```
        int new_capacity = capacity << 1;
        a = new T[new_capacity];
        for (int i = 0; i < length; ++i) {
           a[i] = old[(i + start) % capacity];
        }
        delete[] old;
        capacity = new_capacity;
        start = 0;
     }
     a[(start + length) % capacity] = v;
     ++length;
   }

   int size() { return length; }
};
```

1.3 Priority Queue

```
#include <cassert>
#include <iostream>

#include "vector.cpp"
using namespace std;

template <typename T>
class vector {
  private:
   T* a;
   int capacity;
   int length;

  public:
   vector() : capacity(16), length(0), a(new T[16]) {}
   vector(int n) : capacity(n), length(n), a(new T[n]) {}
   vector(int n, T v) : capacity(n), length(n), a(new T[n]) {
      for (int i = 0; i < length; ++i) {
         a[i] = v;
      }
   }
   ~vector() { delete[] a; }

   void push_back(T v) {
      if (length == capacity) {
         T* old = a;
         capacity <<= 1;
         a = new T[capacity];
         for (int i = 0; i < length; ++i) a[i] = old[i];
         delete[] old;
      }
      a[length++] = v;
   }

   T pop_back() { return a[length-- - 1]; }

   T& back() { return a[length - 1]; }

   int size() { return length; }

   T& operator[](int index) { return a[index]; }
};

// min heap
template <typename T>
class priority_queue {
  private:
   vector<T> a;

  public:
   void push(T v) {
      a.push_back(v);
      if (a.size() == 1) {
         return;
      }
      int cur = a.size() - 1;
      while (cur) {
         int parent = (cur - 1) >> 1;
         if (a[parent] > a[cur]) {
            T temp = a[cur];
            a[cur] = a[parent];
            a[parent] = temp;
            cur = parent;
         } else {
            break;
         }
      }
   }

   T top() {
      assert(a.size());
      return a[0];
   }

   void pop() {
      assert(a.size());
      T ans = a[0];
      a[0] = a.pop_back();
      int here = 0;
      int left = 1;
      while (left < a.size()) {
         int right = left + 1;
         int minchild = left;
         T min = a[left];
         T h = a[here];
         if (right < a.size()) {
            if (min > a[right]) {
               minchild = right;
               min = a[right];
            }
         }
         if (h > min) {
            a[here] = min;
            a[minchild] = h;
            here = minchild;
            left = (minchild << 1) | 1;
         } else {
            break;
         }
      }
   }

   int size() { return a.size(); }
};

int main() {
   int n = 1000000;
   priority_queue<int> p;
   for (int i = 1; i < n; ++i) {
      int x = ((i << 1) * 18719383 | (i << 10)) % 12799;
      cout << "pushing " << x << "\n";
      p.push(x);
      cout << p.top() << '\n';
   }
   for (int i = 1; i < n; ++i) {
      cout << "popping " << p.top() << '\n';
      p.pop();
   }
}
```

1.4 Pair

```
#include <iostream>

template <typename T, typename V>
class pair {
  public:
   T F;
   V S;
   bool operator<(const pair<T, V> p) {
      if (F < p.F) return true;
      if (F == p.F) return S < p.S;
      return false;
   }
   bool operator==(const pair<T, V> p) { return (F == p.F) && (S == p.S); }
};

int main() { return 0; }
```

1.5 Fenwick

```
struct Fenwick {
   int n;
   vector<int> t;
   Fenwick(int n) : n(n), t(n + 1) {}

   // prefix_sum[0..i]
   int query(int i) {
      int s = 0;
      while (i) {
         s += t[i];
         i -= i & (-i);
      }
      return s;
   }

   // increase a[i] by v
   void update(int v, int i) {
      while (i <= n) {
         t[i] += v;
         i += i & (-i);
      }
   }
};
```

1.6 Segment Tree

```
struct SegTree {
   // datatype of nodes of segment tree
   typedef int T;
   // datatype of vector that's generating the segment tree
   typedef int S;
   // identity element of monoid
   // if you have any issues with unit, define it outside the struct as a
   // normal variable
   static constexpr T unit = 0;
   // node of segment tree from a value
   T make_node(S val) { return val; }
   // combine function - needs to be an associative function
   T combine(T a, T b) { return a + b; }
   // point update function - updating the element in the array
   void update_val(T& a, S b) { a += b; }

   vector<T> t;
   int32_t n;

   SegTree(int32_t n = 0, T def = unit) : t(n << 1, def), n(n) {}
   SegTree(vector<S>& a, T def = unit) {
      n = a.size();
      t.assign(n << 1, unit);
      for (int32_t i = 0; i < n; ++i) {
         t[i + n] = make_node(a[i]);
      }
```

```
      for (int32_t i = n - 1; i; --i) {
         t[i] = combine(t[i << 1], t[i << 1 | 1]);
      }
   }

   void update(int32_t pos, S val) {
      for (update_val(t[pos += n], val); pos >>= 1;) {
         t[pos] = combine(t[pos << 1], t[pos << 1 | 1]);
      }
   }

   T query(int32_t l, int32_t r) {
      T ra = unit, rb = unit;
      for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
         if (l & 1) ra = combine(ra, t[l++]);
         if (r & 1) rb = combine(t[--r], rb);
      }
      return combine(ra, rb);
   }
};

typedef struct {
   // datatype
} node;

const node ID;
node t[4 * maxn];

node combine(node n1, node n2) {
   node ans;
   // do something
   return ans;
}

node make_node(int val) {
   // make a node from val and return
}

// build segtree - 1 indexed
void build(int v, int l, int r, vector<int>& a) {
   if (l == r) {
      t[v] = make_node(a[l]);
      return;
   }
   int mid = (l + r) >> 1;
   build(v << 1, l, mid, a);
   build((v << 1) | 1, mid + 1, r, a);
   t[v] = combine(t[(v << 1)], t[(v << 1) | 1]);
}

// update segtree by updating value to val at idx
void update(int v, int l, int r, int idx, int val) {
   if (l == r) {
      t[v] = make_node(val);
      return;
   }
   int mid = (l + r) >> 1;
   if (idx <= mid)
      update(v << 1, l, mid, idx, val);
   else
      update((v << 1) | 1, mid + 1, r, idx, val);
   t[v] = combine(t[v << 1], t[(v << 1) | 1]);
}

// range query from l to r both inclusive
node query(int v, int tl, int tr, int l, int r) {
   if (l > r) return ID;
   if (l == tl && r == tr) {
      return t[v];
   }
   int tm = (tl + tr) >> 1;
   return combine(query(v << 1, tl, tm, l, min(r, tm)),
                  query((v << 1) | 1, tm + 1, tr, max(l, tm + 1), r));
}

// slightly more efficient range query
node query2(int v, int tl, int tr, int l, int r) {
   if (l == tl && r == tr) return t[v];
   int tm = (tl + tr) >> 1;
   if (l > tm) return query((v << 1) | 1, tm + 1, tr, l, r);
   if (tm + 1 > r) return query(v << 1, tl, tm, l, r);
   return combine(query(v << 1, tl, tm, l, tm),
                  query((v << 1) | 1, tm + 1, tr, tm + 1, r));
}
```

1.7 Sparse Table

```
// sparse table
const int N = 1e6 + 6;
const int Log = 26;

int sparse_table[Log][N];

void build_sparse_table(vector<int>& a) {
   int n = a.size();
   for (int i = 0; i < n; ++i) {
      sparse_table[0][i] = a[i];
   }
   for (int j = 0; j < Log - 1; ++j) {
      for (int i = 0; i + (2 << j) <= n; ++i) {
         sparse_table[j + 1][i] =
            min(sparse_table[j][i], sparse_table[j][i + (1 << j)]);
      }
   }
}

//[l, r)
int query(int l, int r) {
   int sz = __lg(r - l);
   return min(sparse_table[sz][l], sparse_table[sz][r - (1 << sz)]);
}
```

