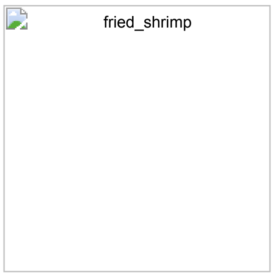
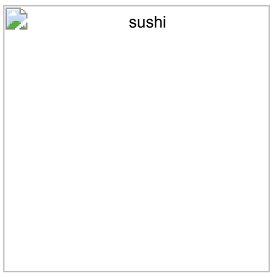
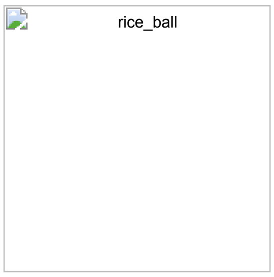
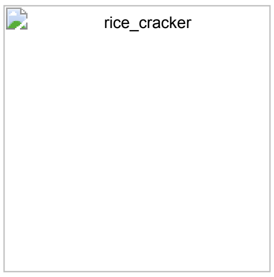
ACM-ICPC Cheat Sheet

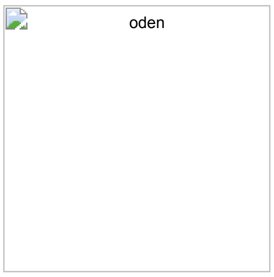
```
Orange Juice 情報
```











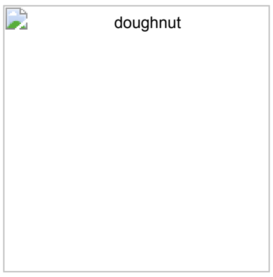


Table of Contents *generated with [DocToc (https://github.com/thlorenz/doctoc)](https://github.com/thlorenz/doctoc)*

1. Basic

1.1 C++ Solution Template

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

#define DEBUG false
#define OJ_DEBUG

#define $(x) {if (DEBUG) {cout << __LINE__ << ": "; {x} cout << endl;}}
#define _(x) {cout << #x << " = " << x << " ";}

const double E = 1e-8;
const double PI = acos(-1);

using namespace std;

int main() {
    ios::sync_with_stdio(false);

}
```

1.1.1 Optional include list

> *Use it when there is no* `bits/stdc++.h`

```
#include <iostream>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <climits>
#include <stack>
#include <queue>
#include <vector>
#include <set>
#include <map>
#include <list>
#include <cassert>
#include <unordered_map>
```

1.2 Strings

1.2.1 C++ String

read one line

getline()

```
string a;
getline(cin, a);
cout << a << endl;
```

Input

```
Hello World!!!
```

Output

```
Hello World!!!
```

Convert to char array

```
string cppstr = "this is a string";
char target[1024];
strcpy(target, cppstr.c_str());
```

1.2.2 C String (Character Array)

Input C String

gets()

> *Reads characters from the standard input (stdin) and stores them as a C string into str until a newline character or the end-of-file is reached.*

```
char s[12];
gets(s);
cout << "\"" << s << "\"" << ", length: " << strlen(s) << endl;
```

Input

```
hello world
new line
```

Output

```
"hello world", length: 11
```

Convert to C++ string

```
char arrstr[] = "this is a string";
string target = string(arr);
```

1.3 STL Algorithm

Include the algorithm library if you do not use the solution template.

```
#include <algorithm>
```

1.3.1 Permutation

Usage

```
bool next_permutation (BidirectionalIterator first, BidirectionalIterator last);
bool next_permutation (BidirectionalIterator first, BidirectionalIterator last, Compare comp);
```

Example

```
// next_permutation example
#include <iostream>     // std::cout
#include <algorithm>    // std::next_permutation, std::sort

int main () {
  int myints[] = {1,2,3};

  std::sort (myints,myints+3);

  std::cout << "The 3! possible permutations with 3 elements:\n";
  do {
    std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
  } while ( std::next_permutation(myints,myints+3) );

  std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';

  return 0;
}
```

Output

```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
After loop: 1 2 3
```

1.3.2 Binary Search

Usage

```
bool binary_search (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);
// return true if found, false if not
```

1.3.3 Lower Bound

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than val.

Usage

```
ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);
```

1.3.4 Swap

Usage

```
void swap (T& a, T& b);
void iter_swap (ForwardIterator1 a, ForwardIterator2 b);
```

`iter_swap` example

```
int myints[]={10,20,30,40,50 };              //   myints:  10  20  30  40  50
std::vector<int> myvector (4,99);            // myvector:  99  99  99  99

std::iter_swap(myints + 3,myvector.begin() + 2); //   myints:  99  20  30 [99] 50
                                             // myvector:  10  99 [40] 99
```

1.3.5 Heap

- make_heap: Rearranges the elements in the range [first,last) in such a way that they form a heap. The element with the highest value is always pointed by first.
- pop_heap: Rearranges the elements in the heap range [first,last) in such a way that the part considered a heap is shortened by one: The element with the highest value is moved to (last-1).
- push_heap: Given a heap in the range [first,last-1), this function extends the range considered a heap to [first,last) by placing the value in (last-1) into its corresponding location within it.
- sort_heap: Sorts the elements in the heap range [first,last) into ascending order.

Usage

```
void make_heap (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
void pop_heap (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
void push_heap (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
void sort_heap (RandomAccessIterator first, RandomAccessIterator last); Compare comp);
```

1.3.6 Sort

> *Sorts the elements in the range [first,last) into ascending order. `stable_sort` preserves the relative order of the elements with equivalent values.*

Usage

```
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
void stable_sort ( RandomAccessIterator first, RandomAccessIterator last, Compare comp );
```

1.3.7 Compare

Using lambda expression

```
auto cmp = [](const T& a, const T& b) { return true; };
set<T, decltype(cmp)> a_set_with_customized_comparator(cmp);
```

Compare function

> *Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. It should returns true if the first element is considered to be "smaller" than the second one.*

Using by `sort`, `make_heap` and etc.

```
bool myfunction (int i,int j) { return (i<j); }
```

Define operator <()

Member function

> *recommended // can use for priority_queue, sort,*

```
struct Edge {
   int from, to, weight;
        bool operator<(Edge that) const {
        return weight > that.weight;
    }
};
```

verbal version

```
struct Edge {
   int from, to, weight;
        bool operator<(const Edge& that) const {
        return this->weight > that.weight;
    }
};
```

Non-member function

```
struct Edge {
    int from, to, weight;
    friend bool operator<(Edge a, Edge b) {
        return a.weight > b.weight;
    }
};
```

Define operator()()

> *You can use comparison function for STL containers by passing them as the first argument of the constructor, and specifying the function type as the additional template argument. For example:*

```
set<int, bool (*)(int, int)> s(cmp);
```

> *A functor, or a function object, is an object that can behave like a function. This is done by defining operator()() of the class. In this case, implement operator()() as a comparison function:*

```
vector<int> occurrences;
struct cmp {
    bool operator()(int a, int b) {
        return occurrences[a] < occurrences[b];
    }
};
set<int, cmp> s;
priority_queue<int, vector<int>, cmp> pq;
```

Used by `priority_queue` .

1.4 STL Containers

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

1.4.1 Map

> *Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.*

```
#include <map>
```

Define a Map

```
template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

Commonly used method

```
begin()
end()

empty()
size()

operator[] // if not found, insert one

insert(pair<first type, second type)
erase()
clear()

find() // if not found, return end()
count() // return 1 or 0
```

> *TODO add more interface*

Red-black Tree

C++ map is implemented as a red-black tree.

A red–black tree is a data structure which is a type of self-balancing binary search tree.

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red–black tree:

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
3. All leaves (NIL) are black. (All leaves are same color as the root.)
4. Every red node must have two black child nodes.
5. Every path from a given node to any of its descendant leaves contains the same number of black nodes.

Hash Map (Unordered Map)

> *Unordered map is implemented as a hash table.*

```
#include <unordered_map>
```

> *Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.*

```
template < class Key,                                    // unordered_map::key_type
           class T,                                      // unordered_map::mapped_type
           class Hash = hash<Key>,                       // unordered_map::hasher
           class Pred = equal_to<Key>,                   // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

Commonly used method

```
// most are similar to map
```

// TODO add more interface

1.4.2 Pair

1.4.3 Vector

Constructor

```
std::vector<int> second (4,100);  // four ints with value 100
```

Methods

- begin(), end()
- front(), back()
- clear()
- size()
- push_back(const value_type& val)
- pop_back()

1.4.4 List

> *List containers are implemented as doubly-linked lists.*

Methods

- begin(), end()
- front(), back()
- clear()
- push_front(const value_type& val)
- push_back(const value_type& val)
- pop_front(): remove the first element.
- pop_back(): remove the last element.
- remove(const value_type& val): remove all elements of value val.
- insert(iterator position, const value_type& val)
- size()
- reverse()
- sort(), sort (Compare comp)
-
- resize()
- reserve()

1.4.5 Queue

```
#include <queue>
```

Constructor

```
queue<int> my_queue;
queue<int, list<int> > my_queue (my_list);
// use list<int> as container, copy my_list into my_queue
```

Methods

```
void queue::push(const value_type& val);
void queue::pop();
bool queue::empty() const;
size_type queue::size() const;
const_reference& queue::front() const;
```

1.4.6 Double-ended Queue

```
#include <dequeue>
```

1.4.7 Stack

```
#include <stack>
```

Constructor

```
stack<int, vector<int> > my_stack (my_data);
// use vector<int> as container, copy my_data into my_stack

bool stack::empty() const;
size_type stack::size() const;
const_reference& stack::top() const;
void stack::push (const value_type& val);
void stack::pop();
```

1.4.8 Priority Queue

```
#include <queue>
```

```
// constructor
priority_queue<int> my_priority_queue;
priority_queue<int, vector<int>, greater<int> > two_priority_queue; // if use greater<int>, must have vector<int>
priority_queue<My_type, vector<My_type>, Comparator_class> my_priority_queue (my_data.begin(), my_data.end()); // use Comparator_clas

bool priority_queue::empty() const; // return true if empty, false if not
size_type priority_queue::size() const; // return size of queue
const_reference priority_queue::top() const; // returns a constant reference to the top element
void priority_queue::push(const value_type& val); // inserts a new element, initialize to val
void priority_queue::pop(); // removes the element on top
```

```
struct My_type {
    int weight;
    int other;
};

struct My_class_for_compare {
    public:
        bool operator() (My_type a, My_type b) {
            return a.weight < b.weight;
        }
};

vector<My_type> my_vector = {(My_type){2, 789}, (My_type){1, 127}, (My_type){3, 456}};

priority_queue<My_type, vector<My_type>, My_class_for_compare> one_priority_queue (my_vector.begin(), my_vector.end());
one_priority_queue.push((My_type){4, 483});
while (one_priority_queue.size() != 0) {
    My_type temp = one_priority_queue.top();
    one_priority_queue.pop();
    SHOW_B(temp.weight, temp.other);
}

vector<int> my_int = {2, 3, 1};

priority_queue<int, vector<int>, greater<int> > two_priority_queue (my_int.begin(), my_int.end());
while (two_priority_queue.size() != 0) {
    SHOW_A(two_priority_queue.top());
    two_priority_queue.pop();
}

priority_queue<int> three_priority_queue (my_int.begin(), my_int.end());
while (three_priority_queue.size() != 0) {
    SHOW_A(three_priority_queue.top());
    three_priority_queue.pop();
}


// output
// temp.weight = 4, temp.other = 483
// temp.weight = 3, temp.other = 456
// temp.weight = 2, temp.other = 789
// temp.weight = 1, temp.other = 127
// two_priority_queue.top() = 1
// two_priority_queue.top() = 2
// two_priority_queue.top() = 3
// three_priority_queue.top() = 3
// three_priority_queue.top() = 2
// three_priority_queue.top() = 1
```

2. Advanced Data Structures

2.1 Heap

2.2 Tree

2.2.0 Tree Traversal

2.2.1 Pointer Jumping

Initialize: O(Nlog(N))

Query: O(Nlog(N))

```
#define MAX_NODE 100030
#define MAX_NODE_LOG 20

#define TREE_ROOT 0
vector<int> g[MAX_NODE];
vector<int> parent_jump[MAX_NODE];
vector<int> path;

void init_jump(int cur = TREE_ROOT) {
        int d = 1;
        while (true) {
                int index = path.size() - d;
                if (index < 0)
                        break;
                parent_jump[cur].push_back(path[index]);
                d <<= 1;
        }
        path.push_back(cur);

    for (int i = 0; i < g[cur].size(); i++) {
        int nx = g[cur][i];
        if (cur == TREE_ROOT || nx != parent_jump[cur][0]) {
            init_jump(nx);
        }
    }
    path.pop_back();
}

int go_up(int cur, int dis) {
        int mask = 1;
        int index = 0;
        while (mask <= dis) {
                if (dis & mask)
                        cur = parent_jump[cur][index];
                mask <<= 1;
                index++;
        }
        return cur;
}
```

2.2.2 Heavy-Light Decomposition

Build: O(N)

Overhead: O(log(N))

```
//
// CodeForces 593D
//
// sol 1 - path must < 64 - not straightforward
// sol 2 - Heavy-Light Decomposition + Segment Tree + Math - not straightforward
//
// floor( floor(A / B) / C) = floor(A / (B * C))
//

#include <stdio.h>
#include <sstream>
#include <iomanip>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <climits>
#include <vector>
#include <stack>
#include <queue>
#include <set>
// #include <unordered_set>
#include <map>
// #include <unordered_map>
#include <cassert>

#define SHOW(...) {;}
#define REACH_HERE {;}
#define LOG(s, ...) {;}
#define LOGLN(s, ...) {;}

// #undef HHHDEBUG
#ifdef HHHDEBUG
#include "template.h"
#endif

using namespace std;

int len(unsigned long long x) {
        if (x == 0)
                return 1;

        int ret = 0;
        while (x > 0) {
                ret++;
                x /= 10;
        }
        return ret;
}

unsigned long long will_boom(unsigned long long a, unsigned long long b) {
    if (a == ULLONG_MAX || b == ULLONG_MAX)
        return ULLONG_MAX;

        int la = len(a);
        int lb = len(b);
        if (la - 1 + lb - 1 + 1 > 20)
                return ULLONG_MAX;
        return a * b;
}

struct SegmentTree {
    struct Node {
        int l;
        int r;
        unsigned long long accu;
```

```
};

int r_most;
vector<Node> node;
void init(int rm) {
    r_most = rm;

    int tree_range = r_most + 1;
    if (tree_range <= 0)
        return ;

    int tree_size = 1;
    while (tree_size <= tree_range)
        tree_size <<= 1;
    if (__builtin_popcount(tree_range) != 1) // count number of '1' bits
        tree_size <<= 1;

    node.resize(tree_size);

    Node& root = node[1];
    root.l = 0;
    root.r = r_most;
    root.accu = 1;
    for (int i = 2; i < node.size(); i++) {
        Node& cur = node[i];
        cur.accu = 1;

        const Node& par = node[i / 2];
        if (par.l == par.r)
            continue;
        int m = (par.l + par.r) / 2;
        if (i & 1) {
            cur.l = m + 1;
            cur.r = par.r;
        }
        else {
            cur.l = par.l;
            cur.r = m;
        }
    }

    // SHOW("init")
    // show();
}

void show() {
    SHOW("SegmentTree", r_most)
    for (int i = 1; i < node.size(); i++) {
        Node& cur = node[i];
        SHOW(i, cur.l, cur.r, cur.accu)
    }
}

void update(int l, int r, unsigned long long new_y, int i = 1) {
    Node& cur = node[i];

    if (cur.l == cur.r) {
            cur.accu = new_y;
            return ;
    }

    int m = (cur.l + cur.r) / 2;
    int il = i * 2;
    int ir = il + 1;

    if (l <= m)
```

```
                update(l, r, new_y, il);
        else
                update(l, r, new_y, ir);

        cur.accu = will_boom(node[il].accu, node[ir].accu);
    }

    unsigned long long query(int l, int r, int i = 1) {
        if (l > r)
            return 1;

        Node& cur = node[i];
        if (l <= cur.l && cur.r <= r)
                return cur.accu;

        if (r < cur.l || cur.r < l) {
                REACH_HERE
                return -1;
        }

        int m = (cur.l + cur.r) / 2;
        int il = i * 2;
        int ir = il + 1;

        unsigned long long ans = 1;
        if (l <= m) {
                unsigned long long temp_l = query(l, r, il);
            ans = will_boom(ans, temp_l);
        }

        if (m < r) {
                unsigned long long temp_r = query(l, r, ir);
            ans = will_boom(ans, temp_r);
        }

        return ans;
    }
};

struct Graph {
    map<pair<int, int>, int> node_edge;

    struct Edge {
        int from;
        int to;
        unsigned long long y;
    };

    const static int MAXNODE = 200005;
    vector<int> g[MAXNODE];
    vector<Edge> edge;
    int n;
    void init(int nn) {
        n = nn;
        for (int i = 0; i <= n; i++)
            g[i].clear();
        edge.clear();
    }

    void add_e(int x, int y, unsigned long long val_y) {
        Edge e1 = {x, y, val_y};
        g[x].push_back(edge.size());
        node_edge[make_pair(x, y)] = edge.size();
        edge.push_back(e1);

        Edge e2 = {y, x, val_y};
        g[y].push_back(edge.size());
        node_edge[make_pair(y, x)] = edge.size();
        edge.push_back(e2);

