Índice

1. C++ Cheat Sheet **2**

2. Data Structures **6**
2.1. Fenwick Tree . . . 6
2.2. Fenwick Tree 2D . . . 6
2.3. Segment Tree . . . 7
2.4. Segment Tree Lazy . . . 8
2.5. Wavelet Tree . . . 8
2.6. Union-Find . . . 10

3. General Algorithms **10**
3.1. Binary Search . . . 10
3.2. Ternary Search . . . 11
3.3. Brute Force . . . 11
3.3.1. Generate all combinations . . . 11

4. Dynamic Programming **12**
4.1. Knapsack . . . 12
4.2. Divide & Conquer Optimization . . . 13

5. Graphs **14**
5.1. BFS . . . 14
5.2. DFS . . . 14
5.3. Dijkstra . . . 14
5.4. Max Flow : Dinic . . . 15
5.5. Minimum Spanning Tree (Kruskal & Prim) . . . 16
5.6. LCA (Lowest Common Ancestor) . . . 17
5.7. Diameter of a Tree . . . 18
5.8. Articulation Points, Cut Edges, Biconnected Components . . . 19
5.9. Centroid Decomposition . . . 19

6. Mathematics **20**
6.1. Euclidean Algorithm . . . 20
6.2. Prime Numbers . . . 21
6.3. Modular Binomial Coefficient . . . 21
6.4. Modular Multinomial Coefficient . . . 22
6.5. Modular Fibonacci . . . 22
6.6. Binary Modular Exponentiation . . . 22
6.7. Integer Root Square . . . 22

7. Geometry **23**
7.1. Geometry 2D Utils . . . 23
7.2. Trigonometry . . . 24
7.3. Polygon Area . . . 25
7.4. Point Inside Polygon . . . 25
7.5. Convex Hull . . . 26
7.6. Green's Theorem . . . 26

8. Strings **27**
8.1. Rolling Hashing . . . 27
8.2. Suffix Array . . . 28
8.3. KMP (Knuth Morris Pratt) . . . 28
8.4. Shortest Repeating Cycle . . . 29
8.5. Trie . . . 29

1. C++ Cheat Sheet

```cpp
// Note: This Cheat Sheet is by no means complete
// If you want a thorough documentation of the Standard C++ Library
// please refer to this link: http://www.cplusplus.com/reference/

/* ========= */
// Template  //
/* ========= */
#include <bits/stdc++.h> // import everything in one shot
using namespace std;
// defines
#define rep(i,a,b) for(int i = a; i <= b; ++i)
#define invrep(i,b,a) for(int i = b; i >= a; --i)
#define umap unordered_map
#define uset unordered_set
// typedefs
typedef unsigned int uint;
typedef unsigned long long int ull;
typedef long long int ll;
typedef vector<int> vi;
typedef pair<int,int> ii;
typedef tuple<int,int,int> iii;

int main() {
    ios::sync_with_stdio(false); // for greater performance if only using cin/cout
    cin.tie(0); // for greater performance if only using cin/cout
    // setvbuf(stdout, NULL, _IONBF, 0);  // if debugging with printf/scanf
    return 0;
}

/* ================== */
/* Reading from stdin */
/* ================== */
scanf("%d",&a); //int
scanf("%x",&a); // int in hexadecimal
scanf("%llx",&a); // long long in hexadecimal
scanf("%lld",&a); // long long int
scanf("%c",&c); // char
scanf("%s",buffer); // string without whitespaces
scanf("%f",&f); // float
scanf("%lf",&d); // double
scanf("%d %*s %d",&a,&b);  //* = consume but skip

// read until EOL
//  - EOL not included in buffer
//  - EOL is not consumed
//  - nothing is written into buffer if EOF is found
scanf(" %[^\n]",buffer);

//reading until EOL or EOF
//  - EOL not included in buffer
//  - EOL is consumed
//  - works with EOF
char* output = gets(buffer);
if(feof(stind)) {} // EOF file found
if(output == buffer) {} // succesful read
if(output == NULL) {} // EOF found without previous chars found
//example
while(gets(buffer) != NULL) {
    puts(buffer);
    if(feof(stdin)) {
        break;
    }
}

// read single char
getchar();
while(true) {c = getchar(); if (c == EOF || c== '\n') break;}

/* ================== */
/* Printing to stdout */
/* ================== */
printf("%d",a); // int
printf("%u",a); // unsigned int
printf("%lld",a); // long long int
printf("%llu",a); // unsigned long long int
printf("%c",c); // char
printf("%s",buffer); // string until \0
printf("%f",f); // float
printf("%lf",d); // double
printf("%0*.*f",x,y,f); // padding = 0, width = x, decimals = y
printf("(%.5s)\n", buffer); // print  at most the first five characters (safe to use on
    short strings)

// print at most first n characters (safe)
printf("(%.*s)\n", n, buffer);  // make sure that n is integer (with long long I had
    problems)
//string + \n
puts(buffer);

/* ==================== */
/* Reading from c string */
/* ==================== */

// same as scanf but reading from s
int sscanf ( const char * s, const char * format, ...);

/* ==================== */
/* Printing to c string */
/* ==================== */
// Same as printf but writing into str, the number of characters is returned
// or negative if there is failure
int sprintf ( char * str, const char * format, ... );
//example:
int n=sprintf (buffer, "%d plus %d is %d", a, b, a+b);
printf ("[%s] is a string %d chars long\n",buffer,n);
```

```

/* ====================== */
/* Peek last char of stdin */
/* ====================== */
bool peekAndCheck(char c) {
    char c2 = getchar();
    ungetc(c2, stdin); // return char to stdin
    return c == c2;
}

/* ================ */
/* Reading from cin */
/* ================ */
// reading a line of unknown length
string line;
getline(cin, line);
while(getline(cin, line)) {}

/* ====================== */
/* USING PAIRS AND TUPLES */
/* ====================== */
// ii = pair<int,int>
ii p(5,5);
ii p = make_pair(5,5)
ii p = {5, 5};
int x = p.first, y = p.second;
// iii = tuple<int,int,int>
iii t(5,5,5);
tie(x,y,z) = t;
tie(x,y,z) = make_tuple(5,5,5);
get<0>(t)++;
get<1>(t)--;

/* ================================ */
/* CONVERTING FROM STRING TO NUMBERS */
/* ================================ */
//------------------------
// string to int
// option #1:
int atoi (const char * str);
// option #2:
sscanf(string,"%d",&i);
//-------------------------
// string to long int:
// option #1:
long int strtol (const char* str, char** endptr, int base);
// it only works skipping whitespaces, so make sure your numbers
// are surrounded by whitespaces only
// Example:
    char szNumbers[] = "2001 60c0c0 -110111010011010010000 0x6fffff";
    char * pEnd;
    long int li1, li2, li3, li4;
    li1 = strtol (szNumbers,&pEnd,10);
    li2 = strtol (pEnd,&pEnd,16);
    li3 = strtol (pEnd,&pEnd,2);
    li4 = strtol (pEnd,NULL,0);
    printf ("The decimal equivalents are: %ld, %ld, %ld and %ld.\n", li1, li2, li3, li4);
// option #2:
long int atol ( const char * str );
// option #3:
sscanf(string,"%ld",&l);
//-------------------------
// string to long long int:
// option #1:
long long int strtoll (const char* str, char** endptr, int base);
// option #2:
sscanf(string,"%lld", &l);
//-------------------------
// string to double:
// option #1:
double strtod (const char* str, char** endptr); //similar to strtol
// option #2:
double atof (const char* str);
// option #3:
sscanf(string,"%lf", &d);

/* ======================== */
/* C STRING UTILITY FUNCTIONS */
/* ======================== */
int strcmp ( const char * str1, const char * str2 ); // (-1,0,1)
int memcmp ( const void * ptr1, const void * ptr2, size_t num ); // (-1,0,1)
void * memcpy ( void * destination, const void * source, size_t num );

/* ========================== */
/* C++ STRING UTILITY FUNCTIONS */
/* ========================== */
// read tokens from string
string s = "tok1 tok2 tok3";
string tok;
stringstream ss(s);
while (getline(ss, tok, ' ')) printf("tok = %s\n", tok.c_str());

// split a string by a single char delimiter
void split(const string &s, char delim, vector<string> &elems) {
    stringstream ss(s);
    string item;
    while (getline(ss, item, delim))
        elems.push_back(item);
}

// find index of string or char within string
string str = "random";
std:size_t pos = str.find("ra");
std:size_t pos = str.find('m');
if (pos == string::npos) // not found

// substrings
string subs = str.substr(pos, length);
```

```cpp
string subs = str.substr(pos); // default: to the end of the string

// std::string from cstring's substring
const char* s = "bla1 bla2";
int offset = 5, len = 4;
string subs(s + offset, len); // bla2

// ------------------------
// string comparisons
int compare (const string& str) const;
int compare (size_t pos, size_t len, const string& str) const;
int compare (size_t pos, size_t len, const string& str,
                         size_t subpos, size_t sublen) const;
int compare (const char* s) const;
int compare (size_t pos, size_t len, const char* s) const;

// examples
// 1) check string begins with another string
string prefix = "prefix";
string word = "prefix suffix";
word.compare(0, prefix.size(), prefix);

/* ==================== */
/*  OPERATOR OVERLOADING */
/* ==================== */

//-------------------------
// method #1: inside struct
struct Point {
    int x, y;
    bool operator<(const Point& p) const {
        if (x != p.x) return x < p.x;
        return y < p.y;
    }
    bool operator>(const Point& p) const {
        if (x != p.x) return x > p.x;
        return y > p.y;
    }
    bool operator==(const Point& p) const {
        return x == p.x && y == p.y;
    }
};

//-------------------------
// method #2: outside struct
struct Point {int x, y; };
bool operator<(const Point& a, const Point& b) {
    if (a.x != b.x) return a.x < b.x;
    return a.y < b.y;
}
bool operator>(const Point& a, const Point& b) {
    if (a.x != b.x) return a.x > b.x;
    return a.y > b.y;
}
bool operator==(const Point& a, const Point& b) {
    return a.x == b.x && a.y == b.y;
}

// Note: if you overload the < operator for a custom struct,
// then you can use that struct with any library function
// or data structure that requires the < operator
// Examples:
priority_queue<Point> pq;
vector<Point> pts;
sort(pts.begin(), pts.end());
lower_bound(pts.begin(), pts.end(), {1,2});
upper_bound(pts.begin(), pts.end(), {1,2});
set<Point> pt_set;
map<Point, int> pt_map;


/* ================== */
/* CUSTOM COMPARISONS */
/* ================== */
// method #1: operator overloading
// method #2: custom comparison function
bool cmp(const Point& a, const Point& b) {
    if (a.x != b.x) return a.x < b.x;
    return a.y < b.y;
}
// method #3: functor
struct cmp {
    bool operator()(const Point& a, const Point& b) {
        if (a.x != b.x) return a.x < b.x;
        return a.y < b.y;
    }
};
// without operator overloading, you would have to use
// an explicit comparison method when using library
// functions or data structures that require sorting
priority_queue<Point, vector<Point>, cmp> pq;
vector<Point> pts;
sort(pts.begin(), pts.end(), cmp);
lower_bound(pts.begin(), pts.end(), {1,2}, cmp);
upper_bound(pts.begin(), pts.end(), {1,2}, cmp);
set<Point, cmp> pt_set;
map<Point, int, cmp> pt_map;

/* ======================== */
/* VECTOR UTILITY FUNCTIONS */
/* ======================== */
vector<int> myvector;
myvector.push_back(100);
myvector.pop_back(); // remove last element
myvector.back(); // peek reference to last element
myvector.front(); // peek reference to first element
myvector.clear(); // remove all elements
// sorting a vector
```

```cpp
vector<int> foo;
sort (foo.begin(), foo.end());
sort (foo.begin(), foo.end(), std::less<int>()); // increasing
sort (foo.begin(), foo.end(), std::greater<int>()); // decreasing

/* ==================== */
/* SET UTILITY FUNCTIONS */
/* ==================== */
set<int> myset;
myset.begin(); // iterator to first elemnt
myset.end(); // iterator to after last element
myset.rbegin(); // iterator to last element
myset.rend(); // iterator to before first element
for (auto it = myset.begin(); it != myset.end(); ++it) { do_something(*it); } // left -> right
for (auto it = myset.rbegin(); it != myset.rend(); ++it) { do_something(*it); } // right -> left
for (auto& i : myset) { do_something(i); } // left->right shortcut
auto ret = myset.insert(5); // ret.first = iterator, ret.second = boolean (inserted / not inserted)
int count = mysert.erase(5); // count = how many items were erased
if (!myset.empty()) {}
// custom comparator 1: functor
struct cmp { bool operator()(int i, int j) { return i > j; } };
set<int, cmp> myset;
// custom comparator 2: function
bool cmp(int i, int j) { return i > j; }
set<int, bool(*)(int,int)> myset(cmp);

/* ==================== */
/* MAP UTILITY FUNCTIONS */
/* ==================== */
struct Point {int x, y; };
bool operator<(const Point& a, const Point& b) {
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}
map<Point, int> ptcounts;

// --------------------
// inserting into map

// method #1: operator[]
// it overwrites the value if the key already exists
ptcounts[{1, 2}] = 1;

// method #2: .insert(pair<key, value>)
// it returns a pair { iterator(key, value) , bool }
// if the key already exists, it doesn't overwrite the value
void update_count(Point& p) {
    auto ret = ptcounts.emplace(p, 1);
    // auto ret = ptcounts.insert(make_pair(p, 1)); //
    if (!ret.second) ret.first->second++;
}

// ------------------------
// generating ids with map
int get_id(string& name) {
    static int id = 0;
    static map<string,int> name2id;
    auto it = name2id.find(name);
    if (it == name2id.end())
        return name2id[name] = id++;
    return it->second;
}

/* ======================== */
/* BITSET UTILITY FUNCTIONS */
/* ======================== */
bitset<4> foo; // 0000
foo.size(); // 4
foo.set(); // 1111
foo.set(1,0); // 1011
foo.test(1); // false
foo.set(1); // 1111
foo.test(1); // true

/* ============== */
/* RANDOM INTEGERS */
/* ============== */
#include <cstdlib>
#include <ctime>
srand(time(NULL));
int x = rand() % 100; // 0-99
int randBetween(int a, int b) { // a-b
    return a + (rand() % (1 + b - a));
}

/* ======= */
/* CLIMITS */
/* ======= */
#include <climits>
INT_MIN
INT_MAX
UINT_MAX
LONG_MIN
LONG_MAX
ULONG_MAX
LLONG_MIN
LLONG_MAX
ULLONG_MAX

/* ============== */
/* Bitwise Tricks */
/* ============== */

// amount of one-bits in number
int __builtin_popcount(int x);
int __builtin_popcountl(long x);
```

```
int __builtin_popcountll(long long x);

// amount of leading zeros in number
int __builtin_clz(int x);
int __builtin_clzl(long x);
int __builtin_clzll(ll x);

// binary length of non-negative number
int bitlen(int x) { return sizeof(x) * 8 - __builtin_clz(x); }
int bitlen(ll x) { return sizeof(x) * 8 - __builtin_clzll(x); }

// index of most significant bit
int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }
int log2(ll x) { return sizeof(x) * 8 - __builtin_clzll(x) - 1; }

// reverse the bits of an integer
int reverse_bits(int x) {
    int v = 0;
    while (x) v <<= 1, v |= x&1, x >>= 1;
    return v;
}

// get string binary representation of an integer
string bitstring(int x) {
    int len = sizeof(x) * 8 - __builtin_clz(x);
    if (len == 0) return "0";

    char buff[len+1]; buff[len] = '\0';
    for (int i = len-1; i >= 0; --i, x >>= 1)
        buff[i] = (char)('0' + (x&1));
    return string(buff);
}

/* ================= */
/* Hexadecimal Tricks */
/* ================= */

// get string hex representation of an integer
string to_hex(int num) {
    static char buff[100];
    static const char* hexdigits = "0123456789abcdef";
    buff[99] = '\0';
    int i = 98;
    do {
        buff[i--] = hexdigits[num & 0xf];
        num >>= 4;
    } while (num);
    return string(buff+i+1);
}

// ['0'-'9' 'a'-'f'] ->  [0 - 15]
int char_to_digit(char c) {
    if ('0' <= c && c <= '9')
        return c - '0';
    return 10 + c - 'a';
}

/* ============ */
/* Other Tricks */
/* ============ */
// swap stuff
int x = 1,  y = 2;
swap(x, y);

/* =========== */
/*    TIPS     */
/* =========== */
// 1) do not use .emplace(x, y) if your struct doesn't have an explicit constructor
//    instead you can use .push({x, y})
// 2) be careful while mixing scanf() with getline(), scanf will not consume \n unless
//    you explicitly tell it to do so (e.g scanf("%d\n", &x)) )
```