1.8 Binary Trie

```
struct trie {
   bool isleaf;
   trie* child[2];
};

trie* create() {
   trie* t = new trie();
   t->isleaf = false;
   memset(t->child, 0, sizeof t->child);
   return t;
}

void add(trie* root, int n) {
   int p = 0;
   for (int i = 31; ~i; --i) {
      p = (n >> i) & 1;
      if (root->child[p] == NULL) {
         root->child[p] = create();
      }
      root = root->child[p];
   }
}

void clean(trie* root) {
#ifdef CLEAN
   if (root == NULL) return;
   clean(root->child[0]);
   clean(root->child[1]);
   delete (root);
#endif
}

int maxxor(trie* root, int n) {
   int ans = 0;
   for (int i = 31; ~i; --i) {
      int p = (n >> i) & 1;
      if (root->child[p ^ 1] != NULL) {
         p ^= 1;
      }
      root = root->child[p];
      ans <<= 1;
      ans |= p ^ ((n >> i) & 1);
   }
   return ans;
}
```

1.9 DSU

```
// dsu implementation
vector<int32_t> par(maxn), siz(maxn);

void make_set(int32_t v) {
   par[v] = v;
   siz[v] = 1;
}

int32_t find_set(int32_t v) {
   if (v == par[v]) return v;
   return par[v] = find_set(par[v]);
}

void union_sets(int32_t a, int32_t b) {
   a = find_set(a);
   b = find_set(b);
   if (a != b) {
      if (siz[a] < siz[b]) swap(a, b);
      par[b] = a;
      siz[a] += siz[b];
   }
}
```

1.10 AVL Tree

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

template <typename T>
class avl {
  private:
   struct node {
      T data;
      node *l, *r;
      int ht;
      node(T v) : data(v), l(NULL), r(NULL), ht(0) {}
   };

   node* root;

   void clear(node* t) {
      if (t == NULL) return;
      clear(t->l);
      clear(t->r);
      delete t;
   }

   int height(node* t) {
      if (t == NULL) return -1;
      return t->ht;
```

```
  }

  int balance(node* t) {
     if (t == NULL) return 0;
     return height(t->l) - height(t->r);
  }

  node* right_rot(node*& t) {
     if (t->l != NULL) {
        node* u = t->l;
        t->l = u->r;
        u->r = t;
        t->ht = 1 + max(height(t->l), height(t->r));
        u->ht = 1 + max(height(u->l), t->ht);
        return u;
     }
     return t;
  }

  node* left_rot(node*& t) {
     if (t->r != NULL) {
        node* u = t->r;
        t->r = u->l;
        u->l = t;
        t->ht = 1 + max(height(t->l), height(t->r));
        u->ht = 1 + max(height(u->r), t->ht);
        return u;
     }
     return t;
  }

  node* double_left_rot(node*& t) {
     t->r = right_rot(t->r);
     return left_rot(t);
  }

  node* double_right_rot(node*& t) {
     t->l = left_rot(t->l);
     return right_rot(t);
  }

  node* begin(node* t) {
     if (t == NULL || (t->l == NULL)) return t;
     return begin(t->l);
  }

  node* end(node* t) {
     if (t == NULL || (t->r == NULL)) return t;
     return end(t->r);
  }

  node* _insert(node* t, T x) {
     if (t == NULL) {
        return new node(x);
     }

     if (x < t->data) {
        t->l = _insert(t->l, x);
        if (height(t->l) - height(t->r) == 2) {
           if (x < t->l->data) {
              t = right_rot(t);
           } else {
              t = double_right_rot(t);
           }
        }
     } else if (x > t->data) {
        t->r = _insert(t->r, x);
        if (height(t->r) - height(t->l) == 2) {
           if (x > t->r->data) {
              t = left_rot(t);
           } else {
              t = double_left_rot(t);
           }
        }
     } else {
        return t;
     }

     t->ht = 1 + max(height(t->l), height(t->r));
     return t;
  }

  node* _remove(node* t, T x) {
     node* temp;

     if (t == NULL) {
        return t;
     } else if (x < t->data) {
        t->l = _remove(t->l, x);
     } else if (x > t->data) {
        t->r = _remove(t->r, x);
     } else if (t->l && t->r) {
        temp = begin(t->r);
        t->data = temp->data;
        t->r = _remove(t->r, x);
     } else {
        temp = t;
        if (t->l == NULL)
           t = t->r;
        else if (t->r == NULL)
           t = t->l;
        delete temp;
     }

     if (t == NULL) {
        return t;
     }

     t->ht = 1 + max(height(t->l), height(t->r));

     int b = balance(t);
     if (b > 1) {
        if (balance(t->l) >= 0) {
           return right_rot(t);
        } else {
           return double_right_rot(t);
        }
     } else if (b < -1) {
        if (balance(t->r) <= 0) {
           return left_rot(t);
        } else {
           return double_left_rot(t);
        }
     }

     return t;
  }

  void print(node* t) {
     if (t == NULL) return;
     print(t->l);
     std::cout << t->data << ' ';
     print(t->r);
  }

 public:
  avl() : root(NULL) {}
  ~avl() { clear(root); }
  void insert(T x) { root = _insert(root, x); }
  void erase(T x) { root = _remove(root, x); }
  void print() {
     print(root);
     std::cout << '\n';
  }
};

int main() {
   avl<int> t;
   for (int i = 0; i < 100000; ++i) {
      t.insert(rand() % 1000000);
      // t.print();
      t.erase(rand() % 1000000);
   }
   // t.print();
}
```

1.11 Matrix

```
// matrix library

template <typename T>
struct Matrix {
   int32_t rows, cols;
   vector<vector<T>> mat;
   Matrix(int32_t r, int32_t c)
      : rows(r), cols(c), mat(vector<vector<T>>(r, vector<T>(c))){};
   void fill(T val) {
      for (int32_t i = 0; i < rows; i++) {
         for (int32_t j = 0; j < cols; j++) {
            mat[i][j] = val;
         }
      }
   }
   void reset() { fill(0); }
   void setid() {
      assert(rows == cols);
      for (int32_t i = 0; i < rows; i++) {
         mat[i][i] = 1;
      }
   }
   static Matrix id(int32_t n) {
      Matrix m(n, n);
      m.setid();
      return m;
   }
   Matrix operator+(const Matrix& a) const {
      assert(rows == a.rows && cols == a.cols);
      Matrix<T> res(rows, cols);
      for (int32_t i = 0; i < rows; i++) {
         for (int32_t j = 0; j < cols; j++) {
            res.mat[i][j] = mat[i][j] + a.mat[i][j];
         }
      }
   }
   Matrix<T> operator*(const Matrix<T>& a) const {
      assert(cols == a.rows);
      Matrix<T> res(rows, a.cols);
      for (int32_t i = 0; i < rows; i++) {
         for (int32_t j = 0; j < a.cols; j++) {
            res.mat[i][j] = 0;
            for (int32_t k = 0; k < cols; k++) {
               res.mat[i][j] += mat[i][k] * a.mat[k][j];
            }
         }
      }
      return res;
   }
   void operator+=(const Matrix& a) { *this = *this + a; }
   void operator*=(const Matrix& a) { *this = *this * a; }
};
```

2 Graphs

2.1 Bridges

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
   visited[v] = true;
   tin[v] = low[v] = timer++;
   for (int to : adj[v]) {
      if (to == p) continue;
      if (visited[to]) {
         low[v] = min(low[v], tin[to]);
      } else {
         dfs(to, v);
         low[v] = min(low[v], low[to]);
         if (low[to] > tin[v]) IS_BRIDGE(v, to);
      }
   }
}

void find_bridges() {
   timer = 0;
   visited.assign(n, false);
   tin.assign(n, -1);
   low.assign(n, -1);
   for (int i = 0; i < n; ++i) {
      if (!visited[i]) dfs(i);
   }
}
```