        // LOGLN("add edge %d - %d = %llu", x, y, val_y);
    }

    int parent[MAXNODE];
    int size[MAXNODE];
    int size_parent(int cur) {
        int& s = size[cur] = 1;
        for (int i = 0; i < g[cur].size(); i++) {
            int ie = g[cur][i];
            const Edge& e = edge[ie];
            int nx = e.to;
            if (nx != parent[cur]) {
                parent[nx] = cur;
                s += size_parent(nx);
            }
        }
        return s;
    }
    void show_size_parent() {
        for (int i = 0; i <= n; i++)
            printf("node %d: size %d, parent %d\n", i, size[i], parent[i]);
    }

    struct Chain {
        int head;
        int head_depth;
        int len;
    };
    vector<Chain> chain;
    int chain_total;
    int chain_no[MAXNODE]; // chain_no[node] == chain_index
    int chain_pos[MAXNODE]; // chain_pos[node] == x'th node in the chain // 0 based
    void hld(int cur, int depth) {
        if (chain_total == chain.size()) {
            Chain c = {cur, depth, 0};
            chain.push_back(c);
        }

        chain_no[cur] = chain_total;
        chain_pos[cur] = chain[chain_total].len;
        chain[chain_total].len++; // 0 based, add later

        if (depth != 0 && g[cur].size() - 1 == 0)
            return ;

        int heavy = edge[g[cur][0]].to;
        for (int i = 0; i < g[cur].size(); i++) {
            int ie = g[cur][i];
            const Edge& e = edge[ie];
            int nx = e.to;
            if (heavy == parent[cur] || (nx != parent[cur] && size[nx] > size[heavy]))
                heavy = nx;
        }

        hld(heavy, depth + 1);
        for (int i = 0; i < g[cur].size(); i++) {
            int ie = g[cur][i];
            const Edge& e = edge[ie];
            int nx = e.to;
            if (nx != parent[cur] && nx != heavy) {
                chain_total++;
                hld(nx, depth + 1);
            }
        }
    }

    void heavy_light_decomposition() {
        int root = 1;
        parent[root] = -1;
        assert(n == size_parent(root));

        chain_total = 0;
        chain.clear();

        hld(root, 0);
        chain_total++;
    }
    void show_hld() {
        printf("chain_total = %d\n", chain_total);
        for (int i = 0; i <= chain_total; i++)
            printf("chain %d: len %d, head %d, head depth %d\n", i, chain[i].len, chain[i].head, chain[i].head_depth);
        for (int i = 0; i <= n; i++)
            printf("node %d: chain %d, pos %d\n", i, chain_no[i], chain_pos[i]);
    }

    vector<SegmentTree> st;
    void init_sol() {
        st.resize(chain_total);
        for (int i = 0; i < st.size(); i++)
            st[i].init(chain[i].len - 2);

        for (int i = 0; i < edge.size(); i += 2) {
                const Edge& e = edge[i];
                int cn1 = chain_no[e.from];
                int cn2 = chain_no[e.to];

                if (cn1 == cn2) {
                        int cp1 = chain_pos[e.from];
                        int cp2 = chain_pos[e.to];
                        if (cp1 > cp2)
                                swap(cp1, cp2);
                        st[cn1].update(cp1, cp2 - 1, e.y);
                }
        }

        // for (int i = 0; i < chain_total; i++)
        //      st[i].show();
    }

    void update(int p, unsigned long long c) {
        p *= 2;
        Edge& e = edge[p];
        e.y = edge[p + 1].y = c;

        int cn1 = chain_no[e.from];
        int cn2 = chain_no[e.to];
        if (cn1 == cn2) {
            int cp1 = chain_pos[e.from];
            int cp2 = chain_pos[e.to];
            if (cp1 > cp2)
                swap(cp1, cp2);
            st[cn1].update(cp1, cp2 - 1, c);
        }
    }

    unsigned long long query(int a, int b, unsigned long long y) {
        unsigned long long accu = 1;
        while (true) {
                int cn1 = chain_no[a];
                int cn2 = chain_no[b];

            if (chain[cn1].head_depth < chain[cn2].head_depth)
                swap(a, b), swap(cn1, cn2);

                if (cn1 == cn2) {
                        int cp1 = chain_pos[a];
                        int cp2 = chain_pos[b];
                        if (cp1 > cp2)
                                swap(cp1, cp2);
                        unsigned long long temp = st[cn1].query(cp1, cp2 - 1);
                        accu = will_boom(accu, temp);
                        break;
                }


                if (a != chain[cn1].head) {
                        int ha = chain[cn1].head;
                int cp1 = chain_pos[ha];
                int cp2 = chain_pos[a];
                if (cp1 > cp2)
                    swap(cp1, cp2);
                        unsigned long long temp = st[cn1].query(cp1, cp2 - 1);
                        accu = will_boom(accu, temp);
                        a = ha;
                }

            int pa = parent[a];
            unsigned long long temp = edge[node_edge[make_pair(a, pa)]].y;
            accu = will_boom(temp, accu);
            a = parent[a];
        }

        if (accu == 0) // no idea why would be 0
            return 0;
        return y / accu;
    }
};

int n;
int m;
Graph g;

int main() {
        scanf("%d %d", &n, &m);
    g.init(n);
    for (int i = 1; i < n; i++) {
        int x, y;
        unsigned long long val_y;
        scanf("%d %d %llu", &x, &y, &val_y);
        g.add_e(x, y, val_y);
    }

    g.heavy_light_decomposition();
    // g.show_hld();
    g.init_sol();

    for (int i = 0; i < m; i++) {
        int op;
        scanf("%d", &op);
        if (op == 1) {
                int a, b;
                unsigned long long y;
            scanf("%d %d %llu", &a, &b, &y);
            unsigned long long ans = g.query(a, b, y);
            printf("%llu\n", ans);
        }
        else {
                int p;
                unsigned long long c;
            scanf("%d %llu", &p, &c);
            g.update(p - 1, c);
        }
    }
}
```

2.2.3 Lowest Common Ancestor

Reduction from LCA to RMQ

let n = number of nodes in the tree

preprocess: euler tour O(n) + RMQ init O(nlog(n))

query: RMQ O(1)

tutorial (https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/)

```
// lowest common ancestor LCA
// --> range minimun range_minimum_query
// preprocess: O(3 * nlog(n))
// range_minimum_query: O(1)
//
// 1471. Tree
// http://acm.timus.ru/problem.aspx?space=1&num=1471
// Time limit: 2.0 second
// Memory limit: 64 MB
//
// A weighted tree is given. You must find the distance between two given nodes.
// Input
// The first line contains the number of nodes of the tree n (1 ≤ n ≤ 50000).
// The nodes are numbered from 0 to n − 1.
// Each of the next n − 1 lines contains three integers u, v, w,
// which correspond to an edge with weight w (0 ≤ w ≤ 1000) connecting nodes u and v.
// The next line contains the number of queries m (1 ≤ m ≤ 75000).
// In each of the next m lines there are two integers.
// Output
// For each range_minimum_query, output the distance between the nodes with the given numbers.
//
// Sample
//
// input
// 3
// 1 0 1
// 2 0 1
// 3
// 0 1
// 0 2
// 1 2
//
// output
// 1
// 1
// 2

#include <vector>
#include <stdio.h>
#include <string.h>
#include <cmath>

using namespace std;

struct Graph {
    struct Edge {
        int to;
        int len;
    };

    const static int MAXNODE = 1 * 1e5 + 2;

    vector<int> g[MAXNODE];
    vector<Edge> edge;
    int n;

    int root = 0;

    void init(int nn, int m=0) {
        n = nn;
        for (int i = 0; i <= n; i++)
            g[i].clear();
        edge.clear();
        m *= 2;
        edge.reserve(m); // may speedup // add_e too slow
```

```
}

void add_e(int x, int y, int len) {
    g[x].push_back(edge.size());
    edge.push_back((Edge){y, len});

    g[y].push_back(edge.size());
    edge.push_back((Edge){x, len});
}

void show() {
    for (int i = 0; i <= n; i++) {
        printf("%d:", i);
        for (int ie : g[i])
            printf(" %d", edge[ie].to);
        printf("\n");
    }
    printf("\n");
}

//
// --- start of LCA ---
//
vector<int> dis_to_root;
vector<int> first_visit_time; // max possible number of visits to all nodes == 2 * number of nodes - 1
vector<int> visit;
int visit_counter;
vector<vector<int>> rmq;

int range_minimum_query(int l, int r) { // query [l, r]
    if (l > r)
        swap(l, r);

    int interval_len = r - l; // (r - l + 1) - 1

    int first_half = 1;
    while ((1 << first_half) <= interval_len)
        first_half++;
    first_half--;

    int second_half = r - (1 << first_half) + 1;
    if (first_visit_time[rmq[l][first_half]] < first_visit_time[rmq[second_half][first_half]])
        return rmq[l][first_half];
    return rmq[second_half][first_half];
}

int get_lca(int x, int y) {
    return range_minimum_query(first_visit_time[x], first_visit_time[y]);
}

int dist(int x, int y) {
    int lca = get_lca(x, y);
    return dis_to_root[x] + dis_to_root[y] - 2 * dis_to_root[lca];
}

void euler_tour(int cur) {
    visit[++visit_counter] = cur; // v_t[node] = time // needed in case don't have two child
    if (first_visit_time[cur] == 0) // if first time
        first_visit_time[cur] = visit_counter; // record time f_v_t[node] = time
    for (int ie : g[cur]) {
        const Edge& e = edge[ie];
        int nx = e.to;
        int len = e.len;
        if (first_visit_time[nx] == 0) {
            dis_to_root[nx] = dis_to_root[cur] + len;
            euler_tour(nx);
```

```
                visit[++visit_counter] = cur; // every two child_visit_time have one parent_visit_time inserted between
            }
        }
    }

    void build_lca() { // O(Nlog(N))
        int one_n = n + 1;
        int two_n = 2 * one_n;
        vector<int>(one_n, 0).swap(dis_to_root);
        vector<int>(one_n, 0).swap(first_visit_time);
        vector<int>(two_n, 0).swap(visit);

        int LOG_MAXLENGTH = log2(two_n) + 2;
        vector<vector<int>>(two_n, vector<int>(LOG_MAXLENGTH)).swap(rmq);

        visit_counter = 0;
        euler_tour(root);

        for (int i = 0; i < visit_counter; i++)
            rmq[i][0] = visit[i];

        for (int j = 1; j < LOG_MAXLENGTH; j++)
            for (int i = 0; i < visit_counter; i++) {
                if (i + (1 << j) > visit_counter)
                    break;
                rmq[i][j] = rmq[i][j - 1];
                if (first_visit_time[rmq[i][j - 1]] > first_visit_time[rmq[i + (1 << (j - 1))][j - 1]])
                    rmq[i][j] = rmq[i + (1 << (j - 1))][j-1];
            }
    }
    //
    // --- end of LCA ---
    //
};

int n, m;
Graph g;

int main(int argc, char const *argv[]) {
    scanf("%d", &n);
    g.init(n, n);
    for (int i = 1; i < n; i++) {
        int x, y, d;
        scanf("%d %d %d", &x, &y, &d);
        g.add_e(x, y, d);
    }

    g.build_lca();

    scanf("%d", &m);
    for (int i = 0; i < m; i++) {
        int x, y;
        scanf("%d %d", &x, &y);
        printf("%d\n", g.dist(x, y));
    }
}
```

Another implementation, only used by ZXZ.

```
const int MAX_N = 1e5 + 10;
const int MAX_LOG_N = 21;

struct node {
    int baba;
    int k, v;
    vector<int> children;
};

node tree[MAX_N];  // tree[0] is not used.


int range_minimun_query(int left, int right) {
    // calculate log(right)
    if (left > right) swap(left, right);
    int log_right = 1;
    while ((1 << log_right) <= right - left) log_right ++;
    log_right --;
    // return the minimum from the lower level RMQ.
    bool is_lower = (first_visit[rmq[left][log_right]] <
                     first_visit[rmq[right - (1<<log_right) + 1][log_right]]);
    return is_lower ? rmq[left][log_right] : rmq[right - (1<<log_right) + 1][log_right];
}

void dfs_rmq(int cur) {
    visit_count ++;
    visit[visit_count] = cur;
    if (!first_visit[cur]) first_visit[cur] = visit_count;
    for (int i = 0; i < tree[cur].children.size(); i++) {
        int child = tree[cur].children[i];
        if (first_visit[child]) continue;
        level[child] = level[cur] + 1;
        dfs_rmq(child);
        visit_count ++;
        visit[visit_count] = cur;
    }
}

void init_rmq() {
    dfs_rmq(1);
    for (int i = 1; i <= visit_count; i++) rmq[i][0] = visit[i];
    for (int log_level = 1; log_level < MAX_LOG_N; log_level++) {
        for (int i = 1; i <= visit_count; i++) {
            if (i + (1<<log_level) > visit_count) continue;
            if (first_visit[rmq[i][log_level - 1]] <
                first_visit[rmq[i + (1<<(log_level-1))][log_level-1]]) {
                rmq[i][log_level] = rmq[i][log_level - 1];
            } else {
                rmq[i][log_level] = rmq[i + (1<<(log_level-1))][log_level-1];
            }
        }
    }
}
```

2.2.3.1 Tarjan's Off-line Algorithm

let n = number of ndoes of the tree, m = number of query

O(n + m)

```
function TarjanOLCA(u)
     MakeSet(u);
     u.ancestor := u;
     for each v in u.children do
         TarjanOLCA(v);
         Union(u,v);
         Find(u).ancestor := u;
     u.colour := black;
     for each v such that {u,v} in P do
         if v.colour == black
             print "Tarjan's Lowest Common Ancestor of " + u +
                   " and " + v + " is " + Find(v).ancestor + ".";
```

TODO refactor add comments

```
//
// 1471. Tree
// http://acm.timus.ru/problem.aspx?space=1&num=1471
// Time limit: 2.0 second
// Memory limit: 64 MB
//
// A weighted tree is given. You must find the distance between two given nodes.
//
// Input
// The first line contains the number of nodes of the tree n (1 ≤ n ≤ 50000).
// The nodes are numbered from 0 to n – 1.
// Each of the next n – 1 lines contains three integers u, v, w,
// which correspond to an edge with weight w (0 ≤ w ≤ 1000) connecting nodes u and v.
// The next line contains the number of queries m (1 ≤ m ≤ 75000).
// In each of the next m lines there are two integers.
//
// Output
// For each range_minimum_query, output the distance between the nodes with the given numbers.
//
// Sample
//
// input
// 3
// 1 0 1
// 2 0 1
// 3
// 0 1
// 0 2
// 1 2
//
// output
// 1
// 1
// 2

#include <iostream>
#include <vector>
#include <string.h>

using namespace std;

#define MAXHHH 50003
#define MAXJJJ 75005

struct Node {
    vector<int> next; // edge list
    vector<int> dist; // edge length
    vector<int> query; // that node of a query
    vector<int> lca; // lca of this and that node
    vector<int> q_i; // index of query in offline query array
};

Node g[MAXHHH];
int n, m;

int father[MAXHHH];
int find(int x) { // find-union set
    if (father[x] == x)
        return x;
    return father[x] = find(father[x]);
}
void mergeFirstInToSecond(int x, int y) { // find-union set
    father[find(x)] = find(y);
}
```

```
pair<int, int> q[MAXJJJ]; // query: node a, b
pair<int, int> q_ans[MAXJJJ]; // record answer's location, first = node index, second = answer index
int came[MAXHHH];
void tarjan_lca_dfs(int cur) {
    // process cur node and all its sub-tree
    // process all query related to this node and nodes in sub-tree
    came[cur] = 1;
    for (unsigned int i = 0; i < g[cur].next.size(); i++) {
        int next = g[cur].next[i];
        if (came[next] == 1) // don't go back, it is dfs
            continue;

        tarjan_lca_dfs(next); // process sub-tree
        mergeFirstInToSecond(cur, next); // order matters
    }

    for (unsigned int i = 0; i < g[cur].query.size(); i++) {
        int that = g[cur].query[i];
        if (came[that] == 0)
            continue;

        // lca must be father[that] because this comes from father[that]
        // and father[that] haven't merge with father[father[that]]
        g[cur].lca[i] = find(that);
        q_ans[g[cur].q_i[i]] = make_pair(cur, i); // record position for later usage
    }
}

int root_dis[MAXHHH];
void dfs(int cur) {
    for (unsigned int i = 0; i < g[cur].next.size(); i++) {
        int next = g[cur].next[i];
        if (root_dis[next] != -1)
            continue;

        root_dis[next] = g[cur].dist[i] + root_dis[cur];
        dfs(next);
    }
}

int main(int argc, char const *argv[]) {
    cin >> n;
    for (int i = 1; i < n; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        g[a].next.push_back(b);
        g[a].dist.push_back(c);
        g[b].next.push_back(a);
        g[b].dist.push_back(c);
        father[i] = i;
    }
    cin >> m;
    for (int i = 0; i < m; i++) { // offline
        cin >> q[i].first >> q[i].second;
        g[q[i].first].query.push_back(q[i].second);
        g[q[i].first].lca.push_back(-1);
        g[q[i].first].q_i.push_back(i);
        g[q[i].second].query.push_back(q[i].first);
        g[q[i].second].lca.push_back(-1);
        g[q[i].second].q_i.push_back(i);
        q_ans[i] = make_pair(-1, -1);
    }

    memset(root_dis, -1, sizeof(root_dis)); root_dis[0] = 0;
    dfs(0);
    tarjan_lca_dfs(0);
    for (int i = 0; i < m; i++) {
        int lca = g[q_ans[i].first].lca[q_ans[i].second];
        int ans = root_dis[q[i].first] + root_dis[q[i].second] - 2 * root_dis[lca];
        cout << ans << endl;
    }
}
```

2.2.4 Centroid Decomposition

O(NlogN)