2. Data Structures

2.1. Fenwick Tree

```
struct FenwickTree {
  vector<int> ft;
  FenwickTree(int n) { ft.assign(n+1, 0); }
  // prefix sum query (sum in range 1 .. b)
  int psq(int b) {
    int sum = 0;
    for (; b; b -= (b & -b)) sum += ft[b];
    return sum;
  }
  // range sum query (sum in range a .. b)
  int rsq(int a, int b) {
    return psq(b) - psq(a-1);
  }
  // increment k'th value by v (and propagate)
  void add(int k, int v) {
    for (; k < ft.size(); k += (k & -k)) ft[k] += v;
  }
  // increment range [i ... j] with v (and propagate)
  void range_add(int i, int j, int v) {
    add(i, v); add(j+1, -v);
  }
};
```

2.2. Fenwick Tree 2D

```
struct FenwickTree2D {
    vector<vector<int>> ft;
    int n, m;

    FenwickTree2D(int n, int m) : n(n), m(m) {
```

```
        ft.assign(n, vector<T>(m, 0));
    }

    void add(int r, int c, int value) {
        for (int i = r; i < n; i += (i&-i))
            for (int j = c; j < m; j += (j&-j))
                ft[i][j] += value;
    }

    // sum[(1, 1), (r, c)]
    int sum(int r, int c) {
        int res = 0;
        for (int i = r; i; i -= (i&-i))
            for (int j = c; j; j -= (j&-j))
                res += ft[i][j];
        return res;
    }

    // sum[(r1, c1), (r2, c2)]
    int sum(int r1, int c1, int r2, int c2) {
        return sum(r2, c2) - sum(r1 - 1, c2) - sum(r2, c1 - 1) + sum(r1 - 1, c1 - 1);
    }

    int get(int r, int c) {
        return sum(r, c, r, c);
    }

    int set(int r, int c, int value) {
        add(r, c, -get(r, c) + value);
    }
};
```

2.3. Segment Tree

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;