2.2 Undirected cycle

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v, int par) { // passing vertex and its parent vertex
   color[v] = 1;
   for (int u : adj[v]) {
      if (u == par) continue; // skipping edge to parent vertex
      if (color[u] == 0) {
         parent[u] = v;
         if (dfs(u, parent[u])) return true;
      } else if (color[u] == 1) {
         cycle_end = v;
         cycle_start = u;
         return true;
      }
   }
   color[v] = 2;
   return false;
}

void find_cycle() {
   color.assign(n, 0);
   parent.assign(n, -1);
   cycle_start = -1;

   for (int v = 0; v < n; v++) {
      if (color[v] == 0 && dfs(v, parent[v])) break;
   }
   if (cycle_start == -1) {
      cout << "Acyclic" << endl;
   } else {
      vector<int> cycle;
      cycle.push_back(cycle_start);
      for (int v = cycle_end; v != cycle_start; v = parent[v])
         cycle.push_back(v);
      cycle.push_back(cycle_start);
      reverse(cycle.begin(), cycle.end());

      cout << "Cycle found: ";
      for (int v : cycle) cout << v << " ";
      cout << endl;
   }
}
```

2.3 Topological sort

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
   visited[v] = true;
   for (int u : adj[v]) {
      if (!visited[u]) dfs(u);
   }
   ans.push_back(v);
}

void topological_sort() {
   visited.assign(n, false);
   ans.clear();
   for (int i = 0; i < n; ++i) {
      if (!visited[i]) dfs(i);
   }
   reverse(ans.begin(), ans.end());
}
```

2.4 Strongly connected components

```
vector<vector<int> > g, gr;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
   used[v] = true;
   for (size_t i = 0; i < g[v].size(); ++i)
      if (!used[g[v][i]]) dfs1(g[v][i]);
   order.push_back(v);
}

void dfs2(int v) {
   used[v] = true;
   component.push_back(v);
   for (size_t i = 0; i < gr[v].size(); ++i)
      if (!used[gr[v][i]]) dfs2(gr[v][i]);
}

int main() {
   int n;
   ... reading n... for (;;) {
      int a, b;
      ... reading next edge(a, b)... g[a].push_back(b);
      gr[b].push_back(a);
   }
   used.assign(n, false);
   for (int i = 0; i < n; ++i)
      if (!used[i]) dfs1(i);
   used.assign(n, false);
   for (int i = 0; i < n; ++i) {
      int v = order[n - 1 - i];
      if (!used[v]) {
         dfs2(v);
         ... printing next component... component.clear();
      }
   }
}
```

2.5 Prim's MST

```
const int INF = 1000000000;

struct Edge {
   int w = INF, to = -1;
   bool operator<(Edge const& other) const {
      return make_pair(w, to) < make_pair(other.w, other.to);
   }
};

int n;
vector<vector<Edge>> adj;

void prim() {
   int total_weight = 0;
   vector<Edge> min_e(n);
   min_e[0].w = 0;
   set<Edge> q;
   q.insert({0, 0});
   vector<bool> selected(n, false);
   for (int i = 0; i < n; ++i) {
      if (q.empty()) {
         cout << "No MST!" << endl;
         exit(0);
      }

      int v = q.begin()->to;
      selected[v] = true;
      total_weight += q.begin()->w;
      q.erase(q.begin());

      if (min_e[v].to != -1) cout << v << " " << min_e[v].to << endl;

      for (Edge e : adj[v]) {
         if (!selected[e.to] && e.w < min_e[e.to].w) {
            q.erase({min_e[e.to].w, e.to});
            min_e[e.to] = {e.w, v};
            q.insert({e.w, e.to});
         }
      }
   }

   cout << total_weight << endl;
}
```

2.6 Minimum cost flow

```
struct Edge {
   int from, to, capacity, cost;
};

vector<vector<int>> adj, cost, capacity;

const int INF = 1e9;

void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p) {
```

```
    d.assign(n, INF);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inq[u] = false;
        for (int v : adj[u]) {
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (!inq[v]) {
                    inq[v] = true;
                    q.push(v);
                }
            }
        }
    }
}

int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }

    int flow = 0;
    int cost = 0;
    vector<int> d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INF) break;

        // find max flow on that path
        int f = K - flow;
        int cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur];
        }

        // apply flow
        flow += f;
        cost += f * d[t];
        cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }

    if (flow < K)
        return -1;
    else
        return cost;
}
```

2.7 LCA

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p) {
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i) up[v][i] = up[up[v][i - 1]][i - 1];

    for (int u : adj[v]) {
        if (u != p) dfs(u, v);
    }

    tout[v] = ++timer;
}

bool is_ancestor(int u, int v) {
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v) {
    if (is_ancestor(u, v)) return u;
    if (is_ancestor(v, u)) return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v)) u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

2.8 Floyd Warshall

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

2.9 Dinic

```
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1) continue;
                if (level[edges[id].u] != -1) continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    long long dfs(int v, long long pushed) {
        if (pushed == 0) return 0;
        if (v == t) return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
            if (tr == 0) continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs()) break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
```

2.10 Directed cycle

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
    color[v] = 1;
    for (int u : adj[v]) {
```

```
      if (color[u] == 0) {
         parent[u] = v;
         if (dfs(u)) return true;
      } else if (color[u] == 1) {
         cycle_end = v;
         cycle_start = u;
         return true;
      }
   }
   color[v] = 2;
   return false;
}

void find_cycle() {
   color.assign(n, 0);
   parent.assign(n, -1);
   cycle_start = -1;

   for (int v = 0; v < n; v++) {
      if (color[v] == 0 && dfs(v)) break;
   }

   if (cycle_start == -1) {
      cout << "Acyclic" << endl;
   } else {
      vector<int> cycle;
      cycle.push_back(cycle_start);
      for (int v = cycle_end; v != cycle_start; v = parent[v])
         cycle.push_back(v);
      cycle.push_back(cycle_start);
      reverse(cycle.begin(), cycle.end());

      cout << "Cycle found: ";
      for (int v : cycle) cout << v << " ";
      cout << endl;
   }
}
```

2.11 Dijkstra

```
#include <cassert>
#include <iostream>

using namespace std;

template <typename T>
class vector {
  private:
   T* a;
   int capacity;
   int length;

  public:
   vector() : capacity(4), length(0), a(new T[4]) {}
   vector(int n) : capacity(n), length(n), a(new T[n]) {}
   vector(int n, T v) : capacity(n), length(n), a(new T[n]) {
      for (int i = 0; i < length; ++i) a[i] = v;
   }
   ~vector() { delete[] a; }
   void push_back(T v) {
      if (length == capacity) {
         T* old = a;
         capacity <<= 1;
         a = new T[capacity];
         for (int i = 0; i < length; ++i) {
            a[i] = old[i];
         }
         delete[] old;
      }
      a[length++] = v;
   }
   void pop_back() { --length; }
   T& back() { return a[length - 1]; }
   int size() { return length; }
   T& operator[](int index) { return a[index]; }
};