```
//
// https://threads-iiith.quora.com/Centroid-Decomposition-of-a-Tree
//
//
// centroid decomposition O(NlogN)
//
// codeforces 342E
//
// E. Xenia and Tree
// time limit per test5 seconds
// memory limit per test256 megabytes
// inputstandard input
// outputstandard output
// Xenia the programmer has a tree consisting of n nodes.
// We will consider the tree nodes indexed from 1 to n.
// We will also consider the first node to be initially painted red,
// and the other nodes — to be painted blue.
//
// The distance between two tree nodes v and u is the number of edges in the shortest path between v and u.
//
// Xenia needs to learn how to quickly execute queries of two types:
//
// paint a specified blue node in red;
// calculate which red node is the closest to the given one and print the shortest distance to the closest red node.
// Your task is to write a program which will execute the described queries.
//
// Input
// The first line contains two integers n and m (2≤n≤105, 1≤m≤105) — the number of nodes in the tree and the number of queries.
// Next n-1 lines contain the tree edges, the i-th line contains a pair of integers ai, bi (1≤ai, bi≤n, ai≠bi) — an edge of the tree
//
// Next m lines contain queries. Each query is specified as a pair of integers ti, vi (1≤ti≤2, 1≤vi≤n).
// If ti=1, then as a reply to the query we need to paint a blue node vi in red.
// If ti=2, then we should reply to the query by printing the shortest distance from some red node to node vi.
//
// It is guaranteed that the given graph is a tree and that all queries are correct.
//
// Output
// For each second type query print the reply in a single line.
//

struct Graph {
    struct Edge {
        int to;
    };

    const static int MAXNODE = 1 * 1e5 + 2;

    vector<int> g[MAXNODE];
    vector<Edge> edge;
    int n;

    int root = 1;

    void init(int nn, int m=0) {
        n = nn;
        for (int i = 0; i <= n; i++)
            g[i].clear();
        edge.clear();
        m *= 2;
        edge.reserve(m); // may speedup // add_e too slow
    }

    void add_e(int x, int y) {
        g[x].push_back(edge.size());
        edge.push_back((Edge){y});
        g[y].push_back(edge.size());
        edge.push_back((Edge){x});
    }

    void show() {
        for (int i = 0; i <= n; i++) {
            printf("%d:", i);
            for (int ie : g[i])
                printf(" %d", edge[ie].to);
            printf("\n");
        }
        printf("\n");
    }

    //
    // --- start of centroid decomposition ---
    //
        vector<int> centroid; // index of upper level centroid node

        vector<int> subtree_size;
        vector<int> parent;

        vector<bool> deleted;
        int compute_subtree(int cur) {
                int& size = subtree_size[cur];
                for (int ie : g[cur]) {
                        const Edge& e = edge[ie];
                        int nx = e.to;
                        if (parent[nx] == -1) {
                                parent[nx] = cur;
                                size += compute_subtree(nx);
                        }
                }
                return size;
        }

        void centroid_decomposite(int cur, int from, int tree_size) { // O(Nlog(N))
                int half_size = tree_size / 2;

                while (subtree_size[cur] <= half_size) // go up until centroid is in subtree
                        cur = parent[cur];

                while (1) {
                        int candidate = cur;
                        for (int ie : g[cur]) { // go down if centroid is in subtree
                                const Edge& e = edge[ie];
                                int nx = e.to;
                                if (!deleted[nx] && nx != parent[cur]) {
                                        if (subtree_size[nx] > half_size) {
                                                SHOW(cur, nx, subtree_size[nx], candidate)
                                                candidate = nx;
                                        }
                                }
                        }
                        if (candidate == cur)
                                break;
                        cur = candidate;
                }

                deleted[cur] = true;
                centroid[cur] = from;

                int temp = parent[cur];
                int cur_size = subtree_size[cur];
                while (!deleted[temp] && temp != parent[temp]) { // update all the subtree_size of parent
```

```
                        subtree_size[temp] -= cur_size;
                        temp = parent[temp];
                }

                for (int ie : g[cur]) { // decomposite parent and children
                        const Edge& e = edge[ie];
                        int nx = e.to;
                        if (!deleted[nx]) {
                                int nx_tree_size; // size of next level centroid tree
                                if (nx == parent[cur])
                                        nx_tree_size = tree_size - cur_size;
                                else
                                        nx_tree_size = subtree_size[nx];

                                if (nx_tree_size == 1)
                                        centroid[nx] = cur; // don't need to go into it
                                else
                                        centroid_decomposite(nx, cur, nx_tree_size);
                        }
                }
        }

        void centroid_decomposite() {
                vector<int>(n + 1, -1).swap(centroid);
                vector<int>(n + 1, 1).swap(subtree_size); // initialized each to be 1 (itself)
                vector<int>(n + 1, -1).swap(parent);
                vector<bool>(n + 1, false).swap(deleted);

                parent[root] = root;
                compute_subtree(root);

                centroid_decomposite(root, -1, n);
        }
    //
    // --- end of centroid decomposition ---
    //
};
```

Key functions

```
int dfs_size(int cur, int from) {
    int total = 1;
    for (int child : tree[cur].children) {
        if (child == from) continue;
        total += dfs_size(child, cur);
    }
    tree[cur].size = total;
    return total;
}

void find_centroid(int cur, int from) {
    #ifdef DEBUG
        cout << "find centroid for " << cur << " from " << from << endl;
    #endif
    // SHOW(cur)
    // finish condition
    if (tree[cur].size == 1) {
        tree[cur].c_parent = from;
    }
    // check if current node is centroid.
    int max_child_size = 0, max_child;
    for (int child : tree[cur].children) {
        // if (child == from) continue;
        if (tree[child].c_parent != -1) continue;
        if (tree[child].size > max_child_size) {
            max_child_size = tree[child].size;
            max_child = child;
        }
    }
    if (max_child_size > tree[cur].size / 2) {
        // move root
        tree[max_child].size = tree[cur].size;
        tree[cur].size -= max_child_size;
        find_centroid(max_child, from);
    } else {
        // cur is centroid
        tree[cur].c_parent = from;
        if (from == 0) ctree_root = cur;
        for (int child : tree[cur].children) {
            // if (child == from) continue;
            if (tree[child].c_parent != -1) continue;
            find_centroid(child, cur);
        }
    }
}

void color(int cur) {
    int parent = cur;
    while (parent) {
        tree[parent].closest = min(tree[parent].closest, distance(cur, parent));
        parent = tree[parent].c_parent;
    }
}

int query(int cur) {
    int ans = MAX_N;
    int parent = cur;
    while (parent) {
        ans = min(ans, tree[parent].closest + distance(cur, parent));
        parent = tree[parent].c_parent;
    }
    return ans;
}
```

2.3 Trie / Trie Graph / AC Automaton

> *O(NL+M) - NL: total len of words in dict, M: len of article*

```
//
// input: n, q, string x n, string x q
// output: for each query, print number of string whose prefix is the query
//
// Sample Input
// 5
// babaab
// babbbaaaa
// abba
// aaaaabaa
// babaababb
// 5
// babb
// baabaaa
// bab
// bb
// bbabbaab
// Sample Output
// 1
// 0
// 3
// 0
// 0
//
//
//
// check if any word in dict appear in article
//
// Sample Input
// 6
// aaabc
// aaac
// abcc
// ac
// bcd
// cd
// aaaaaaaaaaabaaadaaac
//
// Sample Output
// YES
//

using namespace std;

#define HHH 1000002
struct TrieNode
{
    char val; // 'a' ~ 'z'
    bool ended; // is a word ended here
    int count; // number of word ended here
    int childCount; // number of word contianing this prefix

    int next[26]; // index of child node, 'a' ~ 'z'

    int prev; // parent node // for trie-graph
    bool suffixEnded;// suffix node is an end // for trie-graph
    int suffix[26]; // suffix node // for trie-graph

    TrieNode() {
        val = 0;
        memset(next, -1, sizeof(next));
        ended = false;
        count = 0;
        childCount = 0;
        prev = -1;
        suffixEnded = false;
        memset(suffix, -1, sizeof(suffix));
    }

    void show() {
        cerr <<
        "Val: " << val <<
        ", prev = " << prev <<
        ", ended = " << ended <<
        ", count = " << count <<
        ", childCount = " << childCount
        << "\n\t ";
        for (int i = 0; i < 4; i++)
            cerr << (char)('a' + i) << ":" << next[i] << " ";
        cerr
        << "\n\tsuffix suffixEnded = " << suffixEnded << "\n"
        << "\t ";
        for (int i = 0; i < 4; i++)
            cerr << (char)('a' + i) << ":" << suffix[i] << " ";
        cerr << endl;
    }
};

struct Trie
{
    TrieNode node[HHH];
    int size; // the index of last node
    int add(string& s) { // return index of new node
        int preIndex = 0;
        for (int i = 0; i < s.length(); i++) {
            TrieNode& pre = node[preIndex];
            int& curIndex = pre.next[s[i] - 'a'];
            if (curIndex == -1) {
                size++;
                curIndex = size;
            }
            TrieNode& cur = node[curIndex];
            cur.val = s[i];
            cur.childCount++;

            preIndex = curIndex;
        }
        node[preIndex].ended = true;
        node[preIndex].count++;
        node[0].childCount++;
        return preIndex;
    };

    void buildSuffix() {
        queue<int> q;
        for (int i = 0; i < 26; i++) {
            int& next = node[0].next[i];
            int& suffix = node[0].suffix[i];

            suffix = 0;
            if (next == -1)
                next = 0;
            else {
                q.push(next);
                node[next].prev = 0;
            }
        }

        while (q.size()) {
            int cur = q.front(); q.pop();
            int prev = node[cur].prev;
            int prevSuffix = node[prev].suffix[node[cur].val - 'a'];
            if (node[prevSuffix].ended)
                node[cur].suffixEnded = true;

            for (int i = 0; i < 26; i++) {
                int& next = node[cur].next[i];
                int& suffix = node[cur].suffix[i];

                suffix = node[prevSuffix].next[i];
                if (next == -1)
                    next = suffix;
                else {
                    q.push(next);
                    node[next].prev = cur;
                }
            }
        }
    }

    int get(string& s) { // get index of node
        int preIndex = 0;
        for (int i = 0; i < s.length() && preIndex != -1; i++)
            preIndex = node[preIndex].next[s[i] - 'a'];
        return preIndex;
    };

    bool match(string& s) {
        for (int i = 0, cur = 0; i < s.length(); i++) {
            cur = node[cur].next[s[i] - 'a'];
            if (node[cur].ended || node[cur].suffixEnded)
                return true;
        }
        return false;
    }

    Trie() {
        size = 0;
    };

    void show() {
        for (int i = 0; i <= size; i++) {
            show_A(i);
            node[i].show();
        }
    }
};

int n, q;
Trie t;

int main() {
    cin >> n;

    string temp;
    for (int i = 0; i < n; i++) {
        cin >> temp;
        t.add(temp);
    }

    cin >> q;
    for (int i = 0; i < q; i++) {
        cin >> temp;
        int index = t.get(temp);

        if (index == -1)
```

```
            cout << 0 << endl; // not found
        else
            cout << t.node[index].childCount << endl; // found
    }


    // check if any word in dict appear in article
    t.buildSuffix();

    string article; cin >> article;
    if (t.match(article))
        cout << "YES" << endl;
    else
        cout << "NO" << endl;
}
```

```
// http://blog.csdn.net/u010700335/article/details/38930175

const int maxn = 26;//26个小写字母或者大写字母，再加上0~9就是72
//定义字典树结构体
typedef struct Trie
{
    bool flag;//从根到此是否为一个单词
    Trie *next[maxn];//有多少个分支
}Trie;
// 声明一个根，不含任何信息
Trie *root;
//初始化该根
void trie_init()
{
    int i;
    root = new Trie;
    root->flag = false;
    for(i=0;i<maxn;i++)
        root->next[i] = NULL;
}
// 插入一个字符串
void trie_insert(char *word)
{
    //int i = 0;
    //while(word[i] != '\0')
    Trie *tem = root;
    int i;
    while(*word != '\0')
    {
       // cout << "root**" << tem->next[0];
        if(tem->next[*word-'a'] == NULL)// 为空才建立
        {
            Trie *cur = new Trie;
            cur->flag = false;
            for(i=0;i<maxn;i++)
                cur->next[i] = NULL;
            tem->next[*word-'a'] = cur;
        }
        tem = tem->next[*word-'a'];
        //cout << *word << "**";
        word++;
    }
    tem->flag = true;//插入一个完整的单词
}
// 查找一个字符串
bool trie_search(char *word)
{
    Trie *tem = root;
    int i;
    for(i=0; word[i]!='\0'; i++)
    {
        if(tem==NULL || tem->next[word[i]-'a']==NULL)
            return false;
        tem = tem->next[word[i]-'a'];
    }
    return tem->flag;
}


void trie_del(Trie *cur)
{
    int i;
    for(i=0;i<maxn;i++)
    {
        if(cur->next[i] != NULL)
```

```
            trie_del(cur->next[i]);
    }
    delete cur;
}


int main()
{
    int i,n;
    char tmp[50];
    trie_init();
    cout << "请输入初始化字典树的字符串(字符0结束)：" << endl;
    while(cin >> tmp)
    {
        //cout << tmp << endl;
        if(tmp[0] == '0' && tmp[1] =='\0') break;
        trie_insert(tmp);
    }
    cout << "请输入要查找的字符串：" << endl;
    while(cin >> tmp)
    {
        //cout << tmp << endl;
        if(tmp[0] == '0' && tmp[1] =='\0') break;
        if(trie_search(tmp))
            cout << "查找成功！再次输入查找，字符0结束查找：" << endl;
        else
            cout << "查找失败！再次输入查找，字符0结束查找：" << endl;
    }
    return 0;
}
```

2.4 Suffix Tree

2.5 Suffix Array

```
int main() {
        cin >> s;
        build_suffix_array();
        compute_lcp();
        longest_repeated_substring();

        longest_common_substring("GATAGACA", "CATA");
}
```

2.5.1 Build Suffix Array

> *O(nlog(n))*
>
> *reference: Competitve Programming*

```
#include <iostream>
#include <stdio.h>
#include <cstring>

using namespace std;

#define HH 100002

const char base_char = '.';
string s;
int rank_array[HH];
int rank_array_temp[HH];
int suffix_array[HH];
int suffix_array_temp[HH];
int counter[HH];

void counting_sort(int k) {
        memset(counter, 0, sizeof(counter));

        int len = s.length();
        for (int i = 0; i < len; i++) {
                // i will cover all suffix_array[i]
                // so i + k will cover all suffix_array[i] + k
                // so just use i + k instead of suffix_array[i] for counting
                // the order is not important anyway
                int old_rank = i + k < len ? rank_array[i + k] : 0;
                counter[old_rank]++;
        }

        int accu = 0;
        int largest_possible_value = max(256, len); // initial rank is based on ascii value
        for (int i = 0; i < largest_possible_value; i++) {
                // counter[x]: 2, 1, 0, 0, 2, 0
                // become    : 0, 2, 3, 3, 3, 5
                // which stands for the new rank with that old rank x
                // the meaning changes here !!
                int count_temp = counter[i];
                counter[i] = accu;
                accu += count_temp;
        }

        for (int i = 0; i < len; i++) {
                // for each suffix_array[i]
                // get its new rank using its old rank suffix_array[i] + k
                // above if i + k >= n, we change it to 0
                // the same here
                int old_rank = suffix_array[i] + k < len ? rank_array[suffix_array[i] + k] : 0;
                // value of counter[old_rank] is the new rank
                // put suffix_array[i] to its new position == new rank
                // why ++ ? suppose
                // counter[x]: 0, 2, 3, 3, 3, 5
                // gradually it become
                // counter[x]: 1, 2, 3, 3, 3, 5
                // counter[x]: 2, 2, 3, 3, 3, 5
                // counter[x]: 2, 3, 3, 3, 3, 5
                // ... thus assign each suffix_array[x] distinguished value
                // even if there keys are the same
                // but relation between different key keeps
                // why assign distinguished rank ?
                // of course... otherwise multiple suffix_array[i] (different suffix) go to same suffix_array[x]
                // the rank of them are still kept in the rank_array[x]
                // later will compress the rank
                // keeping the order between those with same key
                // eventually all the rank will be different
                suffix_array_temp[counter[old_rank]++] = suffix_array[i];
        }
        memcpy(suffix_array, suffix_array_temp, sizeof(suffix_array));
}
void build_suffix_array() {
        int len = s.length();
        for (int i = 0; i < len; i++)
                rank_array[i] = s[i] - base_char; // initial rank // based on 1st char
        for (int i = 0; i < len; i++)
                suffix_array[i] = i; // initialize

        for (int k = 1; k < len; k <<= 1) {
                // sort based on 2^i portion
                // finally all will be sorted

                // [0, i + k) is first part
                // [i + k, i + k + k) is second part
                // sort second part then first part
                // leading to a stable_sort (?)
                // use [i + k, i + k + k) as key first
                // then use [0, i + k) as key
                counting_sort(k);
                counting_sort(0);

                // after spread suffix with same rank into differnt suffix_array[x] slot (consecutive)
                // compress the rank_array
                // so that suffix with same second part [i + k, i + k + k) (which is the rank...)
                // have same rank...
                int rank = 0;
                rank_array_temp[suffix_array[0]] = rank;
                for (int i = 1; i < len; i++) {
                        if (rank_array[suffix_array[i - 1]] != rank_array[suffix_array[i]]
                                ||
                                rank_array[suffix_array[i] + k] != rank_array[suffix_array[i - 1] + k])
                                rank++;
                        rank_array_temp[suffix_array[i]] = rank;
                }
                memcpy(rank_array, rank_array_temp, sizeof(rank_array));
        }

        for (int i = 0; i < len; i++)
                cout << "i: " << i << ", suffix " << suffix_array[i] << " : " << s.substr(suffix_array[i], s.length() - suffix_array

        cout << endl;
}
```

2.5.2 Pattern Matching

O(mlog(n))