// Example of SegmentTree for rmq (range minimum query)
// Note: instead of storing the minimum value, each node will store
//  the index of the leftmost position of the range in which the minimum
//  value of that range is found

struct SegmentTreeRMQ {
    vi arr; // store original array values
    vi tree; // store nodes of segment tree
    vi leaf; // store index of leaf nodes in segment tree
    int n; // number of leaf nodes (length of arr)
    inline int left (int u) { return u << 1; } // index of left child
    inline int right(int u) { return (u << 1) + 1; } // index of right child

    void build(int u, int i, int j) {
        if (i == j) { // base case: a leaf node
            tree[u] = i;
            leaf[i] = u;
        } else { // recursive case
            int lu = left(u), ru = right(u), m = (i+j)/2;
            build(lu, i, m);
            build(ru, m+1, j);
            // store the index of the minimum value,
            // in case of draw choose the leftmost
            int ii = tree[lu], jj = tree[ru];
            tree[u] = (arr[ii] <= arr[jj]) ? ii : jj;
        }
    }

    // update arr[i] with new_val, and propagate updates in the tree
    // from leaf[i] upwards
    void update(int i, int new_val) {
        arr[i] = new_val;
        int u = leaf[i] >> 1;
        while (u) {
            int lu = left(u), ru = right(u);
            int min_i = (arr[tree[lu]] <= arr[tree[ru]]) ? tree[lu] : tree[ru];
            if (min_i == tree[u]) break; // optimization: no changes, interrupt updates
            // update and move to next parent
            tree[u] = min_i;
            u >>= 1;
        }
    }

    // query for range [a,b], considering that we are at node u
    // which is in charge of range [i, j]
    int query(int a, int b, int u, int i, int j) {
        // case 1: no overlap -> return some neutral / invalid value
        if (j < a or b < i) return -1;
        // case 2: full overlap -> return cached answer
        if (a <= i and j <= b) return tree[u];

        // case 3: partial overlap -> need recursion and merge of answers
        int lu = left(u), ru = right(u), m = (i+j)/2;
        int ii = query(a, b, lu, i, m);
        int jj = query(a, b, ru, m+1, j);
        if (ii == -1) return jj;
        if (jj == -1) return ii;
        return (arr[ii] <= arr[jj]) ? ii : jj;
    }

    // overloading for easier use
    int query(int a, int b) { return query(a, b, 1, 0, n - 1); }

    SegmentTreeRMQ(const vi& _arr) {
        arr = _arr; // copy content for local usage
        n = arr.size();
        leaf.resize(n);
        tree.resize(4 * n + 5); // reserve enough space for the worst case
        build(1, 0, n - 1); // recursive build from root
    }
```

```

};

// usage
int main() {
    vi arr = { 18, 17, 13, 19, 15, 11, 20 };
    SegmentTreeRMQ st(arr);
    st.query(1, 3);
    st.update(5, 100);
    return 0;
}
```

2.4. Segment Tree Lazy

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;

struct SegmentTreeLazy {
    vi arr, tree, lazy;
    int n;
    inline int left (int p) { return p << 1; }
    inline int right(int p) { return (p << 1) + 1; }

    // build the tree
    void build(int node, int a, int b) {
        if(a > b) return; // out of range
        if(a == b) { // leaf node
            tree[node] = arr[a]; // init value
            return;
        }
        int lnode = left(node), rnode = right(node);
        build(lnode, a, (a+b)/2); // init left child
        build(rnode, (a+b)/2 + 1, b); // init right child
        tree[node] = max(tree[lnode], tree[rnode]); // init root value
    }

    // increment elements within range [i, j] with value
    void range_update(int node, int a, int b, int i, int j, int value) {
        if(lazy[node] != 0) { // this node needs to be updated
            tree[node] += lazy[node]; // update it
            if(a != b) {
                lazy[left(node)] += lazy[node]; // mark left child as lazy
                lazy[right(node)] += lazy[node]; // mark right child as lazy
            }
            lazy[node] = 0; // Reset it
        }

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

        if(a >= i && b <= j) { // segment is fully within range
            tree[node] += value;
            if(a != b) { // not leaf node
                lazy[left(node)] += value;
                lazy[right(node)] += value;
            }
            return;
        }

        range_update(left(node), a, (a+b)/2, i, j, value); // updating left child
        range_update(right(node), 1+(a+b)/2, b, i, j, value); // updating right child
        tree[node] = max(tree[left(node)], tree[right(node)]); // Updating root with max
             value
    }

    // query tree to get max element value within range [i, j]
    int range_query(int node, int a, int b, int i, int j) {
        if(a > b || a > j || b < i) return INT_MIN; // out of range
        if(lazy[node] != 0) { // this node needs to be updated
            tree[node] += lazy[node]; // update it
            if(a != b) {
                lazy[left(node)] += lazy[node]; // mark child as lazy
                lazy[right(node)] += lazy[node]; // mark child as lazy
            }
            lazy[node] = 0; // reset it
        }
        if(a >= i && b <= j) // current segment is totally within range [i, j]
            return tree[node];
        int q1 = range_query(left(node), a, (a+b)/2, i, j); // Query left child
        int q2 = range_query(right(node), 1+(a+b)/2, b, i, j); // Query right child
        return = max(q1, q2); // Return final result
    }

    SegmentTree(const vi& A) {
        arr = A; n = (int)A.size();              // copy content for local usage
        tree.assign(4 * n, 0);            // create large enough vector of zeroes
        lazy.assign(4 * n, 0);
        build(1, 0, n - 1);                                  // recursive build
    }
    // overloading
    int range_update(int i, int j, int value) { return range_update(1, 0, n - 1, i, j,
         value); }
    int range_query(int i, int j) { return range_query(1, 0, n - 1, i, j); }

};

// usage
int main() {
    vi A = { 18, 17, 13, 19, 15, 11, 20 };
    SegmentTreeLazy stl(A);
    stl.range_update(1, 5, 100);
    stl.range_query(1, 3);
    return 0;
}
```

2.5. Wavelet Tree

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int>::iterator iter;

struct WaveTree {
  vector<vector<int>> r0; int n, s;
  vector<int> arrCopy;

  void build(iter b, iter e, int l, int r, int u) {
    if (l == r)
      return;
    int m = (l+r)/2;
    r0[u].reserve(e-b+1); r0[u].push_back(0);
    for (iter it = b; it != e; ++it)
      r0[u].push_back(r0[u].back() + (*it<=m));
    iter p = stable_partition(b, e, [=](int i){
                              return i<=m;});
    build(b, p, l, m, u*2);
    build(p, e, m+1, r, u*2+1);
  }

  int q, w;
  int range(int a, int b, int l, int r, int u) {
    if (r < q or w < l)
      return 0;
    if (q <= l and r <= w)
      return b-a;
    int m = (l+r)/2, za = r0[u][a], zb = r0[u][b];
    return range(za, zb, l, m, u*2) +
      range(a-za, b-zb, m+1, r, u*2+1);
  }

  // arr[i] in [0,sigma)
  WaveTree(vector<int> arr, int sigma) {
    n = arr.size(); s = sigma;
    r0.resize(s*2); arrCopy = arr;
    build(arr.begin(), arr.end(), 0, s-1, 1);
  }

  // k in [1,n], [a,b) is 0-indexed, -1 if error
  int quantile(int k, int a, int b) {
    //extra conditions disabled
    if (/*a < 0 or b > n or*/ k < 1 or k > b-a)
      return -1;
    int l = 0, r = s-1, u = 1, m, za, zb;
    while (l != r) {
      m = (l+r)/2;
      za = r0[u][a]; zb = r0[u][b]; u*=2;
      if (k <= zb-za)
        a = za, b = zb, r = m;
      else
        k -= zb-za, a -= za, b -= zb,
        l = m+1, ++u;
    }
    return r;
  }

  // counts numbers in [x,y] in positions [a,b)
  int range(int x, int y, int a, int b) {
    if (y < x or b <= a)
      return 0;
    q = x; w = y;
    return range(a, b, 0, s-1, 1);
  }

  // count occurrences of x in positions [0,k)
  int rank(int x, int k) {
    int l = 0, r = s-1, u = 1, m, z;
    while (l != r) {
      m = (l+r)/2;
      z = r0[u][k]; u*=2;
      if (x <= m)
        k = z, r = m;
      else
        k -= z, l = m+1, ++u;
    }
    return k;
  }

  // x in [0,sigma)
  void push_back(int x) {
    int l = 0, r = s-1, u = 1, m, p; ++n;
    while (l != r) {
      m = (l+r)/2;
      p = (x<=m);
      r0[u].push_back(r0[u].back() + p);
      u*=2; if (p) r = m; else l = m+1, ++u;
    }
  }

  // doesn't check if empty
  void pop_back() {
    int l = 0, r = s-1, u = 1, m, p, k; --n;
    while (l != r) {
      m = (l+r)/2; k = r0[u].size();
      p = r0[u][k-1] - r0[u][k-2];
      r0[u].pop_back();
      u*=2; if (p) r = m; else l = m+1, ++u;
    }
  }

  //swap arr[i] with arr[i+1], i in [0,n-1)
  void swap_adj(int i) {
    int &x = arrCopy[i], &y = arrCopy[i+1];
    int l = 0, r = s-1, u = 1;
    while (l != r) {
      int m = (l+r)/2, p = (x<=m), q = (y<=m);
      if (p != q) {
```

```
        r0[u][i+1] ^= r0[u][i] ^ r0[u][i+2];
        break;
      }
      u*=2; if (p) r = m; else l = m+1, ++u;
    }
    swap(x, y);
  }
};
```

2.6. Union-Find

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;

struct UnionFind {
    vi p, rank, setSize;
    int numSets;
    UnionFind(int n) {
        numSets = n; setSize.assign(n, 1); rank.assign(n, 0); p.resize(n);
        rep(i,0,n-1) p[i] = i;
    }
    int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
    bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
    void unionSet(int i, int j) {
        if (!isSameSet(i, j)) {
            numSets--;
            int x = findSet(i), y = findSet(j);
            // rank is used to keep the tree short
            if (rank[x] > rank[y]) {
                p[y] = x; setSize[x] += setSize[y];
            } else {
                p[x] = y; setSize[y] += setSize[x];
                if (rank[x] == rank[y]) rank[y]++;
            }
        }
    }
    int numDisjointSets() { return numSets; }
    int sizeOfSet(int i) { return setSize[findSet(i)]; }
};
```