// min heap
template <typename T>
class priority_queue {
  private:
   vector<T> a;
   void heapify_down(int i) {
      int left = (i << 1) + 1;
      int right = left + 1;
      int mn = i;
      if (left < a.size() && a[left] < a[i]) mn = left;
      if (right < a.size() && a[right] < a[mn]) mn = right;
      if (mn != i) {
         swap(a[i], a[mn]);
         heapify_down(mn);
      }
   }
   void heapify_up(int i) {
      if (i && a[(i - 1) >> 1] > a[i]) {
         swap(a[i], a[(i - 1) >> 1]);
         heapify_up((i - 1) >> 1);
      }
   }

  public:
   priority_queue() : a() {}
   void push(T val) {
      a.push_back(val);
      int cur = a.size() - 1;
      heapify_up(cur);
   }
   T top() { return a[0]; }
   void pop() {
      a[0] = a.back();
      a.pop_back();
      heapify_down(0);
   }

   int size() { return a.size(); }
};

using ll = long long;

template <typename T, typename V>
class Pair {
  public:
   T F;
   V S;
   Pair() : F(), S() {}
   Pair(T f, V s) : F(f), S(s) {}
   bool operator<(const Pair<T, V> p) {
      if (F == p.F) return S < p.S;
      return F < p.F;
   }
   bool operator>(const Pair<T, V> p) {
      if (F == p.F) return S > p.S;
      return F > p.F;
   }
};

const ll INF = 1e18;
const int MAXN = 5e5 + 5;

int p[MAXN];
ll d[MAXN];
vector<Pair<int, ll>> g[MAXN];
vector<Pair<int, int>> path;

int n, m, s, t, u, v;
ll w;

void dijkstra(int s) {
   for (int i = 0; i < n; ++i) {
      d[i] = INF;
      p[i] = -1;
   }
   d[s] = 0;
   priority_queue<Pair<ll, int>> q;
   q.push(Pair<ll, int>(0ll, s));
   while (q.size()) {
      int v = q.top().S;
      ll dv = q.top().F;
      q.pop();
      if (dv != d[v]) continue;
      auto& gv = g[v];
      int sz = gv.size();
      for (int i = 0; i < sz; ++i) {
         int to = gv[i].F;
         ll len = gv[i].S;
         if (dv + len < d[to]) {
            d[to] = dv + len;
            p[to] = v;
            q.push(Pair<ll, int>(d[to], to));
         }
      }
   }
}

int main() {
   ios_base::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   cin >> n >> m >> s >> t;
   for (int i = 0; i < m; ++i) {
      cin >> u >> v >> w;
      g[u].push_back(Pair<int, ll>(v, w));
   }
   dijkstra(s);
   if (d[t] == INF)
      cout << -1 << '\n';
   else {
      cout << d[t];
      while (t != s) {
         path.push_back(Pair<int, int>(p[t], t));
         t = p[t];
      }
      for (int i = 0, j = path.size() - 1; i < j; ++i, --j) {
         auto temp = path[i];
         path[i] = path[j];
         path[j] = temp;
      }
      cout << ' ' << path.size() << '\n';
      for (int i = 0; i < path.size(); ++i) {
         cout << path[i].F << ' ' << path[i].S << '\n';
      }
      cout << '\n';
   }
   return 0;
}
```

2.12 DFS

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices
```

```
vector<int> color;

vector<int> time_in, time_out;
int dfs_timer = 0;

void dfs(int v) {
   time_in[v] = dfs_timer++;
   color[v] = 1;
   for (int u : adj[v])
      if (color[u] == 0) dfs(u);
   color[v] = 2;
   time_out[v] = dfs_timer++;
}
```

2.13 Bellman Ford

```
struct edge {
   int a, b, cost;
};

int n, m, v;
vector<edge> e;
// take INF as longer than any path
const int INF = 1000000000;

void solve() {
   vector<int> d(n, INF);
   d[v] = 0;
   for (int i = 0; i < n - 1; ++i)
      for (int j = 0; j < m; ++j)
         if (d[e[j].a] < INF)
            d[e[j].b] = min(d[e[j].b], d[e[j].a] + e[j].cost);
   // display d, for example, on the screen
}
```

2.14 Articulation Points

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
   visited[v] = true;
   tin[v] = low[v] = timer++;
   int children = 0;
   for (int to : adj[v]) {
      if (to == p) continue;
      if (visited[to]) {
         low[v] = min(low[v], tin[to]);
      } else {
         dfs(to, v);
         low[v] = min(low[v], low[to]);
         if (low[to] >= tin[v] && p != -1) IS_CUTPOINT(v);
         ++children;
      }
   }
   if (p == -1 && children > 1) IS_CUTPOINT(v);
}

void find_cutpoints() {
   timer = 0;
   visited.assign(n, false);
   tin.assign(n, -1);
   low.assign(n, -1);
   for (int i = 0; i < n; ++i) {
      if (!visited[i]) dfs(i);
   }
}
```

2.15 2-SAT

```
int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;

void dfs1(int v) {
   used[v] = true;
   for (int u : g[v]) {
      if (!used[u]) dfs1(u);
   }
   order.push_back(v);
}

void dfs2(int v, int cl) {
   comp[v] = cl;
   for (int u : gt[v]) {
      if (comp[u] == -1) dfs2(u, cl);
   }
}

bool solve_2SAT() {
   used.assign(n, false);
   for (int i = 0; i < n; ++i) {
      if (!used[i]) dfs1(i);
   }

   comp.assign(n, -1);
   for (int i = 0, j = 0; i < n; ++i) {
      int v = order[n - i - 1];
      if (comp[v] == -1) dfs2(v, j++);
   }

   assignment.assign(n / 2, false);
   for (int i = 0; i < n; i += 2) {
      if (comp[i] == comp[i + 1]) return false;
      assignment[i / 2] = comp[i] > comp[i + 1];
   }
   return true;
}
```

2.16 0-1 BFS

```
vector<int> d(n, INF);
d[s] = 0;
deque<int> q;
q.push_front(s);
while (!q.empty()) {
   int v = q.front();
   q.pop_front();
   for (auto edge : adj[v]) {
      int u = edge.first;
      int w = edge.second;
      if (d[v] + w < d[u]) {
         d[u] = d[v] + w;
         if (w == 1)
            q.push_back(u);
         else
            q.push_front(u);
      }
   }
}
```

3 Strings

3.1 Suffix Array

```
// suffix array

vector<int32_t> sort_cyclic_shifts(string const& s) {
   int32_t n = s.size();
   const int32_t alphabet = 128;
   vector<int32_t> p(n), c(n), cnt(max(alphabet, n), 0);
   // base case : length = 1, so sort by counting sort
   for (int32_t i = 0; i < n; i++) cnt[s[i]]++;
   for (int32_t i = 1; i < alphabet; i++) cnt[i] += cnt[i - 1];
   for (int32_t i = 0; i < n; i++) p[--cnt[s[i]]] = i;
   c[p[0]] = 0;
   int32_t classes = 1;
   for (int32_t i = 1; i < n; i++) {
      if (s[p[i]] != s[p[i - 1]]) ++classes;
      c[p[i]] = classes - 1;
   }
   // inductive case, sort by radix sort on pairs (in fact you only need to
   // sort by first elements now)
   vector<int32_t> p_new(n), c_new(n);
   for (int32_t h = 0; (1 << h) < n; ++h) {
      for (int32_t i = 0; i < n; i++) {
         p_new[i] = p[i] - (1 << h);
         if (p_new[i] < 0) p_new[i] += n;
      }
      fill(cnt.begin(), cnt.begin() + classes, 0);
      for (int32_t i = 0; i < n; i++) cnt[c[p_new[i]]]++;
      for (int32_t i = 1; i < classes; i++) cnt[i] += cnt[i - 1];
      for (int32_t i = n - 1; i >= 0; i--) p[--cnt[c[p_new[i]]]] = p_new[i];
      c_new[p[0]] = 0;
      classes = 1;
      for (int32_t i = 1; i < n; i++) {
         pair<int32_t, int32_t> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
         pair<int32_t, int32_t> prev = {c[p[i - 1]],
                                        c[(p[i - 1] + (1 << h)) % n]};
         if (cur != prev) ++classes;
         c_new[p[i]] = classes - 1;
      }
      c.swap(c_new);
   }
   return p;
}
vector<int32_t> suffix_array_construct(string s) {
   s += "$";
   // what about s += " "; ?
   vector<int32_t> sorted_shifts = sort_cyclic_shifts(s);
   // sorted_shifts.erase(sorted_shifts.begin()); - removes the element
   // corresponding to the empty suffix
   return sorted_shifts;
}

