i*2);
   lazyAdd(l, r, x, (a+b)/2, b, i*2+1);
   setup(i);
}
void lazySet(int l, int r, ll x, int a, int b, int i) {
   if (a>=r || b<=l) return;
   lazyProp(a, b, i);
```

```
    if (a>=l && b<=r) {
        dat[i].min = x;
        dat[i].sum = (b-a)*x;
        dat[i].update = x;
        dat[i].mode = Mode::set;
        return;
    }
    lazySet(l, r, x, a, (a+b)/2, i*2);
    lazySet(l, r, x, (a+b)/2, b, i*2+1);
    setup(i);
}
```

Fenwick tree

```
int leastSignificantOneBit(int i){
    return i & (-i);
}

struct FenwickTree {
    vector<int> data;
    FenwickTree(int N) : data(N) {}
    void add(int pos, int value) {
        if (pos>=data.size()) return;
        data[pos] += value;
        add(pos + leastSignificantOneBit(pos), value);
    }
    int sumUpTo(int pos) {
        if (pos==0) return 0;
        return data[pos] + sumUpTo(pos - leastSignificantOneBit(pos));
    }
};
```

Sparse table

- Per ogni elemento di un array applico l'operazione ai range `[0,1)`, `[0,2)`, `[0,4)`, `...` (potenze di 2) e salvo il valore in un array `st[N][32]`
- (vale per operazioni idempotenti, i.e. `a op a = a`) per trovare il valore nel range `[l,r)` in `O(1)` basta trovare `k = max(k_ tali che 2^k_ <= r-l)` e poi il risultato della query è `st[l][k] op st[r-(1LL<<k)][k]`

Algoritmi vari

Zaino / 0-1 knapsack

- In input ci sono il numero di oggetti `N`, la capienza dello zaino `C`, i pesi `W[N]` e i valori `V[N]`

- Caso base `f(N,c) = 0`
- Caso ricorsivo `f(i,c) = (W[i] <= c ? max(f(i+1, c), f(i+1, c-W[i]) + V[i]) : f(i+1, c))`
- Risultato è `f(0,C)`
- Usare `mem[i][c]` per salvare risultati e fare DP

LIS, Longest Increasing Subsequence

```
int main() {
    int N;
    cin>>n;
    vector<int> pesi(N), ultimoPreso(N+1, numeric_limits<int>::max());
    ultimoPreso[0]=0;

    for(int i=0; i<N; i++) cin>>pesi[i];

    for(auto p : pesi) {
        auto it = lower_bound(ultimoPreso.begin(), ultimoPreso.end(), p);
        *it = p;
    }

    auto it = lower_bound(ultimoPreso.begin(), ultimoPreso.end(), numeric_limits<int>::max(
    cout << (it - ultimoPreso.begin() - 1);
}
```

Matematica

Fast exponentiation

```
int fastExp(int x, int e){
   if (e==0) return 1;
   int half = fastExp(x, e/2);
   return ((half*half % M) * (e%2 == 1 ? x : 1)) % M;
}
```

Euclide esteso

`A/B=d` & `A%B=C` => `Bx+Cy = 1` con `y=-x` e `x = dx - y`

Fermat & inverso moltiplicativo

inverso di `A` = `A^(M-2)%M` = `fastExp(A, M-2)`

Rabin Karp hash

```
#define hash_t uint64_t
#define M 1000000007
```

```
#define P 59

hash_t getHash(const char* s, size_t l) {
   if (l==0) return 0;
   return (P*getHash(s+1, l-1) + s[0]) % M;
}

signed main() {
   array<int, 4002> Pexp;
   array<int, 4002> PexpMulInv;

   int p=1;
   for(int i=0;i<(int)Pexp.size();++i){
      Pexp[i] = p;
      PexpMulInv[i] = fastExp(p, M-2);
      p*=P; p%=M;
   }