3. General Algorithms

3.1. Binary Search

```
// Find the index of the first item that satisfies a predicate
// over a range [i,j), i.e., from i to j-1
// If no such index exists, j is returned
function binsearch(array, i, j) {
    assert(i < j) // since the range is [i,j), then j must be > i
    while (i < j) {
        m = (i+j) >> 1;
        if (predicate(array[m]))
            j = m
        else
            i = m + 1
    }
    return i; // notice that i == j if the predicate is false for the whole range
}

// ----------------------------
// EXAMPLE 1: Integer Lowerbound
// predicate(a, i, key) = (a[i] >= key)
// i.e. "first element >= key"
int lowerbound(vector<int> a, int key, int i, int j) {
    while (i < j) {
        int m = (i + j) / 2;
        if (a[m] >= key)
            j = m;
        else
            i = m + 1;
    }
    return i;
}

// ----------------------------
// EXAMPLE 2: Integer Upperbound
// predicate(a, i, key) = (a[i] > key)
// i.e. "first element > key"
int upperbound(vector<int> a, int key, int i, int j) {
    while (i < j) {
        int m = (i + j) / 2;
        if (a[m] > key)
            j = m;
        else
            i = m + 1;
    }
    return i
}

/* ============================ */
/*  upper_bound(), lower_bound() */
/* ============================ */

// search between [first, last)
// if no value is >= key (lb) / > key (ub), return last

#include <algorithm>
#include <iostream>     // std::cout
#include <algorithm>    // std::lower_bound, std::upper_bound, std::sort
#include <vector>       // std::vector

int main () {
    int myints[] = {10,20,30,30,20,10,10,20};
    std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20

```

```
    std::sort (v.begin(), v.end());               // 10 10 10 20 20 20 30 30

    std::vector<int>::iterator low,up;
    low=std::lower_bound (v.begin(), v.end(), 20); //          ^
    up= std::upper_bound (v.begin(), v.end(), 20); //                    ^

    std::cout << "lower_bound at position " << (low- v.begin()) << '\n';
    std::cout << "upper_bound at position " << (up - v.begin()) << '\n';

    return 0;
}

// ----------------------------------------------
// Query: how many items are LESS THAN (<) value x

lower_bound(v.begin(), v.end(), x) - v.begin();

// ----------------------------------------------
// Query: how many items are GREATER THAN (>) value x

v.end() - upper_bound(v.begin(), v.end(), x);

//=================
// binary_search()
//=================
bool myfunction (int i,int j) { return (i<j); }
int myints[] = {1,2,3,4,5,4,3,2,1};
std::vector<int> v(myints,myints+9);
bool found = std::binary_search (v.begin(), v.end(), 6, myfunction)


/* ====================== */
/* Discrete Ternary Search */
/* ====================== */

int min_search(int i, int j) {
    while (i < j) {
        int m = (i+j)/2;
        int slope = eval(m+1) - eval(m);
        if (slope >= 0)
            j = m;
        else
            i = m+1;
    }
    return i;
}

int max_search(int i, int j) {
    while (i < j) {
        int m = (i+j)/2;
        int slope = eval(m+1) - eval(m);
        if (slope <= 0)
            j = m;
        else
            i = m+1;
    }
    return i;
}
```

3.2. Ternary Search

```
int times = 100;
double left = 0.0;
double right = 1000.0;
double ans, m1, m2, v1, v2, third;

while (times--) {
  third = (right - left) / 3.0;
  m1 = left + third;
  m2 = right - third;
  v1 = eval(m1);
  v2 = eval(m2);
  if (v1 < v2)
    left = m1;
  else if(v2  < v1)
    right = m2;
  else
    left = m1, right = m2;
}

ans = (v1 + v2) * 0.5;
```

3.3. Brute Force

3.3.1. Generate all combinations

```
/* ============================== */
/*  Try all 2^n subsets of n items */
/* ============================== */
void all_subsets(vector<int> items) {
    int n = vals.size();
    int times = (1 << n);
    vector<int> bits(n, 0)
    while(times-- > 0) {
        do_something(bits)
        // generate next set's bit representation
        int i = 0, carry = 1;
        while (i < n) {
            in[i] += carry;
            if (in[i] <= 1)
                carry = 0;
            else
                in[i] = 0;
            i++;
        }
    }
}
```

```
22
23  /* ======================================== */
24  /* Split n items into k containers optimally */
25  /* ======================================== */
26  int capacities[MAXN];
27  int N;
28  // Return cost of storing n items in i-th container
29  storage_cost(int i, int n);
30  // Find best way to split n items among containers
31  // from index i to N-1. For simplicity, the total
32  // remaining capacity is carried along.
33  int search_splits(int i, int n, int tot_cap) {
34      if (i >= N) return 0;
35      int min_k = max(0, n - (tot_cap - capacities[i]));
36      int max_k = min(n, capacities[i]);
37      int min_cost = INT_MAX;
38      rep(k, min_k, max_k) {
39          min_cost = min(min_cost,
40              storage_cost(i, k) +
41              search_splits(i+1, n-k, tot_cap - capacities[i]);
42          )
43
44      }
45  }
46  int best_split(int n) {
47      int tot_cap = 0;
48      rep(i,0,N-1) tot_cap += capacities[i];
49      return search_splits(0,n,tot_cap);
50  }
```

4. Dynamic Programming

4.1. Knapsack

```
1   /* ==================== */
2   /* Knapsack problem : DP */
3   /* ==================== */
4
5   // -------------------------------------
6   // VARIANT 1: without reposition of items
7   // -------------------------------------
8
9   // ---------------------------------
10  // TOP-DOWN RECURSION (pseudo-code)
11
12  function DP(i, c)
13    if i == first
14      if c >= weight[i] && value[i] > 0 // enough space and worth it
15        return value[i]
16      else
17        return 0
18    else
19      ans = DP(i-1, c)
20      if c >= weight[i] && value[i] > 0 // enough space and worth it
21        ans = max(ans, value[i] + DP(i-1, c - weight[i]))
22      return ans
23
24  // -----------
25  // BOTTOM-UP
26
27  #define MAXN 1000 // max num items
28  #define MAXC 500 // max capacity
29  int value[MAXN];
30  int weight[MAXN];
31  int memo[MAXC+1]; // 0 ... MAXC
32  int N, C;
33
34  int dp() {
35    // first item (i = 0)
36    memset(memo, 0, sizeof(memo[0]) * (C+1));
37    if (value[0] > 0) { // worth it
38      rep (c, weight[0], C) {
39        memo[c] = value[0];
40      }
41    }
42    // other items (i = 1 .. N-1)
43    rep (i, 1, N-1) {
44      if (value[i] > 0) { // worth it
45        invrep(c, C, weight[i]) { // <--- REVERSE ORDER !!
46          memo[c] = max(memo[c], value[i] + memo[c - weight[i]]);
47        }
48      }
49    }
50    return memo[C];
51  }
52
53  // -------------------------------------
54  // VARIANT 2: with reposition of items
55  // -------------------------------------
56
57  // ---------------------------------
58  // TOP-DOWN RECURSION (pseudo-code)
59
60  function DP(i, c)
61    if i == first
62      if c >= weight[i] && value[i] > 0 // enough space and worth it
63        return value[i]
64      else
65        return 0
66    else
67      ans = DP(i-1, c)
68      if c >= weight[i] && value[i] > 0 // enough space and worth it
69        ans = max(ans, value[i] + DP(i, c - weight[i])) // << i instead of i-1
70      return ans
71
72  // -----------
73  // BOTTOM-UP
```

```

#define MAXN 1000 // max num items
#define MAXC 500 // max capacity
int value[MAXN];
int weight[MAXN];
int memo[2][MAXC + 1]; // 0 .. MAXC
int N, C;

int dp() {
  // first item (i = 0)
  memset(memo, 0, sizeof(memo[0]) * (C+1));
  if (value[0] > 0) { // worth it
    rep (c, weight[0], C) {
      memo[0][c] = value[0] * (c / weight[0]); // collect it as many times as you can
    }
  }
  // other items (i = 1 .. N-1)
  int prev = 0, curr = 1;
  rep (i, 1, N-1) {
    rep(c, 0, C) { // <--- INCREASING ORDER !!
      if (c >= weight[i] && value[i] > 0) { // if fits in && worth it
        memo[curr][c] = max(
          memo[prev][c], // option 1: don't take it
          value[i] + memo[curr][c - weight[i]] // option 2: take it
        );
      } else {
        memo[curr][c] = memo[prev][c]; // only option is to skip it
      }
    }
    // update prev, curr
    prev = curr;
    curr = 1-curr;
  }
  return memo[(N-1)&1][C]; // last item + full capacity
}
```