```
void find_pattern(const string& pattern) {
        // binary search upper bound & lower bound (?) in the suffix array
        // to get a matched range
        // let m = pattern.length()
        // let n = s.length()
        // time complexity: O(mlog(n))
}
```

2.5.3 Longest Common Prefix

> *length of common prefix between suffix_array[i-1] and suffix_array[i]*
>
> *let n = s.length()*
>
> *O(n)*

```
//
// ...
//

int longest_common_prefix[HH]; // lcp[i] = length of common prefix between sa[i-1] and sa[i]
int phi[HH]; // phi[sa[i]] = sa[i-1] // naming ? // useless when built
int permuted_lcp[HH]; // useless when built temp for lcp

void compute_lcp() {
	// theorem: number of increase/decrese on cur_lcp is O(len)
	// so time complexity: O(len) for computing lcp
	int len = s.length();
	phi[suffix_array[0]] = -1;
	for (int i = 1; i < len; i++)
		phi[suffix_array[i]] = suffix_array[i - 1];
	for (int i = 0, cur_lcp = 0; i < len; i++) {
		if (phi[i] == 0)
			permuted_lcp[i] = 0;
		else {
			while (s[i + cur_lcp] == s[phi[i] + cur_lcp])
				cur_lcp++;
			permuted_lcp[i] = cur_lcp;
			cur_lcp = max(cur_lcp - 1, 0);
		}
	}
	for (int i = 0; i < len; i++)
		longest_common_prefix[i] = permuted_lcp[suffix_array[i]];

	for (int i = 0; i < len; i++)
		cout << "i: " << i << ", suffix " << suffix_array[i] << " (lcp: " << longest_common_prefix[i] << ") : " << s.substr(
	cout << endl;
}
```

2.5.4 Longest Repeated Substring

> *let n = s.length()*
>
> *O(n)*

```
void longest_repeated_substring() {
	int len = s.length();
	int max_repeated = -1;
	int i_max = -1;
	for (int i = 0; i < len; i++) {
		if (longest_common_prefix[i] > max_repeated) {
			max_repeated = longest_common_prefix[i];
			i_max = i;
		}
	}
	cout << "longest repeated substring: " << s.substr(suffix_array[i_max], max_repeated) << endl;
}
```

2.5.5 Longest Common Substring

string a, b (also applies to multiple strings)

construct Suffix Array of a.b & find Longest Common Prefix

let n = max(a.length(), b.length())

O(nlog(n) + n)

```
void longest_common_substring(const string& a, const string& b) {
        s = a + base_char + b;
        cout << "concatenated string: " << s << endl;

        build_suffix_array();
        compute_lcp();

        int len = s.length();
        int max_common = -1;
        int i_max = -1;
        for (int i = 1; i < len; i++) {
                int cur_lcp = longest_common_prefix[i];
                if (cur_lcp > max_common) {
                        if ((suffix_array[i] < a.length()) ^ (suffix_array[i - 1] < a.length())) {
                                max_common = cur_lcp;
                                i_max = i;
                        }
                }
        }
        cout << "longest common prefix: " << s.substr(suffix_array[i_max], max_common) << endl;
}
```

2.6 Binary Indexed Tree

Binary Indexed Tree

O(logN) to query and update SUM(a[1]~a[i])

```
#define MAX_INDEX nnn

int tree[MAX_INDEX + 1]; // 1 <= I <= MAX_INDEX

int low_bit(int i) {
    return i & -i;
}

int query(int i) {
    int ans = 0;
    for (; i > 0; i -= low_bit(i))
        ans += tree[i];
    return ans;
}

void insert(int i, int value) {
    for (; i <= MAX_INDEX; i += low_bit(i))
        tree[i] += value;
}
```

修改区间+查询点，

【1】修改操作：将A[l..r]之间的全部元素值加上c；

【2】求和操作：求此时A[x]的值。

这个模型中需要设置一个辅助数组B：B[i]表示A[1..i]到目前为止共被整体加了多少

```
#define INTERVAL_LIMIT 100005

int tree_add_i_n[INTERVAL_LIMIT];

int low_bit(int i) {
    return i & -i;
}

int query(int i, int* tree, int UP_LIMIT) {
    int ans = 0;
    for (; i > 0; i -= low_bit(i))
        ans += tree[i];
    return ans;
}

void insert(int i, int value, int* tree, int UP_LIMIT) {
    for (; i <= UP_LIMIT; i += low_bit(i))
        tree[i] += value;
}

int main() {
    memset(tree_add_in, 0, sizeof(tree_add_in));
    insert(3, 1, tree_add_in, INTERVAL_LIMIT);
    insert(5, -1, tree_add_in, INTERVAL_LIMIT);
    insert(4, 2, tree_add_in, INTERVAL_LIMIT);
    insert(6, -2, tree_add_in, INTERVAL_LIMIT);

    for (int i = 1; i <= 7; i++)
        SHOW_B(i, query(i, tree_add_in, INTERVAL_LIMIT));
}
```

修改区间+查询区间

b[i]: add b[i] to a[i], a[i+1], ..., a[n]

so

sigma(i): a[1] + a[2] + ... + a[i]

sigma(i) = ib[1] + (i-1)b[2] + ... + 2b[i-1] + b[i]

sigma(i) = (i+1){b[1] + b[2] + ... + b[i]} - {b[1] + 2b[2] + ... + ib[i]}

so use one more tree c[i]

c[i]: 1b[1] + 2b[2] + ... + ib[i]

2.7 Segment Tree

2.7.0 Range Update + Range Query

with lazy propagation

build O(N)

query O(log(N))

update O(log(N))

```
//
// CodeForces 243D      Cubes
//
// dynamic programming + segment tree + math - O(N*N*log(N)) - not straightforward
//
// struct SegmentTree is slow, use with caution
//
//

#include <stdio.h>
#include <sstream>
#include <iomanip>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <climits>
#include <vector>
#include <stack>
#include <queue>
#include <set>
// #include <unordered_set>
#include <map>
// #include <unordered_map>
#include <cassert>

#define SHOW(...) {;}
#define REACH_HERE {;}
#define PRINT(s, ...) {;}
#define PRINTLN(s, ...) {;}

// #undef HHHDEBUG
#ifdef HHHDEBUG
#include "template.h"
#endif

using namespace std;

struct SegmentTree {
        struct Op { // store lazy operation
                int h;
        };
    struct Node {
        int l; // [l, ]
        int r; // [, r]

        int h; // value

        bool lazy;
        Op op;
    };

    vector<Node> node;
    void init(int l, int r) { // [l, r]
        int tree_range = r - l + 1;
        if (tree_range <= 0)
            return ;

        int tree_size = 1;
        while (tree_size <= tree_range)
            tree_size <<= 1;
        if (__builtin_popcount(tree_range) != 1) // count number of '1' bits
            tree_size <<= 1;

        node.resize(tree_size); // (tree_range, tree_size): (001001, 100000) (001000, 010000)
        Node& root = node[1];
        root.l = l, root.r = r;
        root.h = root.op.h = 0;
        for (int i = 2; i < node.size(); i++) {
            Node& cur = node[i];
            cur.h = 0;

            const Node& par = node[i / 2]; // parent node
            if (par.l == par.r) // if parent is end node, skip
                cur.l = cur.r = -1;
            else {
                int m = (par.l + par.r) / 2;
                if (i % 2)
                    cur.l = m + 1, cur.r = par.r;
                else
                    cur.l = par.l, cur.r = m;
            }
        }
    }

    void show() {
        SHOW("SegmentTree NAME")
        for (int i = 1; i < node.size(); i++) {
            Node& cur = node[i];
            if (cur.l == -1 && cur.r == -1)
                continue;
            PRINTLN("(%2d) [%2d,%2d] val: %d", i, cur.l, cur.r, cur.internal)
        }
    }

    int query(int xl, int xr, int i = 1) { // query [xl, xr]
        Node& cur = node[i];
        if (cur.l == cur.r) // if end node
            return cur.h;

        if (xl <= cur.l && cur.r <= xr) // if query cover the node
            return cur.h;

        int lci = i * 2;
        const Node& lc = node[lci];
        int rci = lci + 1;
        const Node& rc = node[rci];
        if (cur.lazy) { // if have lazy operation, push down
            update(lc.l, lc.r, cur.op.h, lci);
            update(rc.l, rc.r, cur.op.h, rci);
            cur.lazy = false;
        }

        int ret = INT_MAX;
        if (xl <= lc.r) { // if query cover left child
            int temp = query(xl, xr, lci);
            if (ret > temp)
                ret = temp;
        }
        if (rc.l <= xr) { // if query cover right child
            int temp = query(xl, xr, rci);
            if (ret > temp)
                ret = temp;
        }
        return ret;
    }

    void update(int xl, int xr, int xh, int i = 1) { // update [xl, xr] value xh
        Node& cur = node[i];
        if (cur.l == cur.r) { // if end node
            if (cur.h < xh)
```

```
                cur.h = xh;
            return ;
        }

        if (xl <= cur.l && cur.r <= xr) { // if query cover the node
            if (cur.h < xh) { // update node value
                cur.h = xh;
            }
            if (cur.lazy) { // update the lazy operation // slow if push down now
                if (cur.op.h < xh)
                    cur.op.h = xh;
            }
            else { // store lazy operation
                cur.op.h = xh;
                cur.lazy = true;
            }
            return ;
        }

        int lci = i * 2;
        const Node& lc = node[lci];
        int rci = lci + 1;
        const Node& rc = node[rci];
        if (cur.lazy) { // if have lazy operation, push down
            update(lc.l, lc.r, cur.op.h, lci);
            update(rc.l, rc.r, cur.op.h, rci);
            cur.lazy = false;
        }

        if (xl <= lc.r) // if update cover left node
            update(xl, xr, xh, lci);
        if (rc.l <= xr) // if update cover right node
            update(xl, xr, xh, rci);

        cur.h = min(lc.h, rc.h); // reduce two children
    }
};

struct Hall {
        int h;
        int proj_index[2];
};

const int HH = 1002;

int n;
int vx;
int vy;
Hall hall[HH][HH];

long long ans;

int init_project() {
        int xx[] = {0, 1};
        int yy[] = {1, 0};
        double EPS = 1e-7;

        struct Proj {
                int i;
                int j;
                double x;
                int k;
        };
        vector<Proj> projection;
        projection.reserve(n * n * 2);
        auto get_x = [](double x, double y) -> double {
                return vx == 0 ? x : x - y / vy * vx;
        };

        for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        for (int k = 0; k < 2; k++)
                                projection.push_back((Proj){i, j, get_x(i + xx[k], j + yy[k]), k});

        sort(begin(projection), end(projection), [](const Proj& a, const Proj& b) {
                return a.x < b.x;
        });

        int n_seg = 0;
        hall[projection[0].i][projection[0].j].proj_index[projection[0].k] = 0;
        for (int i = 1; i < projection.size(); i++) {
                if (abs(projection[i].x - projection[i - 1].x) > EPS)
                        n_seg++;
                hall[projection[i].i][projection[i].j].proj_index[projection[i].k] = n_seg;
        }
        return n_seg;
}

void rotate() {
        // if vy < 0
        // flip left right
        if (vy < 0) {
            for (int i = 0; i < n; i++) {
                int l = 0;
                int r = n - 1;
                while (l < r) {
                        swap(hall[i][l].h, hall[i][r].h);
                        l++;
                        r--;
                }
            }
            vy = -vy;
        }

        // if vx <= 0
        // flip diagonal
        if (vx < 0 || vy == 0) {
                for (int i = 0; i < n; i++)
                        for (int j = 0; j < i; j++)
                                swap(hall[i][j].h, hall[j][i].h);
                swap(vx, vy);
        }

        // if vy < 0
        // flip left right
        if (vy < 0) {
            for (int i = 0; i < n; i++) {
                int l = 0;
                int r = n - 1;
                while (l < r) {
                        swap(hall[i][l].h, hall[i][r].h);
                        l++;
                        r--;
                }
            }
            vy = -vy;
        }
}

SegmentTree st;
```

```
int main() {
        scanf("%d %d %d", &n, &vx, &vy);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
                scanf("%d", &hall[i][j].h);

    rotate();
    int len = init_project();

        st.init(0, len - 1);

        for (int j = 0; j < n; j++) {
            for (int i = 0; i < n; i++) {
                const Hall& h = hall[i][j];
                int min_height = st.query(h.proj_index[0], h.proj_index[1] - 1);
                ans += max(0, h.h - min_height);
                st.update(h.proj_index[0], h.proj_index[1] - 1, h.h);
        }
    }
    printf("%lld\n", ans);
}
```

2.7.1 Color

```
const int MAX = 100000;

struct node {
    int left, right;
    int color;
    bool cover;
};

node nodes[3*MAX];

void build_tree(int left, int right, int u) {
    nodes[u].left = left;
    nodes[u].right = right;
    nodes[u].color = 1;
    nodes[u].cover = true;
    if (left == right) return;
    int mid = (left + right)/2;
    build_tree(left, mid, 2*u);
    build_tree(mid+1, right, 2*u + 1);
}

void get_down(int u) {
    int value = nodes[u].color;
    nodes[u].cover = false;
    nodes[2*u].color = value;
    nodes[2*u].cover = true;
    nodes[2*u + 1].color = value;
    nodes[2*u + 1].cover = true;
}

void update(int left, int right, int value, int u) {
    if (left <= nodes[u].left && nodes[u].right <= right) {
        nodes[u].color = value;
        nodes[u].cover = true;
        return;
    }
    if (nodes[u].color == value) return;  // optimize purpose
    //SHOW(u);
    if (nodes[u].cover) get_down(u);
    if (left <= nodes[2*u].right) {
        update(left, right, value, 2*u);
    }
    if (right >= nodes[2*u+1].left) {
        update(left, right, value, 2*u + 1);
    }
    nodes[u].color = nodes[2*u].color | nodes[2*u+1].color;
}

void query(int left, int right, int &sum, int u) {
    if (nodes[u].cover) {
        sum |= nodes[u].color;
        return;
    }
    if (left <= nodes[u].left && nodes[u].right <= right) {
        sum |= nodes[u].color;
        return;
    }
    if (left <= nodes[2*u].right) {
        query(left, right, sum, 2*u);
    }
    if (right >= nodes[2*u+1].left) {
        query(left, right, sum, 2*u + 1);
    }
}
```

```
// Usage
// build_tree(1, L, 1);
// update(a, b, new_color, 1);
// query(a, b, sum_as_reference, 1);

// only for this question
int bit_count(int sum) {
    int ans = 0;
    while (sum) {
        if (sum%2) ans++;
        sum = sum >> 1;
    }
    return ans;
}

int main() {
    int L, T, O;
    cin >> L >> T >> O;
    build_tree(1, L, 1);
    while (O--) {
        char op;
        int a, b, c;
        cin >> op;
        if (op == 'C') {
            cin >> a >> b >> c;
            if (a > b) swap(a, b);
            update(a, b, 1<<(c-1), 1);
        } else {
            cin >> a >> b;
            if (a > b) swap(a, b);
            int sum = 0;
            query(a, b, sum, 1);
            cout << bit_count(sum) << endl;
        }
    }
    return 0;
}
```

2.7.2 Range Sum + Range Replace

```
const int MAX = 30005;

struct node {
    int left, right;
    long long sum;
    int lazy;
    bool dirty;
};

node nodes[4*MAX];

void build_tree(int left, int right, int u) {
    nodes[u].left = left;
    nodes[u].right = right;
    nodes[u].sum = 0;
    nodes[u].lazy = 0;
    nodes[u].dirty = false;
    if (left == right) return;
    int mid = (left + right)/2;
    build_tree(left, mid, 2*u);
    build_tree(mid+1, right, 2*u + 1);
}

void get_down(int u) {
    if (!nodes[u].dirty) return;
    nodes[2*u].sum = (long long) nodes[u].lazy * (nodes[2*u].right - nodes[2*u].left + 1);
    // if update not replace use +=
    nodes[2*u].lazy = nodes[u].lazy;
    nodes[2*u].dirty = true;
    nodes[2*u + 1].sum = (long long) nodes[u].lazy * (nodes[2*u + 1].right - nodes[2*u + 1].left + 1);
    nodes[2*u + 1].lazy = nodes[u].lazy;
    nodes[2*u + 1].dirty = true;
    nodes[u].dirty = false;
}

void update(int left, int right, int value, int u) {
    if (left <= nodes[u].left && nodes[u].right <= right) {
        nodes[u].sum = (long long)value * (nodes[u].right - nodes[u].left + 1);
        // if update not replace use +=
        nodes[u].lazy = value;
        nodes[u].dirty = true;
        return;
    }
    get_down(u);
    if (left <= nodes[2*u].right) {
        update(left, right, value, 2*u);
    }
    if (right >= nodes[2*u+1].left) {
        update(left, right, value, 2*u + 1);
    }
    nodes[u].sum = nodes[2*u].sum + nodes[2*u+1].sum;
}

void query(int left, int right, long long &sum, int u) {
    if (left <= nodes[u].left && nodes[u].right <= right) {
        sum += nodes[u].sum;
        return;
    }
    get_down(u);
    if (left <= nodes[2*u].right) {
        query(left, right, sum, 2*u);
    }
    if (right >= nodes[2*u+1].left) {
        query(left, right, sum, 2*u + 1);
    }
}

// Usage
// build_tree(1, L, 1);
// update(a, b, new_value, 1);
// query(a, b, sum_as_reference, 1);
```

2.8 Range Minimum Query RMQ