// burrow wheeler transform - find the string consisting of the last elements of
// the sorted rotated arrays

// inverse burrow wheeler transform

string s;
read_str(s);
int n = s.size();
vector<int> nextPosition;
vector<vector<int>> positions(27);

for (int i = 0; i < n; ++i) positions[max(0, s[i] - 'a' + 1)].push_back(i);

for (int i = 0; i < 27; ++i)
   for (auto position : positions[i]) nextPosition.push_back(position);
```

```
int position = -1;
for (int i = 0; i < n; ++i) {
   if (s[i] == '#') {
      position = i;
      break;
   }
}

assert(~position);

for (int i = 1; i < n; ++i) {
   position = nextPosition[position];
   write_char(s[position]);
}

write_char('\n');
```

3.2 KMP

```
// The prefix function for this string is defined as an array pi of length n,
// where pi[i] is the length of the longest proper prefix of the substring
// s[0...i] which is also a suffix of this substring.
vector<int> prefix_function(string s) {
   int n = (int)s.length();
   vector<int> pi(n);
   for (int i = 1; i < n; i++) {
      int j = pi[i - 1];
      while (j > 0 && s[i] != s[j]) j = pi[j - 1];
      if (s[i] == s[j]) j++;
      pi[i] = j;
   }
   return pi;
}
```

3.3 Z-function

```
// z[i] is the length of the longest common prefix between s and the suffix of s
// starting at i.

vector<int> z_function(string s) {
   int n = (int)s.length();
   vector<int> z(n);
   for (int i = 1, l = 0, r = 0; i < n; ++i) {
      if (i <= r) z[i] = min(r - i + 1, z[i - l]);
      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
      if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
   }
   return z;
}
```

4 GCD

```
// use only for non-negative u, v
int gcd(int u, int v) {
   int shift;
   if (u == 0) return v;
   if (v == 0) return u;
   shift = __builtin_ctz(u | v);
   u >>= __builtin_ctz(u);
   do {
      v >>= __builtin_ctz(v);
      if (u > v) {
         swap(u, v);
      }
      v -= u;
   } while (v);
   return u << shift;
}

// use only for non-negative u, v
long long gcd(long long u, long long v) {
   int shift;
   if (u == 0 || v == 0) return u + v;
   shift = __builtin_ctzll(u | v);
   u >>= __builtin_ctzll(u);
   do {
      v >>= __builtin_ctzll(v);
      if (u > v) {
         swap(u, v);
      }
      v -= u;
   } while (v);
   return u << shift;
}
```

5 Grid

```
// grid functions

int32_t n, m;

bool check(int32_t i, int32_t j) {
   return (i >= 0) && (i < n) && (j >= 0) && (j < m);
}

vector<pair<int32_t, int32_t>> dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

string direction = "DURL";
```

6 Hash

```
// custom hash

struct custom_hash {
   // http://xorshift.di.unimi.it/splitmix64.c
   static uint64_t splitmix64(uint64_t x) {
      x += 0x9e3779b97f4a7c15;
      x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
      x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
      return x ^ (x >> 31);
   }
   size_t operator()(uint64_t x) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(x + FIXED_RANDOM);
   }
};

struct pair_hash {
   static uint64_t splitmix64(uint64_t x) {
      x += 0x9e3779b97f4a7c15;
      x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
      x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
      return x ^ (x >> 31);
   }
   size_t operator()(uint64_t x) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(x + FIXED_RANDOM);
   }
   size_t operator()(pair<int, int> p) const {
      static const uint64_t FIXED_RANDOM =
         chrono::steady_clock::now().time_since_epoch().count();
      return splitmix64(p.first * 31 + p.second + FIXED_RANDOM);
   }
};
```

7 O(1) square root

```
// O(1) square root

inline long long isqrt(long long n) {
   double N = n;
   N = sqrtl(N);
   long long sq = N - 2;
   sq = max(sq, 0LL);
   while (sq * sq < n) {
      sq++;
   }
   if ((sq * sq) == n) return sq;
   return sq - 1;
}
```

8 Mint

```
// modular int library

template <int32_t MOD = 998'244'353>
struct Modular {
   int32_t value;
   static const int32_t MOD_value = MOD;

   Modular(long long v = 0) {
      value = v % MOD;
      if (value < 0) value += MOD;
   }
   Modular(long long a, long long b) : value(0) {
      *this += a;
      *this /= b;
   }

   Modular& operator+=(Modular const& b) {
      value += b.value;
      if (value >= MOD) value -= MOD;
      return *this;
   }
   Modular& operator-=(Modular const& b) {
      value -= b.value;
      if (value < 0) value += MOD;
      return *this;
   }
   Modular& operator*=(Modular const& b) {
      value = (long long)value * b.value % MOD;
      return *this;
   }

   friend Modular mexp(Modular a, long long e) {
      Modular res = 1;
      while (e) {
         if (e & 1) res *= a;
         a *= a;
         e >>= 1;
      }
      return res;
   }
   friend Modular inverse(Modular a) { return mexp(a, MOD - 2); }

   Modular& operator/=(Modular const& b) { return *this *= inverse(b); }
   friend Modular operator+(Modular a, Modular const b) { return a += b; }
   friend Modular operator-(Modular a, Modular const b) { return a -= b; }
   friend Modular operator-(Modular const a) { return 0 - a; }
   friend Modular operator*(Modular a, Modular const b) { return a *= b; }
   friend Modular operator/(Modular a, Modular const b) { return a /= b; }
   friend std::ostream& operator<<(std::ostream& os, Modular const& a) {
      return os << a.value;
   }
```

```
   friend bool operator==(Modular const& a, Modular const& b) {
      return a.value == b.value;
   }
   friend bool operator!=(Modular const& a, Modular const& b) {
      return a.value != b.value;
   }
};

using mint = Modular<mod>;
```

9 NT

```
template <class T, class F = multiplies<T>>
T power(T a, long long n, F op = multiplies<T>(), T e = {1}) {
   assert(n >= 0);
   T res = e;
   while (n) {
      if (n & 1) res = op(res, a);
      if (n >>= 1) a = op(a, a);
   }
   return res;
}

template <unsigned Mod = 998'244'353>
struct Modular {
   using M = Modular;
   unsigned v;
   Modular(long long a = 0) : v((a %= Mod) < 0 ? a + Mod : a) {}
   M operator-() const { return M() -= *this; }
   M& operator+=(M r) {
      if ((v += r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator-=(M r) {
      if ((v += Mod - r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator*=(M r) {
      v = (uint64_t)v * r.v % Mod;
      return *this;
   }
   M& operator/=(M r) { return *this *= power(r, Mod - 2); }
   friend M operator+(M l, M r) { return l += r; }
   friend M operator-(M l, M r) { return l -= r; }
   friend M operator*(M l, M r) { return l *= r; }
   friend M operator/(M l, M r) { return l /= r; }
   friend bool operator==(M l, M r) { return l.v == r.v; }
   friend bool operator!=(M l, M r) { return l.v != r.v; }
   friend ostream& operator<<(ostream& os, M& a) { return os << a.v; }
   friend istream& operator>>(istream& is, M& a) {
      int64_t w;
      is >> w;
      a = M(w);
      return is;
   }
};

const unsigned mod = 1e9 + 7;

using mint = Modular<>;