   // calculate hashes for strings in S from 0 to any l
   vector<hash_t> hashes(N+1);
   int lasth=0;
   for(size_t l=0;l<N;++l){
      hashes[l]=lasth;
      lasth+=Pexp[l]*S[l];
      lasth%=M;
   }
   hashes[N]=lasth;

   // obtain the hash of s in range [n, n+l) with prefix sum
   hash_t hcmp = (((hashes[n + l] - hashes[n] + M) % M) * PexpMulInv[n]) % M;
}
```

Geometria

Vettori

- Prodotto vettore: `A^B = A.x*B.y - A.y*B.x = |A|*|B|*sin(angolo fra A e B)`
  - Proprietà: `A^B = -B^A`, `A^A = 0`, `A^(B+C) = A^B + A^C`
- `(0,0)`, `A` e `B` allineati sse `A^B = 0`
- Data retta orientata `AB` e punto `C`, il prodotto `p=(A-C)^(B-C)` indica
  - che `C` è: sulla retta sse `p=0`, a destra della retta sse `p<0`, a sinistra della retta sse `p>0`
  - che `A`, `B` e `C` sono: in ordine orario sse `p<0`, antiorario sse `p>0`
  - **Area** triangolo `ABC` = `|p|/2` (senza `/2` per parallelogramma)

  - **Distanza** di `C` da `AB` = `|p|/(B-A)`
- Area poligono `P_0, P_1, ..., P_n-1` = `1/2 * |P_0 ^ P_1 + P_1 ^ P_2 + ... + P_n-2 ^ P_n-1 + P_n-1 ^ P0|`
- Per vedere se `P` è dentro il poligono *convesso* `P_0, P_1, ..., P_n-1`: controllare se `P` sempre dalla stessa parte di tutti i `P_0P_1, P_1P_2, ..., P_n-1P_0`
- Per vedere se `P` è dentro un poligono *concavo*: controllare `#` intersezioni della semiretta `PQ` con `Q` scelto a caso molto grande: se `#` pari `P` è esterno, se `#` dispari è interno
- `AB` e `CD` si intersecano sse (`C` e `D` da parti opposte di `AB`) e (`A` e `B` da parti opposte di `CD`)
  - Intersezioni mantenute con trasformazioni lineari
- **Ordinare** punti per **angolo**: `sort` con `operator<(P,Q) = P^Q < 0`

Convex Hull

- trovare il punto più in basso (P0)
- ordinare per angolo rispetto a P0 usando `(P-P0)^(Q-P0)`, vedi sopra (NlogN)
- andare avanti, e buttare in uno stack il punto che si trova
- se l'angolo tra gli ultimi 3 è ottuso rimuovo l'elemento centrale dallo stack e ripeto

FFT Fast Fourier Transform

- gestisco numeri complessi (complex della lib standard)
- devo valutare la funzione su `2N` punti circa (`e^((O*2pi)/2n)`)
- visto che alcuni sono sicuramente >0 posso dividergli dagli altri, elevarli ^2 e ripetere (per costruzione funziona)
- moltiplicare i valori dei 2 polinomi
- riapplicare la FFT ma con -O

Utilities

Date e tempo

```
struct std::tm tmp;