```
struct RMQ { // not tested
    const static int MAXLENGTH = 2 * 1e5 + 3;
    const static int LOG_MAXLENGTH = 20;
    int rmq[MAXLENGTH][LOG_MAXLENGTH];

    void init(int* arr, int len) {
        for (int i = 0; i < len; i++)
            rmq[i][0] = arr[i];
        for (int j = 1; j < LOG_MAXLENGTH; j++)
            for (int i = 0; i < len; i++) {
                if (i + (1 << j) > len)
                    break;
                rmq[i][j] = rmq[i][j - 1];
                rmq[i][j] = min(rmq[i][j - 1], rmq[i + (1 << (j - 1))][j-1]);
            }
    }

    int range_minimum_query(int l, int r) {
        if (l > r)
            swap(l, r);

        int interval_len = r - l; // less 1

        int first_half = 1;
        while ((1 << first_half) <= interval_len)
            first_half++;
        first_half--;

        int second_half = r - (1 << first_half) + 1;
        return min(rmq[l][first_half], rmq[second_half][first_half]);
    }
};
```

2.9 Union-find Set

```
struct UnionFindSet {
        vector<int> parent;
        void init(int nn) {
                parent.resize(nn + 1);
                for (int i = 0; i < parent.size(); i++)
                        parent[i] = i;
        }

        void merge(int x, int y) {
                parent[find(x)] = find(y);
        }
        int find(int x) {
                return x == parent[x] ? x : parent[x] = find(parent[x]);
        }
        bool together(int x, int y) {
                return find(x) == find(y);
        }
};
```

2.9.1 Union-find Set - application

place holder

2.10 Bloom Filter (?) (Similar)

Can calculate hash of number sequence quickly.

If too slow, set REPEAT smaller. Or try again:)

```
#include <random>

struct BloomFilterSimilar {
        static const int MAXN = 100002;
        static const int REPEAT = 10;

        unsigned long long hash_constant[REPEAT][MAXN];
        set<unsigned long long> hash[REPEAT];

        void init_hash(int max_n=MAXN) {
                random_device rd;
            mt19937 gen(rd());
            uniform_int_distribution<unsigned long long> dis(1, ULLONG_MAX);
                for (int i = 0; i < REPEAT; i++) {
                        for (int j = 0; j < max_n; j++)
                                hash_constant[i][j] = dis(gen);
                }
        }

        vector<unsigned long long> get_hash(int val) {
                vector<unsigned long long> h(REPEAT);
                for (int i = 0; i < REPEAT; i++)
                        h[i] = hash_constant[i][val];
                return move(h);
        }

        // bool exist(int val) {
        //      return exist(get_hash(val));
        // }
        bool exist(const vector<unsigned long long>& h) {
                for (int i = 0; i < REPEAT; i++)
                        if (hash[i].find(h[i]) == end(hash[i]))
                                return false;
                return true; // possible False Positive
        }

        // void insert(int val) {
        //      insert(get_hash(val));
        // }
        void insert(const vector<unsigned long long>& h) {
                for (int i = 0; i < REPEAT; i++)
                        hash[i].insert(h[i]);
        }
};
```

3. Methodology

3.0 Greedy

> *It's Art.*

3.1 Recursive

3.1.1 Hanoi

```
void hanoi(int n, char x, char y, char z) { // 将 x 上编号 1 至 n 的圆盘移到 z, y 作辅助塔
    if (n == 1)
        printf("%d from %c to %c\n", n, x, z); // 将编号为 n 的圆盘从 x 移到 z
    else {
        hanoi(n-1, x, z, y); // 将 x 上编号 1 至 n-1 的圆盘移到 y, z 作辅助塔
        printf("%d from %c to %c\n", n, x, z); // 将编号为 n 的圆盘从 x 移到 z
        hanoi(n-1, y, x, z); // 将 y 上编号 1 至 n-1 的圆盘移到 z, x 作辅助塔
    }
}
```

3.2 Dynamic Programming

3.2.1 Longest Increasing Subsequence (LIS)

> *O(nlog(n))*

```
vector<int> sequence;
vector<int> lis(sequence.size() + 1, INT_MAX); // [i]: min value in sequence that have LIS = i
for (int i = 0; i < sequence.size(); i++) {
    int r = sequence[i];
    auto ptr = lower_bound(begin(lis), end(lis), r);
    *ptr = min(*ptr, r);
}
```

3.3 Divide and Conquer

3.3.1 binary search

3.4 Search

3.4.2 双向 BFS

3.4.3 从终点开始搜

3.4.4 迭代加深搜索 (binary increase/decrease)

> *placeholder*

3.5 Brute Force

3.5.1 子集生成

4. String

4.1 KMP

Match pattern in a string

O(n) = O(len(pattern) + len(string))

```
#define HHH 10003

int ne[HHH]; // next[], if par[i] not matched, jump to i = ne[i]
int kmp(string& par, string& ori) {
    ne[0] = -1;
    for (int p = ne[0], i = 1; i < par.length(); i++) {
        while (p >= 0 && par[p+1] != par[i])
            p = ne[p];
        if (par[p+1] == par[i])
            p++;
        ne[i] = p;
    }

    int match = 0;
    for (int p = -1, q = 0; q < ori.length(); q++) {
        while (p >= 0 && par[p+1] != ori[q])
            p = ne[p];
        if (par[p+1] == ori[q])
            p++;
        if (p + 1 == par.length()) { // match!
            p = ne[p];
            match++;
        }
    }

    return match; // return number of occurance
}

int main () {
    int n; cin >> n;
    string par, ori;
    while (cin >> par >> ori)
        cout << kmp(par, ori) << endl;
    return 0;
}
```

4.2 Boyer-Moore

4.3 Longest palindromic substring (Manacher's algorithm)

O(n)

```
int dp[HHH];
int lengthLongestPalindromSubstring(string& s) {
    memset(dp, 0, sizeof(dp));
    int ans = 0;
    int pivot = 1;
    int len = s.length() * 2; // _s0_s1_s2 = 2 * length
    for (int i = 1; i < len; i++) {
        int pBorder = pivot + dp[pivot];
        int iBorder = i;
        if (iBorder < pBorder && 2 * pivot - i > 0) {
            dp[i] = dp[2*pivot-i];
            iBorder = min(pBorder, i + dp[i]);
        }

        if (iBorder >= pBorder) {
            int j = iBorder + (iBorder % 2 ? 2 : 1);
            for (; j < len && 2*i-j > 0 && s[j/2] == s[(2*i-j)/2]; j += 2)
                ;
            iBorder = j - 2;
            dp[i] = iBorder - i;
            pivot = i;
        }
        ans = max(ans, dp[i] + 1);
    }

    return ans;
}

int main () {
    int n; cin >> n;
    string s;
    while (cin >> s)
        cout << lengthLongestPalindromSubstring(s) << endl;
    return 0;
}
```

5. Graph

5.1 Graph Structure

```
struct Graph {
    struct Edge {
        int from;
        int to;
        int len;
    };

    const static int MAXNODE = 3 * 1e5 + 2;
    vector<int> g[MAXNODE];
    vector<Edge> edge;
    int n;
    void init(int nn) {
        n = nn;
        for (int i = 0; i <= n; i++)
            g[i].clear();
        edge.clear();
    }

    void add_e(int x, int y, int len) {
        g[x].push_back(edge.size());
        edge.push_back((Edge){x, y, len});
        g[y].push_back(edge.size());
        edge.push_back((Edge){y, x, len});
    }

    void show() {
        for (int i = 0; i <= n; i++) {
            printf("%d:", i);
            for (int ie : g[i])
                printf(" %d", edge[ie].to);
            printf("\n");
        }
        printf("\n");
    }
};
```

```
struct Network {
    struct Edge {
        int to;
        int pre_edge;
        int cap;
        int flow;
    };

    vector<int> last;

    int nv; // total number of vertex, index range: [0, nv)
    vector<Edge> edge;
    void init(int _nv) {
        nv = _nv;
        vector<Edge>().swap(edge);
        vector<int>(nv + 1, -1).swap(last);
    }

    void add_e(int x, int y, int cap, int r_cap = 0) {
        Edge e{y, last[x], cap, 0};
        last[x] = edge.size();
        edge.push_back(move(e));

        Edge r_e{x, last[y], r_cap, 0};
        last[y] = edge.size();
        edge.push_back(move(r_e));
    }
    void show_edge() {
        for (int i = 0; i < nv; i++) {
            printf("%d:", i);
            for (int ie = last[i]; ie != -1; ) {
                const Edge& e = edge[ie];
                ie = e.pre_edge;
                printf(" (%d)%d/%d", e.to, e.flow, e.cap);
            }
            printf("\n");
        }
        printf("\n");
    }
}
```

5.2 Minimium Spanning Tree

5.2.1 Prim's

O((V + E)log(V))

```
struct Graph {
    struct Edge {
        int from;
        int to;
        int len;
    };

    const static int MAXNODE = 3 * 1e5 + 2;
    vector<int> g[MAXNODE];
    vector<Edge> edge;
    int n;
    void init(int nn) {
        n = nn;
        for (int i = 0; i <= n; i++)
            g[i].clear();
        edge.clear();
    }

    void add_e(int x, int y, int len) {
        g[x].push_back(edge.size());
        edge.push_back((Edge){x, y, len});
        g[y].push_back(edge.size());
        edge.push_back((Edge){y, x, len});
    }

    void show() {
        for (int i = 0; i <= n; i++) {
                printf("%d:", i);
                for (int ie : g[i])
                        printf(" %d", edge[ie].to);
                printf("\n");
        }
        printf("\n");
    }

    //
    // ---- start of Minimum Spanning Tree ---
    //
    vector<bool> added;
    vector<int> mindis; // little optimization
    void mst() {
        vector<bool>(n + 1, false).swap(added);
        vector<int>(n + 1, INT_MAX).swap(mindis);

        auto cmp = [](const Edge& a, const Edge& b) {
            return a.len > b.len;
        };
        priority_queue<Edge, vector<Edge>, decltype(cmp)> near(cmp);
        for (int i = 0; i < g[1].size(); i++) {
            const Edge& e = edge[g[1][i]];
            near.push(e);
            mindis[e.to] = e.len; // little optimization
        }
        added[1] = true;

        while (near.size()) {
            Edge cur = near.top(); near.pop();
            added[cur.to] = true;
            // add Edge cur
            for (int ie : g[cur.to]) {
                const Edge& nxe = edge[ie];
                int nx = nxe.to;
                if (!added[nx]
                 && mindis[nx] > nxe.len) { // little optimization
                    mindis[nx] = nxe.len; // little optimization
                    near.push(nxe);
                }
            }
            while (near.size() && added[near.top().to])
                near.pop();
        }
        //
    }
    //
    // ---- end of Minimum Spanning Tree ---
    //
};
```

5.2.2 Kruskal

Elog(E) + Elog(V)

```
struct Graph {
    struct Edge {
        int from;
        int to;
        int len;
    };

    const static int MAXNODE = 3 * 1e5 + 2;
    vector<int> g[MAXNODE];
    vector<Edge> edge;
    int n;
    void init(int nn) {
        n = nn;
        for (int i = 0; i <= n; i++)
            g[i].clear();
        edge.clear();
    }

    void add_e(int x, int y, int len) {
        g[x].push_back(edge.size());
        edge.push_back((Edge){x, y, len});
        g[y].push_back(edge.size());
        edge.push_back((Edge){y, x, len});
    }

    void show() {
        for (int i = 0; i <= n; i++) {
                printf("%d:", i);
                for (int ie : g[i])
                        printf(" %d", edge[ie].to);
                printf("\n");
        }
        printf("\n");
    }

    //
    // ---- start of Minimum Spanning Tree ---
    //
    UnionFindSet ufs;
    void mst() {
        ufs.init(n);
        vector<Edge> eee = edge;
        sort(begin(eee), end(eee), [](const Edge& a, const Edge& b) {
                return a.len < b.len;
        });

        int need = n - 1;
        for (const auto& e : eee) {
                if (!ufs.together(e.from, e.to)) {
                        // add Edge e
                        ufs.merge(e.from, e.to);
                        need--;
                        if (!need)
                                break;
                }
        }
    }
    //
    // ---- end of Minimum Spanning Tree ---
    //
};
```

5.3 Shortest Path

5.3.1 任意两点

```
for (k)
    for (i)
        for (j)
                d(i, j) = min(d(i, j), d(i, k) + d(j, k))
```

5.3.2 Bellman–Ford

> *Bellman–Ford algorithm is O(VE). Can be applied to situations when there is a maximun number of vertices in shortest path.*

```
for (n times of relax)
    for (each node)
        relax each node
```

5.3.3 SPFA

5.3.4 Dijkstra

> *Dijkstra is good for graphs non-negative edges.*

> *O(Vlog(E)) (?)*

```
void dijkstra(int s) {
    map<int, queue<int>> m;
    dist[s] = 0;
    m[0].push(s);
    while (m.size()) {
        if (!m.begin()->second.size()) {
            m.erase(m.begin());
            continue;
        }
        int cur = m.begin()->second.front();
        m.begin()->second.pop();
        if (done[cur]) continue;
        done[cur] = true;
        SHOW(cur, dist[cur])
        for (int next : children[cur]) {
            if (!done[next] && dist[next] > dist[cur] + edge[cur][next]) {
                dist[next] = dist[cur] + edge[cur][next];
                prev[next] = cur;
                m[dist[next]].push(next);
            }
        }
    }
    cout << endl;
}
```

If you want to write compare operator().

```
struct cmp {
    bool operator()(pair<int, int> a, pair<int, int> b) {
        return a.first > b.first;
    }
};

void dijkstra(int s) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> q;
    q.push(make_pair(0, s));
    dist[s] = 0;
    while (q.size()) {
        int cur = q.top().second;
        q.pop();
        if (done[cur]) continue;
        done[cur] = true;
        SHOW(cur, dist[cur])
        for (int next : children[cur]) {
            if (!done[next] && dist[next] > dist[cur] + edge[cur][next]) {
                dist[next] = dist[cur] + edge[cur][next];
                q.push(make_pair(dist[next], next));
                prev[next] = cur;
            }
        }
    }
}
```

5.4 Maximum Matching

5.4.1 on Bipartite Graph 二分图

> 1. *A graph is bipartite if and only if it does not contain an odd cycle.*
> 2. *A graph is bipartite if and only if it is 2-colorable, (i.e. its chromatic number is less than or equal to 2).*
> 3. *The spectrum of a graph is symmetric if and only if it's a bipartite graph.*

5.4.1.1 Hungarian algorithm 匈牙利算法

> *O(E * V)*

```
struct Network {
    struct Edge {
        int to;
        int pre_edge;
        int cap;
        int flow;
    };

    #define MAXNODE 405
    int last[MAXNODE];

    int nv; // total number of vertex, index range: [0, nv)
    vector<Edge> edge;
    void init(int _nv) {
        nv = _nv;
        edge.clear();
        fill(last, last + nv, -1);
    }

    void add_e(int x, int y, int cap, int r_cap = 0) {
        Edge e = {y, last[x], cap, 0};
        // Edge e{y, last[x], cap, 0};
        last[x] = edge.size();
        // edge.push_back(move(e));
        edge.push_back(e);

        Edge r_e = {x, last[y], r_cap, 0};
        // Edge r_e{x, last[y], r_cap, 0};
        last[y] = edge.size();
        // edge.push_back(move(r_e));
        edge.push_back(r_e);
    }
    void show_edge() {
        for (int i = 0; i < nv; i++) {
            printf("v [%d]:", i);
            for (int ie = last[i]; ie != -1; ) {
                const Edge& e = edge[ie];
                ie = e.pre_edge;
                printf(" [%d]%d/%d", e.to, e.flow, e.cap);
            }
            printf("\n");
        }
        printf("\n");
    }

    //
    // bipartite match
    // O(V * E)
    int peer[MAXNODE];
    bool went[MAXNODE];
    int bipartite_match() {
        fill(peer, peer + nv, -1);
        int ans = 0;
        for (int i = 0; i < nv; i++) {
            if (last[i] == -1 || peer[i] != -1)
                continue;
            fill(went, went + nv, false);
            if (match(i))
                ans++;
        }
        return ans;
    }
    bool match(int cur) {
        for (int ie = last[cur]; ie != -1; ) {
            const Edge& e = edge[ie];
            ie = e.pre_edge;
            int to = e.to;
            if (went[to])
                continue;
            went[to] = true;
            if (peer[to] == -1 || match(peer[to])) {
                peer[to] = cur;
                peer[cur] = to;
                return true;
            }
        }
        return false;
    }
    void show_peer() {
        for (int i = 0; i < nv; i++)
            printf("%d peer-> %d\n", i, peer[i]);
    }
    // end of
    // bipartite match
    //
};
```

5.4.1.2 Hopcroft–Karp Algorithm

> *O(sqrt(V)*E)*

```
#define MAXN 1010
#define MAXINT 0x7fffffff

int n, m, top, x, y;
int ans;

int disx[MAXN], disy[MAXN], matx[MAXN], maty[MAXN];//x,y,分别为二分图的两个点集,mat为每个点在对侧集合的匹配点,如果当前没有匹配点则为-1

struct edge {
    int to;
    edge *next;
}e[MAXN*MAXN], *prev[MAXN];

void insert(int u,int v) {
    e[++top].to = v;
    e[top].next = prev[u];
    prev[u] = &e[top];
}

bool bfs() { // 寻找最短增广路
    bool ret = 0;
    queue<int> q;
    memset(disx, 0, sizeof(disx));
    memset(disy, 0, sizeof(disy));
    for (int i = 1; i <= n; i++)
        if (matx[i] == -1)
            q.push(i); // 找到未盖点,入队
    while (!q.empty()) { // 在二分图另一个点集的非盖点中寻找增广路
        int x = q.front(); q.pop();
        for (edge *i = prev[x]; i; i = i->next)
            if (!disy[i->to]) {
                disy[i->to] = disx[x] + 1;
                if (maty[i->to]==-1)
                    ret = 1; // 找到增广路
                else
                    disx[maty[i->to]] = disy[i->to] + 1, q.push(maty[i->to]);
            }
    }
    return ret;
}

bool dfs(int x) { // 沿增广路增广
    for (edge *i = prev[x]; i; i = i->next) {
        if (disy[i->to] == disx[x] + 1) {
            disy[i->to] = 0;
            if (maty[i->to] == -1 || dfs(maty[i->to])) {
                matx[x] = i->to;
                maty[i->to] = x;
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    scanf("%d%d",&n,&m);
    memset(matx, -1, sizeof(matx));
    memset(maty, -1, sizeof(maty));
    /*for (int i=1;i<=n;i++)
    {
        scanf("%d%d",&x,&y);x++;y++;
        insert(i,x);insert(i,y);
    }建图请自动忽略*/
    while (bfs()) {
        for (int i = 1; i <= m; i++)
            if (matx[i] == -1 && dfs(i))
                ans++;
    }
    cout << ans << endl;
}
```

5.4.2 on General Graph

5.4.2.1 Blossom Algorithm

一般图最大匹配 (http://www.conlan.cc/2013/03/08/%E4%B8%80%E8%88%AC%E5%9B%BE%E6%9C%80%E5%A4%A7%E5%8C%B9%E9%85%8D/)