// smallest prime divisor computation

vector<int32_t> spf(maxa, -1);

void precompute() {
   spf[0] = spf[1] = 1;
   for (int32_t i = 2; i < maxa; i++) {
      if (spf[i] == -1) {
         for (int32_t j = i; j < maxa; j += i) {
            if (spf[j] == -1) spf[j] = i;
         }
      }
   }
}

// linear sieve
template <int32_t SZ>
struct Sieve {
   bitset<SZ> isprime;
   vector<int32_t> primes;
   Sieve(int32_t n = SZ - 1) {
      for (int32_t i = 2; i <= n; ++i) {
         if (!isprime[i]) primes.push_back(i);
         for (auto prime : primes) {
            if (i * prime > n) break;
            isprime[i * prime] = true;
            if (i % prime == 0) break;
         }
      }
   }
};

// segmented sieve using O(sqrt(n)) memory, same complexity, cache optimization
struct Sieve {
   vector<int32_t> pr;
   int32_t total_primes;

   Sieve(int32_t n) {
      const int32_t S = 1 << 15;
      int32_t result = 0;
      vector<char> block(S);
      vector<int32_t> primes;

      int32_t nsqrt = sqrt(n);
      vector<char> is_prime(nsqrt + 1, true);

      for (int32_t i = 2; i <= nsqrt; i++) {
         if (is_prime[i]) {
            primes.push_back(i);
            for (int32_t j = i * i; j <= nsqrt; j += i) is_prime[j] = false;
         }
      }

      for (int32_t k = 0; k * S <= n; k++) {
         fill(block.begin(), block.end(), true);
         int32_t start = k * S;
         for (int32_t p : primes) {
            int32_t start_idx = (start + p - 1) / p;
            int32_t j = max(start_idx, p) * p - start;
            for (; j < S; j += p) block[j] = false;
         }
         if (k == 0) block[0] = block[1] = false;
         for (int32_t i = 0; i < S && start + i <= n; i++) {
            if (block[i]) {
               ++result;
               pr.push_back(start + i);
            }
         }
      }
      total_primes = result;
   }
};

// factorial precomputation

vector<mint> fact(maxn);
void precompute_facts() {
   fact[0] = 1;
   for (int32_t i = 0; i < maxn - 1; i++) {
      fact[i + 1] = fact[i] * (i + 1);
   }
}

mint C(int32_t n, int32_t k) { return fact[n] / (fact[k] * fact[n - k]); }

mint P(int32_t n, int32_t k) { return fact[n] / fact[n - k]; }

// O(1) square root

inline int64_t isqrt(int64_t n) {
   // long double ideally
   double N = n;
   N = sqrtl(N);
   int64_t sq = N - 2;
   sq = max(sq, 0LL);
   while (sq * sq < n) ++sq;
   if (sq * sq == n) return sq;
   return sq - 1;
}

namespace primeCount {
// https://en.wikipedia.org/wiki/Prime-counting_function
const int32_t Maxn = 1e7 + 10;
const int32_t MaxPrimes = 1e6 + 10;
const int32_t PhiN = 1e5;
const int32_t PhiK = 100;

// uint32_t ar[(Maxn >> 6) + 5] = {0};
int32_t len = 0; // num of primes
int32_t primes[MaxPrimes];
int32_t pi[Maxn];
int32_t dp[PhiN][PhiK];
bitset<Maxn> fl;

void sieve(int32_t n) {
   fl[1] = true;
   for (int32_t i = 4; i <= n; i += 2) fl[i] = true;
   for (int32_t i = 3; i * i <= n; i += 2) {
      if (!fl[i]) {
         for (int32_t j = i * i; j <= n; j += i << 1) fl[j] = true;
      }
   }
   for (int32_t i = 1; i <= n; i++) {
      if (!fl[i]) primes[len++] = i;
      pi[i] = len;
   }
}

void init() {
   sieve(Maxn - 1);
   int32_t n, k, res;
   for (n = 0; n < PhiN; ++n) dp[n][0] = n;
   for (k = 1; k < PhiK; ++k) {
      int32_t p = primes[k - 1];
      for (n = 0; n < PhiN; ++n) {
         dp[n][k] = dp[n][k - 1] - dp[n / p][k - 1];
      }
   }
}

// for sum of primes, multiply the subtracted term by primes[k - 1] in both this
// and dp
int64_t non_multiples(int64_t n, int32_t k) {
   if (n < PhiN && k < PhiK) return dp[n][k];
   if (k == 1) return ((++n) >> 1);
   if (primes[k - 1] >= n) return 1;
   return non_multiples(n, k - 1) - non_multiples(n / primes[k - 1], k - 1);
}

int64_t Legendre(int64_t n) {
   if (n < Maxn) return pi[n];
   int32_t lim = sqrt(n) + 1;
```

```
   int32_t k = upper_bound(primes, primes + len, lim) - primes;
   return non_multiples(n, k) + k - 1;
}

// complexity: n^(2/3) (log n^(1/3))
// Lehmer's method to calculate pi(n)
int64_t Lehmer(int64_t n) {
   if (n < Maxn) return pi[n];
   int64_t w, res = 0;
   int32_t i, j, a, b, c, lim;
   b = sqrt(n), c = cbrt(n), a = Lehmer(sqrt(b)), b = Lehmer(b);
   res = non_multiples(n, a) + (((b + a - 2) * (b - a + 1)) >> 1);
   for (i = a; i < b; ++i) {
      w = n / primes[i];
      lim = Lehmer(sqrt(w)), res -= Lehmer(w);
      if (i <= c) {
         for (j = i; j < lim; ++j) {
            res += j;
            res -= Lehmer(w / primes[j]);
         }
      }
   }
   return res;
}
} // namespace primeCount