4.2. Divide & Conquer Optimization

```
#include <bits/stdc++.h>
using namespace std;
#define rep(i,a,b) for(int i=a;i<=b;++i)
typedef long long int ll;

#define MAXG 1000
#define MAXL 1000
int G,L;
ll DP[MAXG+1][MAXL+1];

// return cost of forming a group with items in the range i .. j
ll group_cost(int i, int j) { ... }

/**
 Calculates the values of DP[g][l] for l1 <= l <= l2 (a range of cells in row 'g')
 using divide & conquer optimization

 DP[g][l] means: given a list of the first 'l' items, partition them into 'g' groups,
 each group consisting of consecutive items (left to right), so that the total
 cost of forming those groups is the minimum possible.

 If we form one group at a time, from right to left, this leads to the following
 recursion:

 DP[g][l] = min { DP[g-1][k] + group_cost(k,l-1) for k = g-1 .. l-1 }
 DP[1][l] = group_cost(0, l-1)

 in other words:

 DP[g][l] = DP[g-1][best_k] + group_cost(best_k,l-1)
    where best_k is the left most value of k where the minimum is reached

Now, for a given 'g':

    If best_k(g,0) <= best_k(g,1) <= best_k(g,2) <= ... <= best_k(g,L-1) holds

    Then, we can propagate those best_k's recursively to reduce the range of
    candidate k's for each DP[g][l] problem we solve.
    Using Divide & Conquer, we fill the whole row 'g' recursively with
    recursion depth O(log(L)), and each recursion layer taking O(L) time.

Doing this for G groups, the total computation cost is O(G*L*log(L))

*/
void fill_row(int g, int l1, int l2, int k1, int k2) {
    if (l1 > l2) return; // ensure valid range
    int lm = (l1+l2)/2; // solve middle case
    int kmin = max(g-1, k1);
    int kmax = min(lm-1, k2);
    int best_k = -1;
    ll mincost = LLONG_MAX;
    rep(k,kmin,kmax) {
        ll tmp = DP[g-1][k] + group_cost(k, lm-1);
        if (mincost > tmp) mincost = tmp, best_k = k;
    }
    DP[g][lm] = mincost;
    fill_row(g, l1, lm-1, k1, best_k); // solve left cases
    fill_row(g, lm+1, l2, best_k, k2); // solve right cases
}

void fill_dp() {
    // base: g = 1
    rep(1,1,L) DP[1][l] = group_cost(0,l-1);
    // other: g >= 2
    rep(g,2,G) fill_row(g,g,L,0,L);
}
```

5. Graphs

5.1. BFS

```
#define MAXN 1000
vector<vi> g; // graph
vi depth; // bfs depth per node
int n; // num of nodes

void bfs(int s) {
  queue<int> q; q.push(s);
  depth.assign(n,-1);
  depth[s] = 0;
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int v : g[u]) {
      if (depth[v] == -1) {
        depth[v] = depth[u] + 1;
        q.push(v);
      }
    }
  }
}
```

5.2. DFS

```
// ========================
// Depth First Search (DFS)
// ========================
#define MAXN 1000
vector<vi> g[MAXN];
bool visited[MAXN];
int n;

//iterative
void dfs(int root) {
  stack<int> s;
  s.push(root);
  visited[root] = true;
  while (!s.empty()) {
    int u = s.top(); s.pop();
    for (int v : g[u])
      if (!visited[v])
        visited[u] = true, s.push(v);
  }
}

//recursive
void dfs(int u) {
  visited[u] = true;
  for(int v : g[u])
    if(!visited[v])
      dfs(v);
}

//------------------------------
// Finding connected components
//------------------------------
int count_cc() {
  int count = 0;
  memset(visited,0,sizeof(bool)*n);
  rep(i,0,n-1)
    if (!visited[i])
      count++, dfs(i);
  return count;
}

//------------------------------
// Flood Fill
//------------------------------

//explicit graph
#define DFS_WHITE (-1)
vector<int> dfs_num(DFS_WHITE,n);
void floodfill(int u, int color) {
  dfs_num[u] = color;
  for (int v : g[u])
    if (dfs_num[v] == DFS_WHITE)
      floodfill(v, color);
}

//implicit graph
int dirs[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
const char EMPTY = '*';
int floodfill(int r, int c, char color) {
  if (r < 0 || r >= R || c < 0 || c >= C) return 0; // outside grid
  if (grid[r][c] != EMPTY) return 0; // cannot be colored
  grid[r][c] = color;
  int ans = 1;
  rep(i,0,3) ans += floodfill(r + dirs[i][0], c + dirs[i][1], color);
  return ans;
}
```

5.3. Dijkstra

```
// complexity: (|E| + |V|) * log |V|
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii; // (weight, node), in that order

vector<vector<pii>> g; // graph
int N; // number of nodes
vector<int> mindist; // min distance from source to each node
vector<int> parent; // parent of each node in shortest path from source

void dijkstra(int source) {
    parent.assign(N, -1);
```

```
    mindist.assign(N, INT_MAX);
    mindist[source] = 0;
    priority_queue<pii, vector<pii>, greater<pii>> q;
    q.push(pii(0, source));
    while (!q.empty()) {
        pii p = q.top(); q.pop();
        int u = p.second, dist = p.first;
        if (mindist[u] < dist) continue; // skip outdated improvements
        for (pii& e : g[u]) {
            int v = e.second, w = e.first;
            if (mindist[v] > dist + w) {
                mindist[v] = dist + w;
                parent[v] = u;
                q.push(v);
            }
        }
    }
}
```

5.4. Max Flow : Dinic

```
// Time Complexity:
// - general worst case: O (|E| * |V|^2)
// - unit capacities: O( min( V^(2/3), sqrt(E) ) )
// - Bipartite graph (unit capacities) + source & sink (any capacities): O(E sqrt V)

#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;

struct Dinic {
    struct edge {
        int to, rev;
        ll f, cap;
    };

    vector<vector<edge>> g;
    vector<ll> dist;
    vector<int> q, work;
    int n, sink;

    bool bfs(int start, int finish) {
        dist.assign(n, -1);
        dist[start] = 0;
        int head = 0, tail = 0;
        q[tail++] = start;
        while (head < tail) {
            int u = q[head++];
            for (const edge &e : g[u]) {
                int v = e.to;
                if (dist[v] == -1 and e.f < e.cap) {
                    dist[v] = dist[u] + 1;
                    q[tail++] = v;
                }
            }
        }
        return dist[finish] != -1;
    }

    ll dfs(int u, ll f) {
        if (u == sink)
            return f;
        for (int &i = work[u]; i < (int)g[u].size(); ++i) {
            edge &e = g[u][i];
            int v = e.to;
            if (e.cap <= e.f or dist[v] != dist[u] + 1)
                continue;
            ll df = dfs(v, min(f, e.cap - e.f));
            if (df > 0) {
                e.f += df;
                g[v][e.rev].f -= df;
                return df;
            }
        }
        return 0;
    }

    Dinic(int n) {
        this->n = n;
        g.resize(n);
        dist.resize(n);
        q.resize(n);
    }

    void add_edge(int u, int v, ll cap) {
        edge a = {v, (int)g[v].size(), 0, cap};
        edge b = {u, (int)g[u].size(), 0, 0}; //Poner cap en vez de 0 si la arista es
            bidireccional
        g[u].push_back(a);
        g[v].push_back(b);
    }

    ll max_flow(int source, int dest) {
        sink = dest;
        ll ans = 0;
        while (bfs(source, dest)) {
            work.assign(n, 0);
            while (ll delta = dfs(source, LLONG_MAX))
                ans += delta;
        }
        return ans;
    }
};

// usage
int main() {
    Dinic din(2);
    din.add_edge(0,1,10);
```

```
    ll mf = din.max_flow(0,1);
}
```

5.5. Minimum Spanning Tree (Kruskal & Prim)

```
#include <bits/stdc++.h>
using namespace std;
vector<int> vi;
typedef pair<int,int> ii;

/* ================= */
/* METHOD 1: KRUSKAL */
/* ================= */

struct Edge {
    int u, int v, int cost;
    bool operator<(const Edge& o) const {
        return cost < o.cost;
    }
};
namespace Kruskal {
    struct UnionFind {
        vi p, rank;
        int numSets;
        UnionFind(int n) {
            numSets = n;
            rank.assign(n,0);
            p.resize(n);
            rep(i,0,n-1) p[i] = i;
        }
        int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
        bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
        void unionSet(int i, int j) {
            if (!isSameSet(i, j)) {
                numSets--;
                int x = findSet(i), y = findSet(j);
                if (rank[x] > rank[y]) {
                    p[y] = x;
                } else {
                    p[x] = y;
                    if (rank[x] == rank[y]) rank[y]++;
                }
            }
        }
    };

    int find_mst(int n_nodes, vector<Edge>& edges, vector<vector<ii>>& mst) {
        sort(edges.begin(), edges.end());
        UnionFind uf(n_nodes);
        mst.assign(n_nodes, vector<ii>());
        int mstcost = 0;
        int count = 1;
        for (auto& e : edges) {
            int u = e.u, v = e.v, cost = e.cost;
            if (!uf.isSameSet(u, v)) {
                mstcost += cost;
                uf.unionSet(u, v);
                mst[u].emplace_back(v, cost);
                mst[v].emplace_back(u, cost);
                if (++count == n_nodes) break;
            }
        }
        return mstcost;
    }
}

/* ============== */
/* METHOD 2: PRIM */
/* ============== */

struct Edge {
    int u, v, cost;
    bool operator<(const Edge& o) const {
        return cost > o.cost; // we use '>' instead of '<' so that
        // priority_queue<Edge> works as a minheap
    }
};
namespace Prim {
    bool visited[MAXN]
    int find_mst(vector<vector<ii>>& g, vector<vector<ii>>& mst) {
        int n_nodes = g.size();
        memset(visited, false, sizeof(bool) * n_nodes);
        mst.assign(n_nodes, vector<ii>());
        priority_queue<Edge> q;
        int total_cost = 0;
        visited[0] = true;
        for (ii& p : g[0]) q.push({0, p.first, p.second});
        int count = 1;
        while (!q.empty()) {
            Edge edge = q.top(); q.pop();
            if (visited[edge.v]) continue;
            int u = edge.u;
            int v = edge.v;
            int cost = edge.cost;
            visited[v] = true;
            total_cost += cost;
            mst[u].emplace_back(v, cost);
            mst[v].emplace_back(u, cost);
            if (++count == N) break;
            for (ii p : g[v]) {
                if (visited[p.first]) continue;
                q.push({v, p.first, p.second});
            }
        }
        return total_cost;
    }
}
```