ll readDate(){
    cin>>get_time(&tmp, "%Y-%m-%d %H:%M");
    return chrono::system_clock::from_time_t(mktime(&tmp)).time_since_epoch().count();
}
```

STL

- `operator<` Deve ritornare `false` in caso di uguaglianza!

- `priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int, int>>>`
- `auto cmp = [](const T& a, const T& b){ return /* ... */; };`
  `priority_queue<T,vec<T>,decltype(cmp)> pq{cmp};`

In caso di errore

Wrong answer WA

- `#define int long long`
  - `1LL` invece che `1`
- Se ci sono moduli, vengono fatti dappertutto?
- Stampa la tua soluzione!
- Stai cancellando tutte le strutture di dati tra i casi di test?
- Il tuo algoritmo può gestire l'intera gamma di input?
- Leggi di nuovo l'intero testo del problema.
- Il tuo formato di output è corretto? (inclusi gli spazi bianchi)
- Gestisci correttamente tutti i corner case?
- Hai compreso correttamente il problema?
- Sei sicuro che il tuo algoritmo funzioni?
- A quali casi speciali non hai pensato?
- Sei sicuro che le funzioni STL che usi funzionino come pensi?
- Crea alcuni casi di prova su cui far girare il tuo algoritmo.
- Esegui l'algoritmo per un caso semplice.
- Ripassa questa lista.
- Spiega il tuo algoritmo ad un compagno di squadra.
- Chiedi al compagno di squadra di guardare il tuo codice.
- Vai a fare una piccola passeggiata, per esempio al bagno.
- Riscrivi la tua soluzione dall'inizio o fallo fare ad un compagno di squadra.

Runtime error RE

- Hai testato tutti i corner case localmente?
- Stai accedendo ad indici out of bound di qualche vettore? (usa `.at()`)
- Qualche possibile divisione per 0? (mod 0 per esempio)
- Qualsiasi possibile ricorsione infinita?

Time limit exceeded TLE

- `#pragma GCC optimize("O3")`
- Hai qualche possibile loop infinito?
- Qual è la complessità del tuo algoritmo?
- Stai copiando molti dati non necessari? (usa le reference)
- Quanto è grande l'input e l'output?
- Cosa pensano i tuoi compagni di squadra del tuo algoritmo?

Memory limit exceeded MLE

- Qual è la quantità massima di memoria di cui il vostro algoritmo dovrebbe avere bisogno?
- Ci sono memory leak? (usa la STL invece)
- Stai cancellando tutte le strutture di dati tra un test e l'altro?

Output limit exceeded OLE

- Avete rimosso tutte le stampe di debug?
- Il vostro ciclo di output può andare in tilt?

Fft double

- può avere problemi di approssimazione con numeri grandi, ricordarsi di arrotondare bene
- non mi va con cose negative
- (non me ne prendo responsabilità)

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

using namespace std;

#define _USE_MATH_DEFINES
#include <complex>
#include <vector>
#include <cmath>

std::vector<std::complex<double> > fast_fourier_transform(std::vector<std::complex<double>

    std::vector<std::complex<double> > w(x.size(), 0.0);
    w[0] = 1.0;
    for(int pow_2 = 1; pow_2 < (int)x.size(); pow_2 *= 2) {
        w[pow_2] = std::polar(1.0, 2*M_PI * pow_2/x.size() * (inverse ? 1 : -1) );
    }
    for(int i=3, last=2; i < (int)x.size(); i++) {
        if(w[i] == 0.0) {
            w[i] = w[last] * w[i-last];
        } else {
            last = i;
        }
    }

    for(int block_size = x.size(); block_size > 1; block_size /= 2) {
        std::vector<std::complex<double> > new_x(x.size());

        for(int start = 0; start < (int)x.size(); start += block_size) {
```

```
            for(int i=0; i<block_size; i++) {
                new_x[start + block_size/2 * (i%2) + i/2] = x[start + i];
            }
        }
        x = new_x;
    }

    for(int block_size = 2; block_size <= (int)x.size(); block_size *= 2) {
        std::vector<std::complex<double> > new_x(x.size());
        int w_base_i = x.size() / block_size;

        for(int start = 0; start < (int)x.size(); start += block_size) {
            for(int i=0; i < block_size/2; i++) {
                new_x[start+i]              = x[start+i] + w[w_base_i*i] * x[start + block
                new_x[start+block_size/2+i] = x[start+i] - w[w_base_i*i] * x[start + block
            }
        }
        x = new_x;
    }
    return x;
}

struct Polynomial {
    std::vector<double> a;
    Polynomial(std::vector<double> new_a) : a(new_a) {}

    Polynomial operator*(Polynomial r) {
        int power_2 = 1;
        while(power_2 < (int)(a.size() + r.a.size() - 1)) {
            power_2 *= 2;
        }

        std::vector<std::complex<double> > x_l(power_2, 0.0);
        std::vector<std::complex<double> > x_r(power_2, 0.0);
        std::vector<std::complex<double> > product(power_2, 0.0);

        for(int i=0; i<(int)a.size(); i++) {
            x_l[i] = a[i];
        }
        for(int i=0; i<(int)r.a.size(); i++) {
            x_r[i] = r.a[i];
        }
        x_l = fast_fourier_transform(x_l);
        x_r = fast_fourier_transform(x_r);
        for(int i=0; i<power_2; i++) {
          product[i] = x_l[i] * x_r[i];
```

```
        }
        product = fast_fourier_transform(product, true);

        std::vector<double> result_a(a.size() + r.a.size() - 1);
        for(int i=0; i<(int)result_a.size(); i++) {
            result_a[i] = product[i].real() / power_2;
        }
        return result_a;
    }
};

int main() {
  vector<double> t(100000);
  for(int i=0; i<100000; i++) t[i]=i;
    Polynomial x_1(t);
    Polynomial x_2({2, 0, 1});
    Polynomial result = x_1 * x_2;

    ofstream out("output.txt");
    for(int i=0; i<result.a[i]; i++){
      out << (long long)(result.a[i] + 0.5 - (result.a[i]<0)) << " ";
    }
    return 0;
}
```