```
const int NMax = 230;

int Next[NMax];
int spouse[NMax];
int belong[NMax];

int findb(int a) {
    return belong[a] == a ? a : belong[a] = findb(belong[a]);
}

void together(int a,int b) {
    a = findb(a), b = findb(b);
    if (a != b)
        belong[a] = b;
}

vector<int> E[NMax];
int N;
int Q[NMax],bot;
int mark[NMax];
int visited[NMax];

int findLCA(int x,int y) {
    static int t = 0;
    t++;
    while (1) {
        if (x!=-1) {
            x = findb(x);
            if (visited[x] == t)
                return x;
            visited[x] = t;
            if (spouse[x] != -1)
                x = Next[spouse[x]];
            else x = -1;
        }
        swap(x,y);
    }
}

void goup(int a,int p) {
    while (a != p) {
        int b = spouse[a], c = Next[b];
        if (findb(c) != p)
                Next[c] = b;
        if (mark[b] == 2)
                mark[Q[bot++] = b] = 1;
        if (mark[c] == 2)
                mark[Q[bot++] = c] = 1;
        together(a,b);
        together(b,c);
        a = c;
    }
}

void findaugment(int s) {
    for (int i = 0; i < N; i++)
        Next[i] = -1, belong[i] = i, mark[i] = 0, visited[i] = -1;
    Q[0] = s;
    bot = 1;
    mark[s] = 1;
    for (int head = 0; spouse[s] == -1 && head < bot; head++) {
        int x = Q[head];
        for (int i = 0; i < (int)E[x].size(); i++) {
            int y = E[x][i];
            if (spouse[x] != y && findb(x) != findb(y) && mark[y] != 2) {
                if (mark[y] == 1) {
                    int p = findLCA(x,y);
                    if (findb(x) != p)
                        Next[x] = y;
                    if (findb(y) != p)
                        Next[y] = x;
                    goup(x,p);
                    goup(y,p);
                }
                else if (spouse[y] == -1) {
                    Next[y] = x;
                    for (int j = y; j != -1; ) {
                        int k = Next[j];
                        int l = spouse[k];
                        spouse[j] = k;
                        spouse[k] = j;
                        j = l;
                    }
                    break;
                }
                else{
                    Next[y] = x;
                    mark[Q[bot++] = spouse[y]] = 1;
                    mark[y] = 2;
                }
            }
        }
    }
}

int Map[NMax][NMax];

int main() {
        memset(Map, 0, sizeof(Map));

    scanf("%d",&N);
    int x, y;
    while (scanf("%d%d",&x,&y) != EOF) {
        x--;
        y--;
        if (x != y && !Map[x][y]) {
            Map[x][y] = Map[y][x] = 1;
            E[x].push_back(y);
            E[y].push_back(x);
        }
    }
    memset(spouse, -1, sizeof(spouse));
    for (int i = 0; i < N; i++)
        if (spouse[i] == -1)
                findaugment(i);
    int ret = 0;
    for (int i = 0; i < N; i++)
        if (spouse[i] != -1)
                ret++;
    printf("%d\n", ret);
    for (int i = 0; i < N; i++)
        if (spouse[i] != -1 && spouse[i] > i)
            printf("pair: %d %d\n", i + 1, spouse[i] + 1);
}
```

5.5 Maximum Flow Problem 最大流

5.5.1 Dinic

```
// a convenient class

struct Network {
    struct Edge {
        int to;
        int pre_edge;
        int cap;
        int flow;
    };

    #define MAXNODE 405
    int last[MAXNODE];

    int nv; // total number of vertex, index range: [0, nv)
    vector<Edge> edge;
    void init(int _nv) {
        nv = _nv;
        edge.clear();
        fill(last, last + nv, -1);
    }

    void add_e(int x, int y, int cap, int r_cap = 0) {
        Edge e = {y, last[x], cap, 0};
        // Edge e{y, last[x], cap, 0};
        last[x] = edge.size();
        // edge.push_back(move(e));
        edge.push_back(e);

        Edge r_e = {x, last[y], r_cap, 0};
        // Edge r_e{x, last[y], r_cap, 0};
        last[y] = edge.size();
        // edge.push_back(move(r_e));
        edge.push_back(r_e);
    }
    void show_edge() {
        for (int i = 0; i < nv; i++) {
            printf("v [%d]:", i);
            for (int ie = last[i]; ie != -1; ) {
                const Edge& e = edge[ie];
                ie = e.pre_edge;
                printf(" [%d]%d/%d", e.to, e.flow, e.cap);
            }
            printf("\n");
        }
        printf("\n");
    }

    //
    // maximum flow
    // dinic O(V * V * E)
    int lv[MAXNODE];
    bool mark_level(int start, int end) {
        fill(lv, lv + MAXNODE, -1);
        queue<int> q;
        lv[start] = 0;
        q.push(start);
        while (!q.empty()) {
            int cur = q.front(); q.pop();
            for (int ie = last[cur]; ie != -1; ) {
                const Edge& e = edge[ie];
                ie = e.pre_edge;
                if (e.cap != e.flow && lv[e.to] == -1) {
                    lv[e.to] = lv[cur] + 1;
                    q.push(e.to);
                }
            }
        }
        return lv[end] != -1;
    }
    void show_lv() {
        for (int i = 0; i < nv; i++) {
            printf("lv[%d] = %d\n", i, lv[i]);
        }
    }
    int augment(int cur, int end, int min_flow) {
        if (cur == end)
            return min_flow;

        int augmented_flow = 0;
        for (int ie = last[cur]; ie != -1; ) {
            Edge& e = edge[ie];
            Edge& re = edge[ie ^ 1];
            ie = e.pre_edge;
            if (lv[e.to] == lv[cur] + 1 &&
                e.cap > e.flow &&
                (augmented_flow = augment(e.to, end, min(e.cap - e.flow, min_flow)))
            ) {
                e.flow += augmented_flow;
                re.flow -= augmented_flow;
                return augmented_flow;
            }
        }
        return 0;
    }
    int dinic(int start, int end) {
        int total_flow = 0;
        int flow = 0;
        while (mark_level(start, end)) // update level
            while (flow = augment(start, end, INT_MAX)) // eat up all augmented flow
                total_flow += flow;
        return total_flow;
    }
    // end of
    // maximum flow - dinic
    //
};
```

5.5.2 Improved SAP + Gap Optimization

TODO add more optimizations

```
struct Network {
    struct Edge {
        int to;
        int pre_edge;
        int cap;
        int flow;
    };

    #define MAXNODE 405
    int last[MAXNODE];

    int nv; // total number of vertex, index range: [0, nv)
    vector<Edge> edge;
    void init(int _nv) {
        nv = _nv;
        edge.clear();
        fill(last, last + nv, -1);
    }

    void add_e(int x, int y, int cap, int r_cap = 0) {
        Edge e = {y, last[x], cap, 0};
        // Edge e{y, last[x], cap, 0};
        last[x] = edge.size();
        // edge.push_back(move(e));
        edge.push_back(e);

        Edge r_e = {x, last[y], r_cap, 0};
        // Edge r_e{x, last[y], r_cap, 0};
        last[y] = edge.size();
        // edge.push_back(move(r_e));
        edge.push_back(r_e);
    }
    void show_edge() {
        for (int i = 0; i < nv; i++) {
            printf("v [%d]:", i);
            for (int ie = last[i]; ie != -1; ) {
                const Edge& e = edge[ie];
                ie = e.pre_edge;
                printf(" [%d]%d/%d", e.to, e.flow, e.cap);
            }
            printf("\n");
        }
        printf("\n");
    }

    //
    // maximum flow
    // isap + gap O(V * V * E)
    // a bit faster than dinic
    int lv[MAXNODE];
    int lv_count[MAXNODE];
    int from_edge[MAXNODE];
    void mark_r_level(int end) {
        fill(lv, lv + nv, nv);
        fill(lv_count, lv_count + nv, 0);
        queue<int> q;

        lv[end] = 0;
        lv_count[lv[end]]++;
        q.push(end);
        while (!q.empty()) {
            int cur = q.front(); q.pop();
            for (int ie = last[cur]; ie != -1; ) {
                const Edge& e = edge[ie];
                ie = e.pre_edge;
                int to = e.to;
                if (lv[to] != nv)
                    continue;

                lv[to] = lv[cur] + 1;
                lv_count[lv[to]]++;
                q.push(to);
            }
        }
    }
    int isap_gap(int start, int end) {
        mark_r_level(end); // reverse bfs to get level of node

        int total_flow = 0;
        int cur = start;
        from_edge[start] = -1;
        while (lv[start] < nv) {
            if (cur == end) {
                int flow = INT_MAX;
                for (int x = cur; x != start; ) { // backtrack to get min cap along the path
                    int ie = from_edge[x];
                    const Edge& e = edge[ie];
                    flow = min(flow, e.cap - e.flow);
                    x = edge[ie ^ 1].to;
                }

                for ( ; cur != start; ) { // update the cap along the path
                    int ie = from_edge[cur];
                    Edge& e = edge[ie];
                    Edge& re = edge[ie ^ 1];
                    e.flow += flow;
                    re.flow -= flow;
                    cur = re.to;
                }
                total_flow += flow;
            }

            bool found = false;
            for (int ie = last[cur]; ie != -1; ) { // find the next vertex
                const Edge& e = edge[ie];
                if (e.cap != e.flow && lv[cur] == lv[e.to] + 1) {
                    cur = e.to;
                    from_edge[cur] = ie; // record the edge from which we comes
                    found = true;
                    break;
                }
                ie = e.pre_edge;
            }

            if (found)
                continue;

            lv_count[lv[cur]]--;
            if (lv_count[lv[cur]] == 0)
                break;
            int min_lv = nv;
            for (int ie = last[cur]; ie != -1; ) { // find the min level around cur vertex
                const Edge& e = edge[ie];
                ie = e.pre_edge;

                if (e.cap != e.flow)
                    min_lv = min(min_lv, lv[e.to]);
            }

            lv[cur] = min_lv + 1; // raise level of cur vertex by 1
            lv_count[lv[cur]]++;
```

```
            if (cur != start)
                cur = edge[from_edge[cur] ^ 1].to; // revert one step
        }
        return total_flow;
    }
    // end of
    // maximum flow - isap + gap
    //
};
```

5.5.3 Minimum-Cost Maximum-Flow

```
// have not tested
int n_node;
int n_edge;

int cost[405][405]; // cost[i][j] = -cost[j][i]
int residual[405][405];

bool bellman_ford(int& flow_sum, int&cost_sum) { // 0: start, n_node - 1: end
    int min_cost[405]; for (int i = 0; i < n_node; i++) min_cost[i] = INT_MAX; min_cost[0] = 0;
    int pre_node[405]; pre_node[0] = 0;
    int max_flow[405];
    int in_queue[405]; memset(in_queue, 0, sizeof(in_queue));

    queue<int> q;
    q.push(0);
    while (q.size()) {
        int cur = q.front(); q.pop();
        in_queue[cur] = 0;

        for (int i = 0; i < n_node; i++) {
            if (residual[cur][i] > 0 && min_cost[i] > min_cost[cur] + cost[cur][i]) {
                min_cost[i] = min_cost[cur] + cost[cur][i];
                pre_node[i] = cur;
                max_flow[i] = min(max_flow[cur], residual[cur][i]);

                if (in_queue[i] == 0) {
                    in_queue[i] = 1;
                    q.push(i);
                }
            }
        }
    }
    if (min_cost[n_node - 1] == INT_MAX)
        return false;
    flow_sum += max_flow[n_node - 1];
    cost_sum += max_flow[n_node - 1] * min_cost[n_node - 1];
    for (int cur = n_node - 1; cur != 0; cur = pre_node[cur]) {
        residual[pre_node[cur]][cur] -= max_flow[n_node - 1];
        residual[cur][pre_node[cur]] += max_flow[n_node - 1];
    }
    return true;
}

void min_cost_max_flow() {
    int flow_sum = 0;
    int cost_sum = 0;
    while (bellman_ford(flow_sum, cost_sum));
    cout << flow_sum << " " << cost_sum << endl;
}
```

5.5.4 More Applications and Properties

> *placeholder*

5.6 强连通分量 图的 割点, 桥, 双连通分支

`https://www.byvoid.com/blog/biconnect`

> *[点连通度与边连通度]*
>
> *在一个无向连通图中，如果有一个顶点集合，删除这个顶点集合，以及这个集合中所有顶点相关联的边以后，原图变成多个连通块，就称这个点集为割点集合。一个图的点连通度的定义为，最小割点集合中的顶点数。*
>
> *类似的，如果有一个边集合，删除这个边集合以后，原图变成多个连通块，就称这个点集为割边集合。一个图的边连通度的定义为，最小割边集合中的边数。*
>
> *[双连通图、割点与桥]*
>
> *如果一个无向连通图的点连通度大于1，则称该图是点双连通的(point biconnected)，简称双连通或重连通。一个图有割点，当且仅当这个图的点连通度为1，则割点集合的唯一元素被称为割点(cut point)，又叫关节点(articulation point)。*
>
> *如果一个无向连通图的边连通度大于1，则称该图是边双连通的(edge biconnected)，简称双连通或重连通。一个图有桥，当且仅当这个图的边连通度为1，则割边集合的唯一元素被称为桥(bridge)，又叫关节边(articulation edge)。*
>
> *可以看出，点双连通与边双连通都可以简称为双连通，它们之间是有着某种联系的，下文中提到的双连通，均既可指点双连通，又可指边双连通。*
>
> *[双连通分支]*
>
> *在图G的所有子图G'中，如果G'是双连通的，则称G'为双连通子图。如果一个双连通子图G'它不是任何一个双连通子图的真子集，则G'为极大双连通子图。双连通分支(biconnected component)，或重连通分支，就是图的极大双连通子图。特殊的，点双连通分支又叫做块。*
>
> *[求割点与桥]*
>
> *该算法是R.Tarjan发明的。对图深度优先搜索，定义DFS(u)为u在搜索树（以下简称为树）中被遍历到的次序号。定义Low(u)为u或u的子树中能通过非父子边追溯到的最早的节点，即DFS序号最小的节点。根据定义，则有：*
>
> *Low(u)=Min { DFS(u) DFS(v) (u,v)为后向边(返祖边) 等价于 DFS(v)<DFS(u)且v不为u的父亲节点 Low(v) (u,v)为树枝边(父子边) }*
>
> *一个顶点u是割点，当且仅当满足(1)或(2) (1) u为树根，且u有多于一个子树。 (2) u不为树根，且满足存在(u,v)为树枝边(或称父子边，即u为v在搜索树中的父亲)，使得DFS(u)<=Low(v)。*
>
> *一条无向边(u,v)是桥，当且仅当(u,v)为树枝边，且满足DFS(u)<Low(v)。*
>
> *[求双连通分支]*
>
> *下面要分开讨论点双连通分支与边双连通分支的求法。*
>
> *对于点双连通分支，实际上在求割点的过程中就能顺便把每个点双连通分支求出。建立一个栈，存储当前双连通分支，在搜索图时，每找到一条树枝边或后向边(非横叉边)，就把这条边加入栈中。如果遇到某时满足DFS(u)<=Low(v)，说明u是一个割点，同时把边从栈顶一个个取出，直到遇到了边(u,v)，取出的这些边与其关联的点，组成一个点双连通分支。割点可以属于多个点双连通分支，其余点和每条边只属于且属于一个点双连通分支。*
>
> *对于边双连通分支，求法更为简单。只需在求出所有的桥以后，把桥边删除，原图变成了多个连通块，则每个连通块就是一个边双连通分支。桥不属于任何一个边双连通分支，其余的边和每个顶点都属于且只属于一个边双连通分支。*
>
> *[构造双连通图]*
>
> *一个有桥的连通图，如何把它通过加边变成边双连通图？方法为首先求出所有的桥，然后删除这些桥边，剩下的每个连通块都是一个双连通子图。把每个双连通子图收缩为一个顶点，再把桥边加回来，最后的这个图一定是一棵树，边连通度为1。*
>
> *统计出树中度为1的节点的个数，即为叶节点的个数，记为leaf。则至少在树上添加(leaf+1)/2条边，就能使树达到边二连通，所以至少添加的边数就是(leaf+1)/2。具体方法为，首先把两个最近公共祖先最远的两个叶节点之间连接一条边，这样可以把这两个点到祖先的路径上所有点收缩到一起，因为一个形成的环一定是双连通的。然后再找两个最近公共祖先最远的两个叶节点，这样一对一对找完，恰好是(leaf+1)/2次，把所有点收缩到了一起。*

find articulation point (cut vertex) / bridge (cutedge) in directed / undirected graph

find and merge biconnected component in undirected graph

find and merge strongly connected component in directed graph

time complexity `O(E+V)`