5.6. LCA (Lowest Common Ancestor)

```cpp
/* ========================== */
/* LCA (Lowest Common Ancestor) */
/* ========================== */
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
#define rep(i,a,b) for (int i=a; i<=b; ++i)
#define invrep(i,b,a) for (int i=b; i>=a; --i)

// General comments:
// * Both of these methods assume that we are working with a connected
//   graph 'g' of 'n' nodes, and that nodes are compactly indexed from 0 to n-1.
//   In case you have a forest of trees, a simple trick is to create a fake
//   root and connect all the trees to it (make sure to re-index all your nodes)
// * 'g' need not be a 'tree', DFS fill implictly find a tree for you
//   in case you don't care of the specific tree (e.g. if cycles are not important)

// ------------------------------------------------------------
// METHOD 1: SPARSE TABLE - BINARY LIFTING (aka JUMP POINTERS)
// ------------------------------------------------------------
// construction: O(|V| log |V|)
// query: O(log|V|)
// ** advantages:
//   - it's possible to append new leaf nodes to the tree
//   - the lca query can be modified to compute querys over the path between 2 nodes

namespace LCA1 {
    const int MAXN = 1000000;
    const int MAXLOG = 31 - __builtin_clz(MAXN);
    // const int MAXLOG = sizeof(int) * 8 - __builtin_clz(MAXN)-1;

    int P[MAXN][MAXLOG+1]; // level ancestor table
    int D[MAXN]; // depths
    int n; // num of nodes
    vector<vi> *g; // pointer to graph
    int root; // root of the tree

    // get highest exponent e such that 2^e <= x
    inline int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }

    // dfs to record direct parents and depths
    void dfs(int u, int p, int depth) {
        P[u][0] = p;
        D[u] = depth;
        for (int v : (*g)[u]) {
            if (D[v] == -1) {
                dfs(v, u, depth + 1);
            }
        }
    }

    void init(vector<vi> &_g, int _root) {
        g = &_g;
        root = _root;
        n = _g.size();
        memset(D, -1, sizeof(int) * n);
        dfs(root, -1, 0);
        rep(j, 1, MAXLOG) {
            rep (i, 0, n-1) {
                // i's 2^j th ancestor is
                // i's 2^(j-1) th ancestor's 2^(j-1) th ancestor
                int p = P[i][j-1];
                P[i][j] = (p == -1 ? -1 : P[p][j-1]);
            }
        }
    }

    int raise(int u, int dist) {
        // move node u "dist" steps up towards the root
        for (int i = 0; dist; i++, dist>>=1) if (dist&1) u = P[u][i];
        return u;
    }

    int find_lca(int u, int v) {
        if (D[u] < D[v]) swap(u, v);
        u = raise(u, D[u] - D[v]); // raise lowest to same level
        if (u == v) return u; // same node, we are done
        // raise u and v to their highest ancestors below the LCA
        invrep (j, MAXLOG, 0) {
            // greedily takes the biggest 2^j jump possible as long as
            // u and v still remain BELOW the LCA
            if (P[u][j] != P[v][j]) {
                u = P[u][j], v = P[v][j];
            }
        }
        // the direct parent of u (or v) is lca(u,v)
        return P[u][0];
    }

    int dist(int u, int v) {
        return D[u] + D[v] - 2 * D[find_lca(u,v)];
    }

    int add_child(int p, int u) {
        // add to graph
        (*g)[p].push_back(u);
        // update depth
        D[u] = D[p] + 1;
        // update ancestors
        P[u][0] = p;
        rep (j, 1, MAXLOG){
            p = P[p][j-1];
            if (p == -1) break;
            P[u][j] = p;
        }
    }
```

```
}

// ----------------------------------------
// METHOD 2: SPARSE TABLE - EULER TOUR + RMQ
// ----------------------------------------
// construction: O(2|V| log 2|V|) = O(|V| log |V|)
// query: O(1) (** assuming that __builtin_clz is mapped to an
//               efficient processor instruction)

namespace LCA2 {
    const int MAXN = 10000;
    const int MAXLOG = 31 - __builtin_clz(MAXN);
    // const int MAXLOG = sizeof(int) * 8 - __builtin_clz(MAXN)-1;

    int E[2 * MAXN]; // records sequence of visited nodes
    int D[2 * MAXN]; // records depth of each visited node
    int H[MAXN]; // records index of first ocurrence of node u in E
    int idx; // tracks node ocurrences
    int rmq[2 * MAXN][MAXLOG+1]; // memo table for range minimun query
    vector<vi> *g; // pointer to graph
    int n; // number of nodes
    int root; // root of the tree

    // get highest exponent e such that 2^e <= x
    inline int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }

    void dfs(int u, int depth) {
        H[u] = idx; // index of first u's ocurrence
        E[idx] = u; // record node ocurrence
        D[idx++] = depth; // record depth
        for (int v : (*g)[u]) {
            if (H[v] == -1) {
                dfs(v, depth + 1); // explore v's subtree and come back to u
                E[idx] = u; // new ocurrence of u
                D[idx++] = depth;
            }
        }
    }

    void init(vector<vi> &_g, int _root) {
        g = &_g;
        root = _root;
        n = _g.size();
        memset(H, -1, sizeof(int)*n);
        idx = 0;
        dfs(root, 0); // euler tour
        int nn = idx; // <-- make sure you use the correct number
        int m = log2(nn);
        // build sparse table with bottom-up DP
        rep(i,0,nn-1) rmq[i][0] = i; // base case
        rep(j,1,m) { // other cases
            rep(i, 0, nn - (1 << j)) {
                // i ... i + 2 ^ (j-1) - 1
                int i1 = rmq[i][j-1];
                // i + 2 ^ (j-1) ... i + 2 ^ j  - 1
                int i2 = rmq[i + (1 << (j-1))][j-1];
                // choose index with minimum depth
                rmq[i][j] = D[i1] < D[i2] ? i1 : i2;
            }
        }
    }

    int find_lca(int u, int v) {
        // get ocurrence indexes in increasing order
        int l = H[u], r = H[v];
        if (l > r) swap(l, r);
        // get node with minimum depth in range [l .. r] in O(1)
        int len = r - l + 1;
        int m = log2(len);
        int i1 = rmq[l][m];
        int i2 = rmq[r - ((1 << m) - 1)][m];
        return D[i1] < D[i2] ? E[i1] : E[i2];
    }

    int dist(int u, int v) {
        // make sure you use H to retrieve the indexes of u and v
        // within the Euler Tour sequence before using D
        return D[H[u]] + D[H[v]] - 2 * D[H[find_lca(u,v)]];
    }
}

// -----------------
// EXAMPLE OF USAGE
// -----------------
int main() {
    // build graph
    int n, m;
    scanf("%d", &n, &m);
    vector<vi> g(n);
    while (m--) {
        int u, v; scanf("%d%d", &u, &v);
        g[u].push_back(v);
        g[v].push_back(u);
    }
    // init LCA
    LCA1::init(g, 0);
    // answer queries
    int q; scanf("%d", &q);
    while (q--) {
        int u, v; scanf("%d%d", &u, &v);
        printf("LCA(%d, %d) = %d\n", u, v, LCA1::find_lca(u,v));
        printf("dist(%d, %d) = %d\n", u, v, LCA1::dist(u,v));
    }
}
```

5.7. Diameter of a Tree

```
// ========================
```

```cpp
// Find Tree's Diameter Ends
// =========================
const int MAXN = 10000;

int farthest_from(vector<vi>& g, int s) { // find farthest node from 's' with BFS
    static int dist[MAXN];
    memset(dist, -1, sizeof(int) * g.size());
    int farthest = s;
    queue<int> q;
    q.push(s);
    dist[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : g[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
                if (dist[v] > dist[farthest]) farthest = v;
            }
        }
    }
    return farthest;
}

void find_diameter(vector<vi>& g, int& e1, int& e2) {
    e1 = farthest_from(g, 0);
    e2 = farthest_from(g, e1);
}
```

5.8. Articulation Points, Cut Edges, Biconnected Components

```cpp
//sources:
//https://www.youtube.com/watch?v=jFZsDDB0-vo
//https://www.hackerearth.com/practice/algorithms/graphs/articulation-points-and-bridges/tutorial/
//https://www.hackerearth.com/practice/algorithms/graphs/biconnected-components/tutorial/
//http://web.iitd.ac.in/~bspanda/biconnectedMTL776.pdf
typedef pair<int,int> ii;
const int MAXN = 1000;
int depth[MAXN];
int low[MAXN];
vector<int> g[MAXN];
stack<ii> edge_stack;

void print_and_remove_bicomp(int u, int v) {
    puts("biconnected component found:");
    ii uv(u,v);
    while (true) {
        ii top = edge_stack.top();
        edge_stack.pop();
        printf("(%d, %d)\n", top.first, top.second);
        if (top == uv) break;
    }
}

void dfs(int u, int p, int d) { // (node, parent, depth)
    static num_root_children = 0;
    depth[u] = d;
    low[u] = d; // u at least can reach itself (ignoring u-p edge)
    for(int v : g[u]) {
        if (v == p) continue; // direct edge to parent -> ignore
        if (depth[v] == -1) { // exploring a new, unvisited child node
            edge_stack.emplace(u,v); // add edge to stack
            dfs(v, u, d + 1); // explore recursively v's subtree
            // 1) detect articulation points and biconnected components
            if (p == -1) { // 1.1) special case: if u is root
                if (++num_root_children == 2) {
                    // we detected that root has AT LEAST 2 children
                    // therefore root is an articulation point
                    printf("root = %d is articulation point\n", root);
                }
                // whenever we come back to the root, we just finished
                // exploring a whole biconnected component
                print_and_remove_bicomp(u,v);
            } else if (low[v] >= d) { // 1.2) general case: non-root
                printf("u = %d is articulation point\n",  u);
                // we entered through and came back to an AP,
                // so we just finished exploring a whole biconnected component
                print_and_remove_bicomp(u,v);
            }
            // 2) detect cut edges (a.k.a. bridges)
            if (low[v] > depth[u]) {
                printf("(u,v) = (%d, %d) is cut edge\n", u, v);
            }
            // propagate low
            low[u] = min(low[u], low[v]);
        } else if (depth[v] < d) { // back-edge to proper ancestor
            edge_stack.emplace(u,v); // add edge to stack
            low[u] = min(low[u], depth[v]); // propagate low
        } else { // forward-edge to an already visited descendant
            // => do nothing, because this edge was already considered as a
            // back-edge from v -> u
        }
    }
}
```