fft modulo M

- accetta numeri negativi, bisogna stare attenti ai moduli
- (non me ne prendo responsabilità)

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

#define N 100001
#define L 18  /* L = ceil(log2(N * 2 - 1)) */
#define N_    (1 << L)
#define MD    469762049   /* MD = 56 * 2^23 + 1 */

int *wu[L + 1], *wv[L + 1];

int power(int a, int k) {
  long long b = a, p = 1;

  while (k) {
    if (k & 1)
      p = p * b % MD;
```

```
    b = b * b % MD;
    k >>= 1;
  }
  return p;
}
void init() {
  int l, i, u, v;
  u = power(3, (MD - 1) >> L);
  v = power(u, MD - 2);

  for (l = L; l > 0; l--) {
    int n = 1 << (l - 1);

    wu[l] = (int *) malloc(n * sizeof *wu[l]);
    wv[l] = (int *) malloc(n * sizeof *wv[l]);

    wu[l][0] = wv[l][0] = 1;
    for (i = 1; i < n; i++) {
      wu[l][i] = (long long) wu[l][i - 1] * u % MD;
      wv[l][i] = (long long) wv[l][i - 1] * v % MD;
    }

    u = (long long) u * u % MD, v = (long long) v * v % MD;
  }
}
void ntt_(int *aa, int l, int inverse) {
  if (l > 0) {
    int n = 1 << l;
    int m = n >> 1;
    int *ww = inverse ? wv[l] : wu[l];
    int i, j;
    ntt_(aa, l - 1, inverse);
    ntt_(aa + m, l - 1, inverse);
    for (i = 0; (j = i + m) < n; i++) {
      int a = aa[i];
      int b = (long long) aa[j] * ww[i] % MD;
      if ((aa[i] = a + b) >= MD)
        aa[i] -= MD;
      if ((aa[j] = a - b) < 0)
        aa[j] += MD;
    }
  }
}
void ntt(int *aa, int l, int inverse) {
  int n_ = 1 << l, i, j;
  for (i = 0, j = 1; j < n_; j++) {
```