```
#define NN 20002
#define MM 100002

int n;
int m;

int visit_order[NN];
int smallest_order_can_reach[NN];
int parent[NN];
int in_stack[NN];
int temp_component[NN];

vector<int> g[NN];

void merge(const vector<int>& component) {
    vector<int> out_vertex;
    int new_vertex = component.back();
    for (int v : component)
        temp_component[v] = 1;
    for (int v : component) {
        for (int o : g[v]) {
            if (!temp_component[o])
                out_vertex.push_back(o);
        }
        g[v].clear();
        g[v].push_back(new_vertex);
    }
    for (int v : component)
        temp_component[v] = 0;

    // use last vertex in the component as new vertex
    g[new_vertex] = out_vertex;
}

void dfs(int cur) {
    static int order = 0;
    static stack<int> s;

    visit_order[cur] = smallest_order_can_reach[cur] = ++order;
    s.push(cur);
    in_stack[cur] = 1;

    int subtree = 0;
    for (int next : g[cur]) {
        if (visit_order[next] == 0) {
            subtree++;
            parent[next] = cur;
            dfs(next);
            smallest_order_can_reach[cur] = min(smallest_order_can_reach[cur], smallest_order_can_reach[next]);

            // if cur is root, and subtree > 1
            // it is an articulation point
            if (visit_order[cur] == 1 && subtree > 1)
                ;

            // if cur is not root, and next cannot reach smaller vertex
            // it is an articulation point
            if (visit_order[cur] != 1 && visit_order[cur] <= smallest_order_can_reach[next])
                ;

            // if cannot use this edge to reach a smaller vertex
            // it is a bridge
            if (visit_order[cur] < smallest_order_can_reach[next])
                ;
        }
```

```
            // for undirected graph
            // update the smallness of vertex that can reach
//        else if (next != parent[cur])
//            smallest_order_can_reach[cur] = min(smallest_order_can_reach[cur], visit_order[next]);

            // for directed graph
        else if (in_stack[next])
            smallest_order_can_reach[cur] = min(smallest_order_can_reach[cur], visit_order[next]);
    }

    if (visit_order[cur] == smallest_order_can_reach[cur]) {
        // because visit_order[cur] == smallest_order_can_reach[cur]
        // and visit_order[cur] > visit_order[parent[cur]]
        // so visit_order[parent[cur]] < smallest_order_can_reach[cur]
        // so cur-parent[cur] is a bridge
        // cur is root of the biconnected component
        // so pop all util cur

        vector<int> component;
        int min_vertex = s.top();
        while (1) {
            int vertex = s.top(); s.pop();
            in_stack[vertex] = 0;
            min_vertex = min(min_vertex, vertex);
            component.push_back(vertex);
            if (vertex == cur)
                break;
        }
        // cur is the last vertex in the component
        merge(component);
    }
}

int main() {
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
    }

    dfs(1);
}
```

5.7 Topological Sort / 拓扑排序

Topological Sorting on Directed Acyclic Graph (DAG)

time complexity $O(N)$

```
struct Graph {
    struct Edge {
        int to;
    };

    const static int MAXNODE = 3 * 1e5 + 2;
    vector<int> g[MAXNODE];
    vector<Edge> edge;
    int n;
    void init(int nn) {
        n = nn;
        for (int i = 0; i <= n; i++)
            g[i].clear();
        edge.clear();
    }

    void add_e(int x, int y) {
        Edge e = {y};
        g[x].push_back(edge.size());
        edge.push_back(e);
    }

    void show() {
        for (int i = 0; i <= n; i++) {
                printf("%d:", i);
                for (int ie : g[i])
                        printf(" %d", edge[ie].to);
                printf("\n");
        }
        printf("\n");
    }

    //
        // Node index ~ [0, N)
        // matters for topological sort
        //
    int in_order[MAXNODE];
    void init_in_order() {
        fill_n(in_order, n + 1, 0);
        for (int i = 0; i < n; i++)
                for (int ie : g[i])
                        in_order[edge[ie].to]++;
    }
    bool topological_sort(vector<int>& result) {
        init_in_order();
        queue<int> q;
        for (int i = 0; i < n; i++)
                if (in_order[i] == 0)
                        q.push(i);
        while (q.size()) {
                int cur = q.front(); q.pop();
                result.push_back(cur);
                for (int ie : g[cur]) {
                        const Edge& e = edge[ie];
                        in_order[e.to]--;
                        if (in_order[e.to] == 0)
                                q.push(e.to);
                }
        }
        return result.size() == n;
    }
    void show_order() {
        for (int i = 0; i < n; i++)
                printf("in_order[%d] = %d\n", i, in_order[i]);
    }
};
```

5.8 Euler Cycle/Path, Hamilton Cycle/Path

> *place holder*

5.9 find negative (weight) Cycle on a graph

> *place holder*

6. Number + Mathematics

6.1 BigInteger + BigDecimal

6.1.1 C++ Big Integer

```
const int BASE_LENGTH = 2;
const int BASE = (int) pow(10, BASE_LENGTH);
const int MAX_LENGTH = 500;

string int_to_string(int i, int width, bool zero) {
    string res = "";
    while (width--) {
        if (!zero && i == 0) return res;
        res = (char)(i%10 + '0') + res;
        i /= 10;
    }
    return res;
}

struct bigint {
    int len, s[MAX_LENGTH];

    bigint() {
        memset(s, 0, sizeof(s));
        len = 1;
    }

    bigint(unsigned long long num) {
        len = 0;
        while (num >= BASE) {
            s[len] = num % BASE;
            num /= BASE;
            len ++;
        }
        s[len++] = num;
    }

    bigint(const char* num) {
        int l = strlen(num);
        len = l/BASE_LENGTH;
        if (l % BASE_LENGTH) len++;
        int index = 0;
        for (int i = l - 1; i >= 0; i -= BASE_LENGTH) {
            int tmp = 0;
            int k = i - BASE_LENGTH + 1;
            if (k < 0) k = 0;
            for (int j = k; j <= i; j++) {
                tmp = tmp*10 + num[j] - '0';
            }
            s[index++] = tmp;
        }
    }

    void clean() {
        while(len > 1 && !s[len-1]) len--;
    }

    string str() const {
        string ret = "";
        if (len == 1 && !s[0]) return "0";
        for(int i = 0; i < len; i++) {
            if (i == 0) {
                ret += int_to_string(s[len - i - 1], BASE_LENGTH, false);
            } else {
                ret += int_to_string(s[len - i - 1], BASE_LENGTH, true);
            }
        }
        return ret;
    }
```

```
unsigned long long ll() const {
    unsigned long long ret = 0;
    for(int i = len-1; i >= 0; i--) {
        ret *= BASE;
        ret += s[i];
    }
    return ret;
}

bigint operator + (const bigint& b) const {
    bigint c = b;
    while (c.len < len) c.s[c.len++] = 0;
    c.s[c.len++] = 0;
    bool r = 0;
    for (int i = 0; i < len || r; i++) {
        c.s[i] += (i<len)*s[i] + r;
        r = c.s[i] >= BASE;
        if (r) c.s[i] -= BASE;
    }
    c.clean();
    return c;
}

bigint operator - (const bigint& b) const {
    if (operator < (b)) throw "cannot do subtract";
    bigint c = *this;
    bool r = 0;
    for (int i = 0; i < b.len || r; i++) {
        c.s[i] -= b.s[i];
        r = c.s[i] < 0;
        if (r) c.s[i] += BASE;
    }
    c.clean();
    return c;
}

bigint operator * (const bigint& b) const {
    bigint c;
    c.len = len + b.len;
    for(int i = 0; i < len; i++)
        for(int j = 0; j < b.len; j++)
            c.s[i+j] += s[i] * b.s[j];
    for(int i = 0; i < c.len-1; i++){
        c.s[i+1] += c.s[i] / BASE;
        c.s[i] %= BASE;
    }
    c.clean();
    return c;
}

bigint operator / (const int b) const {
    bigint ret;
    int down = 0;
    for (int i = len - 1; i >= 0; i--) {
        ret.s[i] = (s[i] + down * BASE) / b;
        down = s[i] + down * BASE - ret.s[i] * b;
    }
    ret.len = len;
    ret.clean();
    return ret;
}

bool operator < (const bigint& b) const {
    if (len < b.len) return true;
    else if (len > b.len) return false;
    for (int i = 0; i < len; i++)
```

```
            if (s[i] < b.s[i]) return true;
            else if (s[i] > b.s[i]) return false;
        return false;
    }

    bool operator == (const bigint& b) const {
        return !(*this<b) && !(b<(*this));
    }

    bool operator > (const bigint& b) const {
        return b < *this;
    }
};
```

Examples

```
ASSERT((a+b).str()=="10001")
ASSERT((a+b)==bigint(10001))
ASSERT((b/2)==4999)
ASSERT(c == 12345)
ASSERT(c < 123456)
ASSERT(c > 123)
ASSERT(!(c > 123456))
ASSERT(!(c < 123))
ASSERT(!(c == 12346))
ASSERT(!(c == 12344))
ASSERT(c.str() == "12345")
ASSERT((b-1)==9998)
ASSERT(a.ll() == 2)
ASSERT(b.ll() == 9999)
```

6.1.2 The Java Approach

BigInteger & BigDecimal

6.2 Matrix

```
operator+
operator*
```

Square matrix

```
struct Matrix {
    // int height;
    // int width;

    long long value[32][32];

    Matrix operator* (const Matrix& that);
    Matrix operator+ (const Matrix& that);
    Matrix mirror();
    void show() {
        cout << endl;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                cout << this->value[i][j] << " ";
            cout << endl;
        }
    }
};

void mod_it(Matrix& temp) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            temp.value[i][j] %= m;
}

Matrix Matrix::operator* (const Matrix& that) {
    Matrix temp;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            temp.value[i][j] = 0;
            for (int k = 0; k < n; k++)
                temp.value[i][j] += this->value[i][k] * that.value[k][j];
        }
    }
    mod_it(temp);
    return temp;
}

Matrix Matrix::operator+ (const Matrix& that) {
    Matrix temp;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            temp.value[i][j] = this->value[i][j] + that.value[i][j];
    mod_it(temp);
    return temp;
}

Matrix Matrix::mirror() {
    Matrix temp;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            temp.value[i][j] = this->value[i][j];
    return temp;
}
```

6.3 Number Theory

6.3.1 欧拉函数？

6.3.2 欧几里得算法 / gcd

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

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

6.3.3 扩展欧几里得算法

http://www.cnblogs.com/frog112111/archive/2012/08/19/2646012.html

> *对于不完全为 0 的非负整数 a, b, 必然存在整数对 (x, y), 使得 gcd(a, b) = ax + by*
>
> *suppose: a > b, we want to get (x1, y1)*
>
> *(i) if b == 0, then gcd(a, b) = a = ax + 0, then x1 = 1, y1 = 0*
>
> *(ii) if b != 0:*
>
> *(1): a * x1 + b * y1 = gcd(a, b)*
>
> *(2): b * x2 + (a % b) * y2 = gcd(b, a % b)*
>
> *(1) == (2)*
>
> *so: a * x1 + b * y1 = b * x2 + (a % b) * y2*
>
> *so: a * x1 + b * y1 = b * x2 + (a - (int)(a / b) * b) * y2*
>
> *so: a * x1 + b * y1 = a * y2 + b * (x2 - (int)(a / b) * y2)*
>
> *so: x1 = y2, y1 = x2 - (int)(a / b) * y2, can get (x1, y1) from (x2, y2)*
>
> *next:*
>
> ```
>   (1): b * x2 + (a % b) * y2 = gcd(b, a % b)
> ```

> ```
> (2): (a % b) * x3 + b % (a % b) * y3 = gcd(a % b, b % (a % b))
> ```

> ```
> so: can get (x2, y2) from (x3, y3)
> ```

> ```
> next: ... until in gcd(a, b), b == 0, then xi = 1, yi = 0, go back ...
> ```

```
long long ansx, ansy, ansd;

void euclidean(long long a, long long b) {
    if (b == 0) {
        ansx = 1;
        ansy = 0;
        ansd = a;
    } else {
        euclidean(b, a % b);
        long long temp = ansx;
        ansx = ansy;
        ansy = temp - a / b * ansy;
    }
}

int main(int argc, char const *argv[]) {
    long long a, b, c;
    cin >> a >> b >> c;

    ansx = 0;
    ansy = 0;
    ansd = 0;
    euclidean(a, b);

    // now (ansx, ansy) is the answer (x, y) for a * x1 + b * y1 = gcd(a, b)
    // ansd is the a when b == 0, which is just gcd(a, b)
}
```

6.3.4 求解不定方程

> *for: p * a + q * b = c*

> *if c % gcd(a, b) == 0, then 有整数解 (p, q), else NO*

> *if we get (p0, q0) for p0 * a + q0 * b = gcd(a, b)*

> *then: for p * a + q * b = gcd(a, b) (k is any integer)*

> *p = p0 + b / gcd(a, b) * k*

> *q = q0 - a / gcd(a, b) * k*
>
> *then: for p * a + q * b = c = c / gcd(a, b) * gcd(a, b) (k is any integer)*
>
> *p = (p0 + b / gcd(a, b) * k) * c / gcd(a, b)*
>
> *q = (q0 - a / gcd(a, b) * k) * c / gcd(a, b)*

```
// after get ansx, ansy, ansd
// test if c % ansd == 0
// ansx = (ansx + b / gcd(a, b) * k) * c / gcd(a, b)
// ansy = (ansy - a / gcd(a, b) * k) * c / gcd(a, b)
// smallest: ansx % (b / gcd(a, b) + b / gcd(a, b)) % (b / gcd(a, b))
```

6.3.5 求解模线性方程（线性同余方程）

> *(a * x) % n = b % n, x = ?*
>
> *same as: a * x + n * y= b*
>
> *so: one answer for a * x + n * y= b is: x * b / gcd(a, n)*
>
> *so: one answer for (a * x) % n = b % n is: x0 = (x * b / gcd(a, n)) % n*
>
> *other answer xi = (x0 + i * (n / gcd(a, n))) % n, i = 0...gcd(a, n)-1*
>
> *smallest answer is x0 % (n / gcd(a, n) + gcd(a, n)) % gcd(a, n)*

6.3.6 求解模的逆元

> *(a * x) % n = 1, x = ?*
>
> *if gcd(a, n) != 1, then NO answer*
>
> *else:*
>
> *same as: a * x + n * y = 1*

> *can get only one answer x*

```
// after get ansx, ansy, ansd
// if ansd != 1, then NO answer
// smallest ansx = (ansx % (n / gcd(a, n)) + (n / gcd(a, n))) % (n / gcd(a, n))
```

6.3.7 中国剩余定理

6.3.8 最小公倍数

```
a / gcd(a, b) * b
```

6.3.9 分解质因数

```
long long x;
cin >> x;
for (long long factor = 2; x != 1; factor++) {
    if (x % factor == 0)
        cout << factor << " is a prime factor" << endl;
    while (x % factor == 0)
        x = x / factor;
}
```

6.3.10 因数个数

```
n = p1 ^ x1 * p2 ^ x2 * ... * pn ^ xn
total = (x1 + 1) * (x2 + 1) * ... * (xn + 1)
```

6.3.11 素数判定

> *大于 3 的质数可以被表示为 6n - 1 或 6n + 1*

```
bool is_prime(int n) {
    if (n == 1 || n % 2 == 0)
        return false;
    int t = sqrt(n);
    for (int i = 3; i <= t; i += 2)
        if (n % i == 0)
            return false;
    return true;
}
```

6.3.11.1 Miller Rabin Primality Test

> *O(k(logN)^3)*

```
class MillerRabin { // O(k(logX)^3)
    long long mulmod(long long a, long long b, long long c) {
        if (a < b)
                swap(a, b);
        long long res = 0, x = a;
        while (b > 0) {
            if (b & 1) {
                res = res + x;
                if (res >= c)
                        res -= c;
            }
            x = x * 2;
            if (x >= c)
                x -= c;
            b >>= 1;
        }
        return res % c;
    }

    long long bigmod(long long a, long long p, long long mod) {
        long long x = a, res = 1;
        while (p) {
            if (p & 1)
                res = mulmod(res, x, mod);
            x = mulmod(x, x, mod);
            p >>= 1;
        }
        return res;
    }

    bool witness(long long a, long long d, long long s, long long n) {
        long long r = bigmod(a, d, n);
        if (r == 1 || r == n - 1)
                return false;
        for (int i = 0; i < s - 1; i++) {
            r = mulmod(r, r, n);
            if (r == 1)
                return true;
            if (r == n - 1)
                return false;
        }
        return true;
    }

public:
        const vector<long long> aaa{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}; // enough for N < 2^64

    bool test(long long n) {
        if (n <= 1)
                return false;

        long long p = n - 1;
        long long s = 0;
        while (!(p & 1)) {
            p /= 2;
            s++;
        }

        for (int i = 0; i < aaa.size() && aaa[i] < n; i++)
            if (witness(aaa[i], p, s, n))
                return false;
        return true;
    }
};
```

6.3.12 进制转换

```
void convert_dec_to_base(int n, const int base) {
    if (n == 0)
        cout << 0 << endl;
    while (n != 0) {
        int e = n % base;
        cout << e << " "; // printing in reverse order
        n /= base;
    } cout << endl;
}

int convert_base_to_dec(const int s[], const int len, const int base) {
    int result = 0;
    for (int i = 0; i < len; i++)
        result = result * base + s[i];
    return result;
}
```

6.3.13 A / C

> *C(n, k) = C(n-1, k) + C(n-1, k-1)*
>
> *C(n, k) = C(n, n-k)*

```
#define MAXN 1002
#define MOD 1000000007
long long choose[MAXN][MAXN];

void init_choose_n_k() {
    for (int i = 1; i < MAXN; i++) {
        choose[i][i] = choose[i][0] = 1;
        for (int j = 1; j < i; j++)
            choose[i][j] = (choose[i-1][j-1] + choose[i-1][j]) % MOD;
    }
}
```

6.3.14 质数表