namespace fastPrimeCount {
inline int64_t isqrt(int64_t n) {
   // long double ideally
   double N = n;
   N = sqrtl(N);
   int64_t sq = N - 2;
   sq = max(sq, (int64_t)0);
   while (sq * sq < n) ++sq;
   if (sq * sq == n) return sq;
   return sq - 1;
}
int64_t prime_pi(const int64_t N) {
   if (N <= 1) return 0;
   if (N == 2) return 1;
   const int32_t v = isqrt(N);
   int32_t s = (v + 1) / 2;
   vector<int32_t> smalls(s);
   for (int32_t i = 1; i < s; ++i) smalls[i] = i;
   vector<int32_t> roughs(s);
   for (int32_t i = 0; i < s; ++i) roughs[i] = 2 * i + 1;
   vector<int64_t> larges(s);
   for (int32_t i = 0; i < s; ++i) larges[i] = (N / (2 * i + 1) - 1) / 2;
   vector<bool> skip(v + 1);
   const auto divide = [](int64_t n, int64_t d) -> int32_t {
      return (double)(n) / d;
   };
   const auto half = [](int32_t n) -> int32_t { return (n - 1) >> 1; };
   int32_t pc = 0;
   for (int32_t p = 3; p <= v; p += 2)
      if (!skip[p]) {
         int32_t q = p * p;
         if (int64_t(q) * q > N) break;
         skip[p] = true;
         for (int32_t i = q; i <= v; i += 2 * p) skip[i] = true;
         int32_t ns = 0;
         for (int32_t k = 0; k < s; ++k) {
            int32_t i = roughs[k];
            if (skip[i]) continue;
            int64_t d = int64_t(i) * p;
            larges[ns] = larges[k] -
                     (d <= v ? larges[smalls[d >> 1] - pc]
                        : smalls[half(divide(N, d))]) +
                     pc;
            roughs[ns++] = i;
         }
         s = ns;
         for (int32_t i = half(v), j = ((v / p) - 1) | 1; j >= p; j -= 2) {
            int32_t c = smalls[j >> 1] - pc;
            for (int32_t e = (j * p) >> 1; i >= e; --i) smalls[i] -= c;
         }
         ++pc;
      }
   larges[0] += int64_t(s + 2 * (pc - 1)) * (s - 1) / 2;
   for (int32_t k = 1; k < s; ++k) larges[0] -= larges[k];
   for (int32_t l = 1; l < s; ++l) {
      int32_t q = roughs[l];
      int64_t M = N / q;
      int32_t e = smalls[half(M / q)] - pc;
      if (e < l + 1) break;
      int64_t t = 0;
      for (int32_t k = l + 1; k <= e; ++k)
         t += smalls[half(divide(M, roughs[k]))];
      larges[0] += t - int64_t(e - l) * (pc + l - 1);
   }
   return larges[0] + 1;
}
} // namespace fastPrimeCount
```

10 NTT

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

template <class T, class F = multiplies<T>>
T power(T a, long long n, F op = multiplies<T>(), T e = {1}) {
   assert(n >= 0);
   T res = e;
   while (n) {
      if (n & 1) res = op(res, a);
      if (n >>= 1) a = op(a, a);
   }
   return res;
}

template <unsigned Mod = 998'244'353>
struct Modular {
   using M = Modular;
   unsigned v;
   Modular(long long a = 0) : v((a %= Mod) < 0 ? a + Mod : a) {}
   M operator-() const { return M() -= *this; }
   M& operator+=(M r) {
      if ((v += r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator-=(M r) {
      if ((v += Mod - r.v) >= Mod) v -= Mod;
      return *this;
   }
   M& operator*=(M r) {
      v = (uint64_t)v * r.v % Mod;
      return *this;
   }
   M& operator/=(M r) { return *this *= power(r, Mod - 2); }
   friend M operator+(M l, M r) { return l += r; }
   friend M operator-(M l, M r) { return l -= r; }
   friend M operator*(M l, M r) { return l *= r; }
   friend M operator/(M l, M r) { return l /= r; }
   friend bool operator==(M l, M r) { return l.v == r.v; }
   friend bool operator!=(M l, M r) { return l.v != r.v; }
   friend ostream& operator<<(ostream& os, M a) { return os << a.v; }
   friend istream& operator>>(istream& is, M& a) {
      int64_t w;
      is >> w;
      a = M(w);
      return is;
   }
};

using mint = Modular<998244353>;

namespace ntt {

template <unsigned Mod>
void ntt(vector<Modular<Mod>>& a, bool inverse) {
   static vector<Modular<Mod>> dw(30), idw(30);
   if (dw[0] == 0) {
      Modular<Mod> root = 2;
      while (power(root, (Mod - 1) / 2) == 1) root += 1;
      for (int32_t i = 0; i < 30; ++i)
         dw[i] = -power(root, (Mod - 1) >> (i + 2)), idw[i] = 1 / dw[i];
   }
   int32_t n = a.size();
   assert((n & (n - 1)) == 0);
   if (not inverse) {
      for (int32_t m = n; m >>= 1;) {
         Modular<Mod> w = 1;
         int32_t m2 = m << 1;
         for (int32_t s = 0, k = 0; s < n; s += m2) {
            for (int32_t i = s, j = s + m; i < s + m; ++i, ++j) {
               auto x = a[i], y = a[j] * w;
               if (x.v >= Mod) x.v -= Mod;
               a[i].v = x.v + y.v, a[j].v = x.v + (Mod - y.v);
               // here a[i] is not normalised
            }
            w *= dw[__builtin_ctz(++k)];
         }
      }
   } else {
      for (int32_t m = 1; m < n; m <<= 1) {
         Modular<Mod> w = 1;
         int32_t m2 = m << 1;
         for (int32_t s = 0, k = 0; s < n; s += m2) {
            for (int32_t i = s, j = s + m; i < s + m; ++i, ++j) {
               auto x = a[i], y = a[j];
               a[i] = x + y, a[j].v = x.v + (Mod - y.v), a[j] *= w;
            }
            w *= idw[__builtin_ctz(++k)];
         }
      }
      auto inv = 1 / Modular<Mod>(n);
      for (auto&& e : a) e *= inv;
   }
}
template <unsigned Mod>
vector<Modular<Mod>> operator*(vector<Modular<Mod>> l, vector<Modular<Mod>> r) {
   if (l.empty() or r.empty()) return {};
   int32_t n = l.size(), m = r.size(), sz = 1 << __lg(((n + m - 1) << 1) - 1);
   if (min(n, m) < 30) {
      vector<long long> res(n + m - 1);
      for (int32_t i = 0; i < n; ++i)
         for (int32_t j = 0; j < m; ++j) res[i + j] += (l[i] * r[j]).v;
      return {begin(res), end(res)};
   }
   bool eq = l == r;
   l.resize(sz), ntt(l, false);
   if (eq)
      r = l;
   else
      r.resize(sz), ntt(r, false);
   for (int32_t i = 0; i < sz; ++i) l[i] *= r[i];
   ntt(l, true), l.resize(n + m - 1);
   return l;
}

// for 1e9+7 ntt
```

```
constexpr long long mod = 1e9 + 7;
using Mint197 = Modular<mod>;

vector<Mint197> operator*(const vector<Mint197>& l, const vector<Mint197>& r) {
   if (l.empty() or r.empty()) return {};
   int n = l.size(), m = r.size();
   static constexpr int mod0 = 998244353, mod1 = 1300234241, mod2 = 1484783617;
   using Mint0 = Modular<mod0>;
   using Mint1 = Modular<mod1>;
   using Mint2 = Modular<mod2>;
   vector<Mint0> l0(n), r0(m);
   vector<Mint1> l1(n), r1(m);
   vector<Mint2> l2(n), r2(m);
   for (int i = 0; i < n; ++i) l0[i] = l[i].v, l1[i] = l[i].v, l2[i] = l[i].v;
   for (int j = 0; j < m; ++j) r0[j] = r[j].v, r1[j] = r[j].v, r2[j] = r[j].v;
   l0 = l0 * r0, l1 = l1 * r1, l2 = l2 * r2;
   vector<Mint197> res(n + m - 1);
   static const Mint1 im0 = 1 / Mint1(mod0);
   static const Mint2 im1 = 1 / Mint2(mod1), im0m1 = im1 / mod0;
   static const Mint197 m0 = mod0, m0m1 = m0 * mod1;
   for (int i = 0; i < n + m - 1; ++i) {
      int y0 = l0[i].v;
      int y1 = (im0 * (l1[i] - y0)).v;
      int y2 = (im0m1 * (l2[i] - y0) - im1 * y1).v;
      res[i] = y0 + m0 * y1 + m0m1 * y2;
   }
   return res;
}

} // namespace ntt

using namespace ntt;

namespace IO {
const int BUFFER_SIZE = 1 << 15;
char input_buffer[BUFFER_SIZE];
int input_pos = 0, input_len = 0;
char output_buffer[BUFFER_SIZE];
int output_pos = 0;
char number_buffer[100];
uint8_t lookup[100];
void _update_input_buffer() {
   input_len = fread(input_buffer, sizeof(char), BUFFER_SIZE, stdin);
   input_pos = 0;
   if (input_len == 0) input_buffer[0] = EOF;
}
inline char next_char(bool advance = true) {
   if (input_pos >= input_len) _update_input_buffer();

   return input_buffer[advance ? input_pos++ : input_pos];
}

template <typename T>
inline void read_int(T& number) {
   bool negative = false;
   number = 0;

   while (!isdigit(next_char(false)))
      if (next_char() == '-') negative = true;

   do {
      number = 10 * number + (next_char() - '0');
   } while (isdigit(next_char(false)));

   if (negative) number = -number;
}

template <typename T, typename... Args>
inline void read_int(T& number, Args&... args) {
   read_int(number);
   read_int(args...);
}

void _flush_output() {
   fwrite(output_buffer, sizeof(char), output_pos, stdout);
   output_pos = 0;
}

inline void write_char(char c) {
   if (output_pos == BUFFER_SIZE) _flush_output();

   output_buffer[output_pos++] = c;
}

template <typename T>
inline void write_int(T number, char after = '\0') {
   if (number < 0) {
      write_char('-');
      number = -number;
   }
   int length = 0;
   while (number >= 10) {
      uint8_t lookup_value = lookup[number % 100];
      number /= 100;
      number_buffer[length++] = (lookup_value & 15) + '0';
      number_buffer[length++] = (lookup_value >> 4) + '0';
   }
   if (number != 0 || length == 0) write_char(number + '0');
   for (int i = length - 1; i >= 0; i--) write_char(number_buffer[i]);
   if (after) write_char(after);
}
void IOinit() {
   // Make sure _flush_output() is called at the end of the program.
   bool exit_success = atexit(_flush_output) == 0;
   assert(exit_success);
   for (int i = 0; i < 100; i++) lookup[i] = (i / 10 << 4) + i % 10;
}
} // namespace IO

using namespace IO;

int32_t main() {
   IOinit();
   int n, m;
   read_int(n, m);
   vector<Mint197> a(n), b(m);
   for (auto& x : a) read_int(x);
   for (auto& x : b) read_int(x);
   a = a * b;
   for (int32_t i = 0; i < n + m - 1; ++i) {
      write_int(a[i].v, ' ');
   }
}
```