```
      int b;
      int tmp;
      for (b = n_ >> 1; (i ^= b) < b; b >>= 1)
        ;
      if (i < j)
        tmp = aa[i], aa[i] = aa[j], aa[j] = tmp;
    }
    ntt_(aa, l, inverse);
  }
  void mult(int *aa, int n, int *bb, int m, int *out) {
    static int aa_[N_], bb_[N_];
    int l, n_, i, v;
    l = 0;
    while (1 << l <= n - 1 + m - 1)
      l++;
    n_ = 1 << l;
    memcpy(aa_, aa, n * sizeof *aa), memset(aa_ + n, 0, (n_ - n) * sizeof *aa_);
    memcpy(bb_, bb, m * sizeof *bb), memset(bb_ + m, 0, (n_ - m) * sizeof *bb_);
    ntt(aa_, l, 0), ntt(bb_, l,  0);
    for (i = 0; i < n_; i++)
      out[i] = (long long) aa_[i] * bb_[i] % MD;
    ntt(out, l, 1);
    v = power(n_, MD - 2);
    for (i = 0; i < n_; i++)
      out[i] = (long long) out[i] * v % MD;
  }
  int main() {
    static int aa[N], bb[N], out[N_];
    int n, m, i;
    init();
    scanf("%d%d", &n, &m), n++, m++;
    for (i = 0; i < n; i++)
      scanf("%d", &aa[i]);
    for (i = 0; i < m; i++)
      scanf("%d", &bb[i]);
    mult(aa, n, bb, m, out);
    for (i = 0; i < n + m - 1; i++)
      printf("%d ", out[i]);
      printf("\n");
    return 0;
  }
```

Euclide esteso, ma iterativo (che è un po' più veloce)

```
int gcd(int a, int b, int& x, int& y) {
    x = 1, y = 0;
```

```
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}
```

Convex hull

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}
bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }
void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
```

```
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }
    a = st;
}
```

Spfa

- ricordarsi di aggiungere limite al numero di esecuzioni se possono esserci cicli negativi, altrimenti va all'infinito

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

bool spfa(int s, vector<int>& d) {
    int n = adj.size();
    d.assign(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;
    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        inqueue[v] = false;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                if (!inqueue[to]) {
                    q.push(to);
                    inqueue[to] = true;
                    cnt[to]++;
                    if (cnt[to] > n)
                        return false;  // negative cycle
                }
            }
        }
    }
    return true;
```

```
}
```

Mst n^2

```cpp
int n;
vector<vector<int>> adj; // adjacency matrix of graph
const int INF = 1000000000; // weight INF means there is no edge

struct Edge {
    int w = INF, to = -1;
};
void prim() {
    int total_weight = 0;
    vector<bool> selected(n, false);
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }
    cout << total_weight << endl;
}
```

Mst mlog(n)

```cpp
vector<int> parent, rank;
void make_set(int v) {
    parent[v] = v;
```

```
    rank[v] = 0;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<Edge> result;
parent.resize(n);
rank.resize(n);
for (int i = 0; i < n; i++)
    make_set(i);

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
        cost += e.weight;
        result.push_back(e);
        union_sets(e.u, e.v);
    }
}
```

Matexp

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

const int N = 3;

const long long M = 1000000007;

void multiply (long long A[N][N], long long B[N][N]){
    long long R[N][N];

    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            R[i][j] = 0;
            for (int k = 0; k < N; k++){
                R[i][j] = (R[i][j] + A[i][k] * B[k][j]) % M;
            }
        }
    }

    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            A[i][j] = R[i][j];
        }
    }
}

void power_matrix (long long A[N][N], int n){
    long long B[N][N];

    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            B[i][j] = A[i][j];
        }
    }

    n = n - 1;
    while (n > 0)
    {
        if (n & 1)
            multiply (A, B);

        multiply (B,B);

        n = n >> 1;
```

```
    }
}

long long solve_recurrence (long long A[N][N], long long B[N][1], int n){
    if (n < N)
        return B[N - 1 - n][0];

    power_matrix (A, n - N + 1);

    long long result = 0;

    for (int i = 0; i < N; i++)
        result = (result + A[0][i] * B[i][0]) % M;

    return result;
}

int main ()
{

    long long A[N][N] = {{2, 1, 3}, {1, 0, 0}, {0, 1, 0}};
    long long B[N][1] = {{3}, {2}, {1}};

    int n = 5;

    long long R_n = solve_recurrence (A, B, n);

    cout << "R_" << n << " = " << R_n;

    return 0;
}
```

Gauss

```
const double EPS = 1e-9;
const int INF = 2; // no need for infinity

int gauss (vector < vector<double> > a, vector<double> & ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
```

```
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```