```
int is_prime[UP_LIMIT + 1];
for (int i = 1; i <= UP_LIMIT; i++) // init to 1
    is_prime[i] = 1;
for (int i = 4; i <= UP_LIMIT; i += 2) // even number is not
    is_prime[i] = 0;
for (int k = 3; k*k <= UP_LIMIT; k++) // start from 9, end at sqrt
    if (is_prime[k])
        for(int i = k*k; i <= UP_LIMIT; i += 2*k) // every two is not
            is_prime[i] = 0;
```

6.3.15 Fast Exponention

> *To calculate n ^ p % M*

```
int power_modulo(int n, int p, int M) {
    int result = 1;
    while (p > 0) {
        if (p % 2 == 1)
            result = (result*n) % M;
        p /= 2;
        n = (n*n) % M;
    }
    return result;
}
```

6.3.16 Fast Fourier Transform FFT

Reference 1 (www.gatevin.moe/acm/fft%E7%AE%97%E6%B3%95%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0/)

Reference 2 (https://github.com/marioyc/ACM-ICPC-Library/blob/master/math/fft.cpp)

*Example: calculate two number C = A * B*

```cpp
#include <iostream>
#include <vector>
#include <complex>

#ifndef M_PI
const double M_PI = acos(-1);
#endif

using namespace std;

typedef complex<long double> Complex;

void FFT(vector<Complex> &A, int s) {
    int n = A.size();
    int p = __builtin_ctz(n);

    vector<Complex> a = A;

    for (int i = 0; i < n; ++i) {
        int rev = 0;
        for (int j = 0; j < p; ++j) {
            rev <<= 1;
            rev |= ((i >> j) & 1);
        }
        A[i] = a[rev];
    }

    Complex w, wn;

    for (int i = 1; i <= p; ++i) {
        int M = (1<<i), K = (M>>1);
        wn = Complex(cos(s*2.0*M_PI/(double)M), sin(s*2.0*M_PI/(double)M));

        for (int j = 0; j < n; j += M) {
            w = Complex(1.0, 0.0);
            for (int l = j; l < K + j; ++l) {
                Complex t = w;
                t *= A[l + K];
                Complex u = A[l];
                A[l] += t;
                u -= t;
                A[l + K] = u;
                w *= wn;
            }
        }
    }

    if (s == -1) {
        for (int i = 0; i < n; ++i)
            A[i] /= (double)n;
    }
}

vector<Complex> FFT_Multiply(vector<Complex> &P, vector<Complex> &Q) {
    int n = P.size() + Q.size();

#define lowbit(x) (((x) ^ (x-1)) & (x))
    while (n != lowbit(n)) n += lowbit(n);
#undef lowbit

    P.resize(n, 0);
    Q.resize(n, 0);

    FFT(P, 1);
    FFT(Q, 1);
    vector<Complex> R;
    for (int i = 0; i < n; i++) R.push_back(P[i] * Q[i]);

    FFT(R, -1);

    return R;
}

int main() { // multiply two number
        string sa; // [a1 * 10^(?)] + [a2 * 10^(?)] + ... + [an * 10^(?)]
        string sb; // [b1 * 10^(?)] + [b2 * 10^(?)] + ... + [bm * 10^(?)]
        while (cin >> sa >> sb) {
                vector<Complex> a;
        vector<Complex> b;

        for (int i = 0; i < sa.length(); i++)
                a.push_back(Complex(sa[sa.length() - i - 1] - '0', 0)); // add [a_ * (x)^(?)]
        for (int i = 0; i < sb.length(); i++)
                b.push_back(Complex(sb[sb.length() - i - 1] - '0', 0)); // add [b_ * (x)^(?)]

                // [c1 * 10^(?)] + [c2 * 10^(?)] + ... + [cn * 10^(?)]
                // =
                // [a1 * 10^(?)] + [a2 * 10^(?)] + ... + [an * 10^(?)]
                // *
                // [b1 * 10^(?)] + [b2 * 10^(?)] + ... + [bm * 10^(?)]
        vector<Complex> c = FFT_Multiply(a, b);

        vector<int> ans(c.size());
        for (int i = 0; i < c.size(); i++)
                ans[i] = c[i].real() + 0.5; // extract [c_ * (x)^(?)] // equivalent to [c_ * (x=10)^(?)]
        for (int i = 0; i < ans.size() - 1; i++) { // process carry
            ans[i + 1] += ans[i] / 10;
            ans[i] %= 10;
        }

        bool flag = false;
        for (int i = ans.size() - 1; i >= 0; i--) { // print from most significant digit
            if (ans[i])
                cout << ans[i], flag = true;
            else if (flag || i == 0)
                cout << 0;
        }
        cout << endl;
        }
}
```

6.4 Game Theory 博弈论

6.4.1 Impartial Combinatorial Game

In combinatorial game theory, an impartial game is a game in which the allowable moves depend only on the position and not on which of the two players is currently moving, and where the payoffs are symmetric. In other words, the only difference between player 1 and player 2 is that player 1 goes first.

Impartial games can be analyzed using the Sprague–Grundy theorem.

Impartial games include Nim, Sprouts, Kayles, Quarto, Cram, Chomp, and poset games. Go and chess are not impartial, as each player can only place or move pieces of their own color. Games like ZÈRTZ and Chameleon are also not impartial, since although they are played with shared pieces, the payoffs are not necessarily symmetric for any given position.

A game that is not impartial is called a partisan game.

source: wiki (https://en.wikipedia.org/wiki/Impartial_game)

tutorial: topcoder (https://www.topcoder.com/community/data-science/data-science-tutorials/algorithm-games/)

6.4.1.1 Nim Game

One good choice: brute force to find some pattern.

We will not be able to play many of the games without decomposing them to smaller parts (sub-games), pre-computing some values for them, and then obtaining the result by combining these values.

Positions have the following properties:

- *All terminal positions are losing.*
- *If a player is able to move to a losing position then he is in a winning position.*
- *If a player is able to move only to the winning positions then he is in a losing position.*

Rules of the Game of Nim: There are n piles of coins. When it is a player's turn he chooses one pile and takes at least one coin from it. If someone is unable to move he loses (so the one who removes the last coin is the winner).

Let n1, n2, ..., nk, be the sizes of the piles. It is a losing position for the player whose turn it is if and only if n1 xor n2 xor ... xor nk = 0.

6.4.1.1 Composite Games – Sprague-Grundy Theorem and Nim Value

- *Break composite game into subgames*
- *Assign grundy number to every subgame according to which size of the pile in the Game of Nim it is equivalent to.*
- *Now we have some subgames (piles), each subgame has its grundy number (size of pile)*
- *xor all subgames*

- *Losing position of subgame has grundy number = 0.*
- *A position has grundy number = smallest number among its next positions don't have.*

If the table of grundy number is too large, we can precompute and find the pattern.

```
// hihocoer 1173
const int MAXSTATE = 2e4 + 2;

bool appear[MAXSTATE];
int sg[MAXSTATE]; // Sprague-Grundy // Nim Value
void init_sg() {
        sg[state] = 0;
        for (int state = 1; state < MAXSTATE; state++) { // sg(x) = mex{sg(y) | y是x的后继局面} // mex{a[i]}表示a中未出现的最小非负整数
                fill_n(appear, MAXSTATE, false);
                for (int nx = 0; nx < state; nx++)
                        appear[sg[nx]] = true;
                int pile_a = 1;
                int pile_b = state - pile_a;
                while (pile_a <= pile_b) {
                        appear[sg[pile_a] ^ sg[pile_b]] = true;
                        pile_a++;
                        pile_b--;
                }
                while (appear[sg[state]])
                        sg[state]++;
        }
}

int main() {

    init_sg();

        // --- start of finding pattern ---
        //
    // --- end of finding pattern ---

    int n;
    cin >> n;
    int result = 0;
    for (int i = 0; i < n; i++) {
        int a;
        cin >> a;

                // by grundy number
        // result ^= sg[a];

        // by pattern
        // if (a % 4 == 0)
        //      a--;
        // else if (a % 4 == 3)
        //      a++;
        // result ^= a;
    }
    if (result)
        cout << "Alice" << endl;
    else
        cout << "Bob" << endl;
}
```

7. Geometry

7.1 2-Dimension Space

7.1.1 Template of Point

```
struct point {
    int x, y;

    double length() {
        return sqrt(x*x + y*y);
    }

    point operator + (const point &rhs) const {
        return (point){x + rhs.x, y + rhs.y};
    }

    point operator - (const point &rhs) const {
        return (point){x - rhs.x, y - rhs.y};
    }

    long operator* (const point& b) {
        return x*b.y - y*b.x;
    }

    long cross_product(const point& b) {
      return x * b.x + y * b.y;
    }

    bool at_right_of(const point& a, const point& b) const {
        // a: relative point, b: base
        point vec_self = {x - b.x, y - b.y};
        point vec_that = {a.x - b.x, a.y - b.y};
        long product = vec_self * vec_that;
        if (product>0) return true;
        if (product==0 && vec_self.length()>vec_that.length()) return true;
        return false;
    }

    double to_point(const point& b) const {
        return sqrt(pow(x-b.x,2) + pow(y-b.y,2));
    }

    double to_segment(const point& a, const point& b) const {
        double len_ab = a.to_point(b);
        if (abs(len_ab)<E) return to_point(a);
        double r = ((a.y-y)*(a.y-b.y) - (a.x-x)*(a.x-b.x))/pow(len_ab,2);
        if (r>1 || r<0) return min(to_point(a), to_point(b));
        // projection of p is on extension of AB
        r = ((a.y - y)*(b.y - y) - (a.x - x)*(b.y - a.y))/pow(len_ab,2);
        return fabs(r*len_ab);
    }

    double to_segment(const point& a, const point& b) const {
        point vec_ab = {b.x - a.x, b.y - a.y};
        point vec_ia = {x - a.x, y - a.y};
        point vec_ib = {x - b.x, y - b.y};
        if (vec_ia.cross_product(vec_ab) < 0 || vec_ib.cross_product(vec_ab) > 0)
            return min(to_point(a), to_point(b));
        return abs(vec_ab * vec_ia) / vec_ab.length();
    } // same meaning with v1, need test

    double to_line(const point& a, const point& b) const {
        point vec_ab = {b.x - a.x, b.y - a.y};
        point vec_ia = {x - a.x, y - a.y};
        return abs(vec_ab * vec_ia) / vec_ab.length();
    } // same meaning with v1, need test

    point rotate(const point &rhs, double angle) const {
        point t = (*this) - rhs;
        double c = cos(angle), s = sin(angle);
        return (point){rhs.x + t.x * c - t.y * s, rhs.y + t.x * s + t.y * c};
    }
};
```

7.1.2 向量点乘 叉乘

a = (x1, y1)

b = (x2, y2)

i ... |i| = 1, vertical to a-b surface

7.1.3 dot product

*a dot b = x1 * x2 + y1 * y2 = |a| * |b| * cos(angle)*

if = 0: 90 degree

a dot b / |b| = a project to b

7.1.4 cross product

*a x b = x1 * y2 - x2 * y1 = |a| * |b| * sin(angle) * i*

if < 0: b is at left of a

if = 0: a, b in a line

if 0: b is at right of a

a x b = area of 平行四边形

a x b x c = area of 平行六面体, c = (x3, y3)

7.1.5 直线公式

*(x, y) = (x1, y1) + k * ((x2, y2) - (x1, y1))*

7.1.6 Convex Hull

Gift Wrapping

place holder

QuickHull

place holder

Graham scan

O(VlogV)

```
struct Point {
    long x;
    long y;

    bool at_right_of(Point& that, Point& base) {
        Point vec_self = {this->x - base.x, this->y - base.y};
        Point vec_that = {that.x - base.x, that.y - base.y};

        long product = vec_self * vec_that;
        if (product > 0)
            return true; // "this" is at right of "that"
        if (product == 0 && vec_self.length() > vec_that.length())
            return true; // "this" is at right of "that"
        return false; // "this" is NOT at right of "that"
    };
    long operator* (Point& that) {
        return this->x * that.y - this->y * that.x;
    };
    double distance_to(Point& that) {
        long x_diff = this->x - that.x;
        long y_diff = this->y - that.y;
        return sqrt(x_diff * x_diff + y_diff * y_diff);
    };
    double length() {
        return sqrt(this->x * this->x + this->y * this->y);
    }
};

Point p[1005];
int my_stack[1005];
int n, l, my_stack_top = -1;

bool compare(Point p1, Point p2) {
    return p1.at_right_of(p2, p[0]);
}

void push(int index) {
    my_stack[++my_stack_top] = index;
}
int pop() {
    int temp = my_stack[my_stack_top--];
    return temp;
}

void graham_scan() {
    push(0);
    push(1);

    int pre;
    int prepre;
    for (int i = 2; i < n; i++) {
        pre = my_stack_top;
        prepre = my_stack_top - 1;
        while (p[i].at_right_of(p[my_stack[pre]], p[my_stack[prepre]])) {
            pop();
            if (my_stack_top == 0)
                break;
            pre = my_stack_top;
            prepre = my_stack_top - 1;
        }
        push(i);
    }

    int last = my_stack_top;
    if (p[0].at_right_of(p[my_stack[last]], p[my_stack[pre]]))
```

```
        pop();
}

int main(int argc, char const *argv[]) {
    cin >> n >> l;

    int minimun = 0;
    for (int i = 0; i < n; ++i) {
        int temp_x, temp_y;
        cin >> temp_x >> temp_y;
        p[i] = {temp_x, temp_y};

        if ((p[i].y < p[minimun].y) || (p[i].y == p[minimun].y && p[i].x < p[minimun].x))
            minimun = i;
    }

    Point temp = {p[minimun].x, p[minimun].y}; // swap lowest and most left point to p[0]
    p[minimun] = p[0];
    p[0] = temp;

    sort(p + 1, p + n, compare); // use p[0] as base, sort according to polar angle
    graham_scan();
    // now all points in the stack is on Convex Hull // size of stack = 1 + stack_top

    for (int i = 0; i <= my_stack_top; i++)
        cout << "point " << my_stack[i] << " is on Convex Hull" << endl;
}
```

8. Tricks + Miscellaneous

8.1 Bit Manipulation

```
#define GET_BIT(n, i) (((n) & (1LL << ((i)-1))) >> ((i)-1)) // i start from 1
#define SET_BIT(n, i) ((n) | (1LL << ((i)-1)))
#define CLR_BIT(n, i) ((n) & ~(1LL << ((i)-1)))
```

8.1 Cantor Expansion / Reverse Cantor Expansion

> *for hashing, or ...*

```
long long factorial(int n) {
    if (n == 0)
        return 1;

    long long ans = n;
    for (int i = 1; i < n; i++)
        ans = ans * i;
    return ans;
}

long long cantor_expansion(int permutation[], int n) {
    // input: (m-th permutation of n numbers, n)
    // return: m
    int used[n + 1];
    memset(used, n, sizeof(used));

    long long ans = 0;
    for (int i = 0; i < n; i++) {
        int temp = 0;
        used[permutation[i]] = 1;
        for (int j = 1; j < permutation[i]; j++)
            if (used[j] != 1)
                temp += 1;
        ans += factorial(n - 1 - i) * temp;
    }

    return ans + 1;
}

void reverse_cantor_expansion(int n, long long m) {
    // m-th permutation of n numbers
    int ans[n + 1], used[n + 1];
    memset(ans, -1, sizeof (ans));
    memset(used, 0, sizeof (used));

    m = m - 1;
    for (int i = n - 1; i >= 0; i--) {
        long long fac = factorial(i);
        int temp = m / fac + 1;
        m = m - (temp - 1) * fac;
        for (int j = 1; j <= temp; j++)
            if (used[j] == 1)
                temp++;

        ans[n - i] = temp;
        used[temp] = 1;
    }

    for (int i = 1; i < n + 1; i++)
        cout << ans[i] << " ";
    cout << "\n";
}
```

8.2 pass 2-D array

```
// The parameter is a 2D array
int array[10][10];
void passFunc(int a[][10]) {
    // ...
}
passFunc(array);

// The parameter is an array containing pointers
int *array[10];
for(int i = 0; i < 10; i++)
    array[i] = new int[10];
void passFunc(int *a[10]) {
    // ...
}
passFunc(array);

// The parameter is a pointer to a pointer
int **array;
array = new int *[10];
for(int i = 0; i <10; i++)
    array[i] = new int[10];
void passFunc(int **a) {
    // ...
}
passFunc(array);
```

8.3 Binary Display

```
#include <bitset>
void show_binary(unsigned long long x) {
        printf("%s\n", bitset<64>(x).to_string().c_str());
}
```

8.4 Fast Log

> *Built-in `log(double)` is not accurate for integer.*
>
> *Should `(int)(log(double)+0.000....001)`*

```
int fastlog(unsigned long long x, unsigned long long base) {
        // ERROR VALUE IF X == BASE == ULLONG_MAX

        const unsigned long long HALF = 1ULL << 32;
        unsigned long long cache[7];
#define INIT(i) { cache[i] = base; if (base < HALF) base *= base; else base = ULLONG_MAX; }
        INIT(0); INIT(1); INIT(2); INIT(3); INIT(4); INIT(5); INIT(6);
#undef INIT

        int ret = -(x == 0);
#define S(i, k) if (x >= cache[i]) ret += k, x /= cache[i]; else return ret;
        S(6, 64); S(5, 32); S(4, 16); S(3, 8); S(2, 4); S(1, 2); S(0, 1);
#undef S
}
```

8.5 Squre Root

```
long long sq(long long a) {
    long long l = 1;
    long long r = a + 1;
    while (l + 1 < r) {
        long long m = (l + r) / 2;
        if (a / m < m)
            r = m;
        else
            l = m;
    }
    return l;
}
```