5.9. Centroid Decomposition

```cpp
/* ====================== */
/* Centroid Decomposition */
/* ====================== */

// construction: O(n log n)
// query: O(log n)

#include <vector>
#include <queue>
#include <cstring>
```

```cpp
using namespace std;

#define MAXN 100000
typedef vector<int> vi;

vector<vi> g; // graph
vector<vi> cg; // centroid graph
int N; // num of nodes
bool removed[MAXN]; // nodes removed from tree
int desc[MAXN]; // num of descendants
int cpar[MAXN]; // centroid parent

// count descendants
int dfs_count(int u, int p) {
  int count = 1;
  for (int v : g[u])
    if (v != p && !removed[v])
      count += dfs_count(v, u);
  return desc[u] = count;
}

// recursive search of centroid
int dfs_cent(int u, int p, int lim) {
  for (int v : g[u])
    if (v != p && !removed[v] && desc[v] > lim)
      return dfs_cent(v, u, lim);
  return u;
}

// find centroid of u's subtree
int centroid(int u) {
  dfs_count(u, -1);
  return dfs_cent(u, -1, desc[u] / 2);
}

// perform centroid decomposition
void decomp() {
  memset(removed, 0, sizeof(removed[0]) * N);
  cg.assign(N, vi());
  int c = centroid(0);
  cpar[c] = -1;
  removed[c] = true;
  queue<int> q; q.push(c);
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int v : g[u]) {
      if (!removed[v]) {
        c = centroid(v);
        cpar[c] = u; // set parent of c to u
        cg[u].push_back(c); // add edge (u -> c)
        removed[c] = true;
        q.push(c);
      }
    }
  }
}
```

6. Mathematics

6.1. Euclidean Algorithm

```cpp
/* ============================ */
/* GCD (greatest common divisor) */
/* ============================ */
// euclid's algorithm
int gcd (int a, int b) {
    int tmp;
    while (b) tmp = a, a = b, b = tmp % b;
    return a;
}

/* ============ */
/* extended GCD */
/* ============ */
// extended euclid's algorithm
// a * x + b * y = d = gcd(a, b)
// x = x0 + n * (b/d)
// y = y0 - n * (a/d)
// where n is integer

// recursive
void xgcd(int a, int b, int& g, int& x, int& y) {
    if (b == 0) { x = 1; y = 0; g = a; return; }
    xgcd(b, a % b, g, x, y);
    int x1 = y, y1 = x - y * (a / b);
    x = x1, y = y1;
}

// iterative
void xgcd(int a, int b, int& g, int& x, int& y)
{
    int x0 = 1, x1 = 0, y0 = 0, y1 = 1;
    int q, r, tmp;
    while (b) {
        q = a / b, r = a % b;
        tmp = x1, x1 = x0 - q * x1, x0 = tmp;
        tmp = y1, y1 = y0 - q * y1, y0 = tmp;
        a = b, b = r;
    }
    g = a, x = x0, y = y0;
}

/* ==================== */
/* multiplicative inverse */
/* ==================== */
int mulinv(int a, int m) {
    int g, x, y; xgcd(a, m, g, x, y);
```

```
    if (g == 1) return x % m;
    return -1;
}
```

6.2. Prime Numbers

```
#define MAXN 9999
#define umap unordered_map
#define rep(i,a,b) for(int i=a;i<=b;++i)

//==========================================
// Sieve of Eratosthenes (all primes up to N)
//==========================================
vector<int> get_primes_up_to(int n) {
  vector<bool> is_prime(n + 1, true);
  int limit = (int) floor(sqrt(n));
  rep (i,2,limit)
    if (is_prime[i])
      for (int j = i * i; j <= n; j += i)
        is_prime[j] = false;
  vector<int> primes;
  rep(i,2,n) if (is_prime[i]) primes.push_back(i);
  return primes;
}

//==================================
// Prime Factorization of Factorials
//==================================
// source: http://mathforum.org/library/drmath/view/67291.html
void factorial_prime_factorization(int x, umap<int,int>& counts) {
  static vector<int> primes = get_primes(MAXN);

  for (int p : primes) {
    if (p > x) break;
    int count = 0;
    int n = x;
    while ((n /= p) > 0)
      count += n;
    if (count) counts[p] = count;
  }
}
```

6.3. Modular Binomial Coefficient

```
#define rep(i,a,b) for(int i = a; i <= b; ++i)
typedef long long int ll;
const ll MOD = 100000000711; // a prime number
const int MAXN = 1000;

/* ================== */
/*  MODULAR BINOMIAL  */
/* ================== */
// choose_mod(n,k) = n! / (k! * (n-k)!) % MOD

// --------------------
// method 1: DP
// choose(n,k) = (choose(n-1,k-1) + choose(n-1,k)) % MOD
// choose(n,0) = choose(n,n) = 1

// 1.1) DP top-down
ll memo[MAXN+1][MAXN+1];
ll choose(int n, int k) {
    ll& ans = memo[n][k];
    if (ans != -1) return ans;
    if (k == 0) return ans = 1;
    if (n == k) return ans = 1;
    if (n < k) return ans = 0;
    return ans = (choose(n-1,k) + choose(n-1,k-1)) % MOD;
}

// 1.2) DP bottom-up
ll choose[MAXN+1][MAXN+1];
rep(m,1,MAXN) {
    choose[m][0] = choose[m][m] = 1;
    rep(k,1,m-1) choose[m][k] = (choose[m-1][k] + choose[m-1][k-1]) % MOD;
}

// ------------------------------------------------
// method 3: factorials and multiplicative inverse
// n! / (k! * (n-k)!) =  n! * (k! * (n-k)!)^-1  (MOD N)
// we need to find the multiplicative inverse of (k! * (n-k)!) MOD N


// --- multiplicative inverse --
void xgcd(ll a, ll b, ll&g, ll&x, ll& y) {
    ll x0 = 1, x1 = 0, y0 = 0,  y1 = 1;
    ll q, r, tmp;
    while (b) {
        q = a / b, r = a % b;
        tmp = x1, x1 = x0 - q*x1, x0 = tmp;
        tmp = y1, y1 = y0 - q*y1, y0 = tmp;
        a=b,b=r;
    }
    g=a, x=x0, y=y0;
}
ll multinv(ll a, ll mod) {
    ll g,x,y; xgcd(a,mod,g,x,y);
    if (g == 1) return (x+mod) % mod;
    return -1;
}

// -- choose_mod(n, k) --
ll fac[MAXN+1];
ll choose_memo[MAXN+1][MAXN+1];
void init() {
    fac[0] = 1;
    rep(i,1,MAXN) fac[i] = (i * fac[i-1]) % MOD;
```

```
    memset(choose_memo, -1, sizeof choose_memo);
}
ll mult(ll a, ll b) { return (a * b) % MOD; }
ll choose_mod(int n, int k) {
    if (choose_memo[n][k] != -1) return choose_memo[n][k];
    return choose_memo[n][k] = mult(fac[n], multinv(mult(fac[k], fac[n-k]), MOD));
}
```

6.4. Modular Multinomial Coefficient

```
typedef long long int ll;
const ll MOD = 100000000711; // a prime number
const int MAXN = 1000;

/* ==================== */
/*  MODULAR MULTINOMIAL  */
/* ==================== */

ll memo[MAXN+1][MAXN+1];
ll choose(int n, int k) {
    ll& ans = memo[n][k];
    if (ans != -1) return ans;
    if (k == 0) return ans = 1;
    if (n == k) return ans = 1;
    if (n < k) return ans = 0;
    return ans = (choose(n-1,k) + choose(n-1,k-1)) % MOD;
}

// reference: https://math.stackexchange.com/a/204209/503889
ll multinomial(vector<int> ks) {
    int n = 0;
    ll ans = 1;
    for (int k : ks) {
        n += k;
        ans = (ans * choose(n,k)) % MOD;
    }
    return ans;
}
```