11 FFT

```
namespace fft {

class cmplx {
  public:
   double a, b;
   cmplx() { a = 0.0, b = 0.0; }
   cmplx(double na, double nb = 0.0) { a = na, b = nb; }
   const cmplx operator+(const cmplx& c) { return cmplx(a + c.a, b + c.b); }
   const cmplx operator-(const cmplx& c) { return cmplx(a - c.a, b - c.b); }
   const cmplx operator*(const cmplx& c) {
      return cmplx(a * c.a - b * c.b, a * c.b + b * c.a);
   }
   double magnitude() { return sqrt(a * a + b * b); }
   void print() { cout << "(" << a << ", " << b << ")\n"; }
};

const double PI = acos(-1);

class fft {
  public:
   vector<cmplx> data, roots;
   vector<int32_t> rev;
   int32_t n, s;

   void setSize(int32_t ns) {
      s = ns;
      n = (1 << s);
      int32_t i, j;
      rev = vector<int32_t>(n);
      data = vector<cmplx>(n);
      roots = vector<cmplx>(n + 1);
      for (i = 0; i < n; ++i) {
         for (j = 0; j < s; ++j) {
            if (i & (1 << j)) {
               rev[i] += (1 << (s - j - 1));
            }
         }
      }
      roots[0] = cmplx(1);
      cmplx mult = cmplx(cos(2 * PI / n), sin(2 * PI / n));
      for (i = 1; i <= n; ++i) {
         roots[i] = roots[i - 1] * mult;
      }
   }

   void bitReverse(vector<cmplx>& arr) {
      vector<cmplx> temp(n);
      int32_t i;
      for (i = 0; i < n; ++i) temp[i] = arr[rev[i]];
      for (i = 0; i < n; ++i) arr[i] = temp[i];
   }

   void transform(bool inverse = false) {
      bitReverse(data);
      int32_t i, j, k;
      for (i = 1; i <= s; ++i) {
         int32_t m = (1 << i), md2 = m >> 1;
         int32_t start = 0, increment = (1 << (s - i));
         if (inverse) {
            start = n;
            increment *= -1;
         }
         cmplx t, u;
         for (k = 0; k < n; k += m) {
            int32_t index = start;
            for (j = k; j < md2 + k; ++j) {
               t = roots[index] * data[j + md2];
               index += increment;
               data[j + md2] = data[j] - t;
               data[j] = data[j] + t;
            }
         }
      }
      if (inverse) {
         for (int32_t i = 0; i < n; ++i) {
            data[i].a /= n;
            data[i].b /= n;
         }
      }
   }

   static vector<int32_t> convolution(vector<int32_t>& a, vector<int32_t>& b) {
      int32_t alen = a.size();
      int32_t blen = b.size();
      int32_t resn = alen + blen - 1;
      int32_t s = 0, i;
```

```
    while ((1 << s) < resn) ++s;
    int32_t n = 1 << s;

    fft pga, pgb;
    pga.setSize(s);
    for (i = 0; i < alen; ++i) pga.data[i] = cmplx(a[i]);
    for (i = alen; i < n; ++i) pga.data[i] = cmplx(0);
    pga.transform();

    pgb.setSize(s);
    for (i = 0; i < blen; ++i) pgb.data[i] = cmplx(b[i]);
    for (i = blen; i < n; ++i) pgb.data[i] = cmplx(0);
    pgb.transform();

    for (i = 0; i < n; ++i) pga.data[i] = pga.data[i] * pgb.data[i];
    pga.transform(true);
    vector<int32_t> result(resn);
    for (i = 0; i < resn; ++i) result[i] = (int32_t)(pga.data[i].a + 0.5);

    int32_t actualSize = resn - 1;
    while (~actualSize && result[actualSize] == 0) --actualSize;
    if (actualSize < 0) actualSize = 0;
    result.resize(actualSize + 1);
    return result;
  }
};
} // namespace fft
```

12 Template

```
#pragma GCC optimize("Ofast")
#pragma GCC target("avx")
#pragma GCC optimize("unroll-loops")

#include <bits/stdc++.h>

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/rope>

using namespace __gnu_pbds;
using namespace __gnu_cxx;
using namespace std;

#define fastio \
  ios_base::sync_with_stdio(0); \
  cin.tie(0); \
  cout.tie(0);

#define bitcount __builtin_popcountll
// for trailing 1s, do trailing0(n + 1)
#define leading0 __builtin_clzll
#define trailing0 __builtin_ctzll
#define isodd(n) (n & 1)
#define iseven(n) (!(n & 1))

#define del_rep(v) \
  sort(v.begin(), v.end()); \
  v.erase(unique(v.begin(), v.end()), v.end());
#define checkbit(n, b) ((n >> b) & 1)

// order_of_key(k) - number of elements e such that func(e, k) returns true,
// where func is less or less_equal find_by_order(k) - kth element in the set
// counting from 0

typedef tree<int, null_type, less<int>, rb_tree_tag,
         tree_order_statistics_node_update>
  ordered_set;
typedef tree<int, null_type, less_equal<int>, rb_tree_tag,
         tree_order_statistics_node_update>
  ordered_multiset;

const int INF = 1e9;
const long long LINF = INF * INF;
const double EPS = 1e-9;
const double PI = acosl(-1);
const int mod = 1e9 + 7;
const int maxn = 5e5 + 5;
const int maxa = 1e6 + 5;
const int logmax = 25;

void solve(int case_no) {}

signed main() {
  fastio;
  cout << setprecision(10) << fixed;
  int t = 1;
  for (int _t = 1; _t <= t; _t++) {
    solve(_t);
  }
  _flush_output();
  return 0;
}
```