6.5. Modular Fibonacci

```
//=========================================================
// Modular Fibonacci with (Modular) Matrix Exponentiation
//=========================================================
//source: http://mathoverflow.net/questions/40816/fibonacci-series-mod-a-number
#include <cstdio>
#include <vector>
using namespace std;
typedef unsigned long long ull;
const ull MOD = 1000000000;

vector<ull> mult(const vector<ull>& A, const vector<ull>& B) {
  vector<ull> res
  { (((A[0] * B[0]) % MOD) + ((A[1] * B[2]) % MOD)) % MOD, //m11
    (((A[0] * B[1]) % MOD) + ((A[1] * B[3]) % MOD)) % MOD, //m12
    (((A[2] * B[0]) % MOD) + ((A[3] * B[2]) % MOD)) % MOD, //m21
    (((A[2] * B[1]) % MOD) + ((A[3] * B[3]) % MOD)) % MOD //m22
  };
  return res;
}

vector<ull> raise(const vector<ull>& matrix, ull exp) {
  if (exp == 1)
    return matrix;
  ull m = exp / 2;
  vector<ull> A = raise(matrix, m);
  if (exp % 2 == 0)
    return mult(A, A);
  else
    return mult(mult(A, A), matrix);
}

int main() {
  int P;
  int k;
  ull y;
  scanf("%d", &P);
  vector<ull> fib_matrix { 1, 1, 1, 0 }; //starting fibonacci matrix [f2, f1, f1, f0]
  while (P-- > 0) {
    scanf("%d %llu", &k, &y);
    vector<ull> ansm = raise(fib_matrix, y);
    ull ans = ansm[1];
    printf("%d %llu\n", k, ans);
  }
  return 0;
}
```

6.6. Binary Modular Exponentiation

```
int mod_pow(int b, int e, int m) {
  if (e == 1)
    return b % m;
  int he = e / 2;
  int x = mod_pow(b, he, m);
  x = (x * x) % m;
  if (e % 2 == 1)
    x = (x * b) % m;
  return x;
}
```

6.7. Integer Root Square

```
// using sqrt()
bool perfect_square(ll x, ll& root) {
  if (x < 0) return false;
  root = (ll)sqrt(x);
```

```
  return (root * root == x || ++root * root == x);
}

// Newton's method
ll isqrt(ll x) {
  ll y0 = x;
  while (true) {
    ll y1 = (y0 +  x / y0) / 2;
    if (y1 == y0) break;
    y0 = y1;
  }
  return y0;
}
bool isPerfectSquare(ll x, ll& root) {
  root = isqrt(x);
  return root * root == x;
}
```

7. Geometry

7.1. Geometry 2D Utils

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;
// --------------------------------
const double PI = acos(-1);
const double EPS = 1e-8;

/* ========================== */
/* Example of Point Definition */
/* ========================== */
struct Point {
    double x, y;
    bool operator==(const Point& p) const { return x==p.x && y == p.y; }
    Point operator+(const Point& p) const { return {x+p.x, y+p.y}; }
    Point operator-(const Point& p) const { return {x-p.x, y-p.y}; }
    Point operator*(double d) const { return {x*d, y*d}; }
    double norm2() { return x*x + y*y; }
    double norm() { return sqrt(norm2()); }
    double dot(const Point& p) { return x*p.x + y*p.y; }
    double cross(const Point& p) { return x*p.y - y*p.x; }
    double angle() {
        double angle = atan2(y, x);
        if (angle < 0) angle += 2 * PI;
        return angle;
    }
    Point unit() {
        double d = norm();
        return {x/d,y/d};
    }
};

/* ========================================================= */
/* Cross Product -> orientation of point with respect to ray */
/* ========================================================= */
// cross product (b - a) x (c - a)
ll cross(Point& a, Point& b, Point& c) {
    ll dx0 = b.x - a.x, dy0 = b.y - a.y;
    ll dx1 = c.x - a.x, dy1 = c.y - a.y;
    return dx0 * dy1 - dx1 * dy0;
    // return (b - a).cross(c - a); // alternatively, using struct function
}

// calculates the cross product (b - a) x (c - a)
// and returns orientation:
// LEFT (1):      c is to the left of  ray (a -> b)
// RIGHT (-1):    c is to the right of ray (a -> b)
// COLLINEAR (0): c is collinear to    ray (a -> b)
// inspired by: https://www.geeksforgeeks.org/orientation-3-ordered-points/
int orientation(Point& a, Point& b, Point& c) {
    ll tmp = cross(a,b,c);
    return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
}

/* ======================================================== */
/* Check if a segment is below another segment (wrt a ray) */
/* ======================================================== */
// i.e: check if a segment is intersected by the ray first
// Assumptions:
// 1) for each segment:
//  p1 should be LEFT (or COLLINEAR) and p2 should be RIGHT (or COLLINEAR) wrt ray
// 2) segments do not intersect each other
// 3) segments are not collinear to the ray
// 4) the ray intersects all segments
struct Segment { Point p1, p2;};
Segment segments[MAXN]; // array of line segments
bool is_si_below_sj(int i, int j) { // custom comparator based on cross product
    Segment& si = segments[i];
    Segment& sj = segments[j];
    return (si.p1.x >= sj.p1.x) ?
        cross(si.p1, sj.p2, sj.p1) > 0:
        cross(sj.p1, si.p1, si.p2) > 0;
}
// this can be used to keep a set of segments ordered by order of intersection
// by the ray, for example, active segments during a SWEEP LINE
set<int, bool(*)(int,int)> active_segments(is_si_below_sj); // ordered set

/* ====================== */
/* Rectangle Intersection */
/* ====================== */
bool do_rectangles_intersect(Point& dl1, Point& ur1, Point& dl2, Point& ur2) {
    return max(dl1.x, dl2.x) <= min(ur1.x, ur2.x) && max(dl1.y, dl2.y) <= min(ur1.y, ur2.
        y);
}

/* ======================== */
```

```cpp
/* Line Segment Intersection */
/* ======================== */
// returns whether segments p1q1 and p2q2 intersect, inspired by:
// https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
bool do_segments_intersect(Point& p1, Point& q1, Point& p2, Point& q2) {
    int o11 = orientation(p1, q1, p2);
    int o12 = orientation(p1, q1, q2);
    int o21 = orientation(p2, q2, p1);
    int o22 = orientation(p2, q2, q1);
    if (o11 != o12 and o21 != o22) // general case -> non-collinear intersection
        return true;
    if (o11 == o12 and o11 == 0) { // particular case -> segments are collinear
        Point dl1 = {min(p1.x, q1.x), min(p1.y, q1.y)};
        Point ur1 = {max(p1.x, q1.x), max(p1.y, q1.y)};
        Point dl2 = {min(p2.x, q2.x), min(p2.y, q2.y)};
        Point ur2 = {max(p2.x, q2.x), max(p2.y, q2.y)};
        return do_rectangles_intersect(dl1, ur1, dl2, ur2);
    }
    return false;
}

/* ================== */
/* Circle Intersection */
/* ================== */
struct Circle { double x, y, r; }
bool is_fully_outside(double r1, double r2, double d_sqr) {
    double tmp = r1 + r2;
    return d_sqr > tmp * tmp;
}
bool is_fully_inside(double r1, double r2, double d_sqr) {
    if (r1 > r2) return false;
    double tmp = r2 - r1;
    return d_sqr < tmp * tmp;
}
bool do_circles_intersect(Circle& c1, Circle& c2) {
    double dx = c1.x - c2.x;
    double dy = c1.y - c2.y;
    double d_sqr = dx * dx + dy * dy;
    if (is_fully_inside(c1.r, c2.r, d_sqr)) return false;
    if (is_fully_inside(c2.r, c1.r, d_sqr)) return false;
    if (is_fully_outside(c1.r, c2.r, d_sqr)) return false;
    return true;
}

/* ==================== */
/* Point - Line distance */
/* ==================== */
// get distance between p and projection of p on line <- a - b ->
double point_line_dist(Point& p, Point& a, Point& b) {
    Point d = b-a;
    double t = d.dot(p-a) / d.norm2();
    return (a + d * t - p).norm();
}

/* ======================= */
/* Point - Segment distance */
/* ======================= */
// get distance between p and truncated projection of p on segment a -> b
double point_segment_dist(Point& p, Point& a, Point& b) {
    if (a==b) return (p-a).norm(); // segment is a single point
    Point d = b-a; // direction
    double t = d.dot(p-a) / d.norm2();
    if (t <= 0) return (p-a).norm(); // truncate left
    if (t >= 1) return (p-b).norm(); // truncate right
    return (a + d * t - p).norm();
}

/* ==================================== */
/* Straight Line Hashing (integer coords) */
/* ==================================== */
// task: given 2 points p1, p2 with integer coordinates, output a unique
// representation {a,b,c} such that a*x + b*y + c = 0 is the equation
// of the straight line defined by p1, p2. This representation must be
// unique for each straight line, no matter which p1 and p2 are sampled.
struct Point {int x, y; };
struct Line { int a, b, c; };
int gcd(int a, int b) { // greatest common divisor
    a = abs(a); b = abs(b);
    while(b) { int c = a; a = b; b = c % b; }
    return a;
}
Line getLine(Point p1, Point p2) {
    int a = p1.y - p2.y;
    int b = p2.x - p1.x;
    int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
    int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
    int f = gcd(a, gcd(b, c)) * sgn;
    a /= f;
    b /= f;
    c /= f;
    return {a, b, c};
}
```

7.2. Trigonometry

```cpp
/* ================ */
/* Angle of a vector */
/* ================ */
const double PI = acos(-1);
const double _2PI = 2 * PI;

double correct_angle(double angle) { // to ensure 0 <= angle <= 2PI
    while (angle < 0) angle += _2PI;
    while (angle > _2PI) angle -= _2PI;
    return angle;
}
double angle(double x, double y) {
    // atan2 by itself returns an angle in range [-PI, PI]
```

```
    // no need to "correct it" if that range is ok for you
    return correct_angle(atan2(y, x));
}

/* ============== */
/* Cosine Theorem */
/* ============== */
// Given triangle with sides a, b and c, returns the angle opposed to side a.
// a^2 = b^2 + c^2 - 2*b*c*cos(alpha)
// => alpha = acos((b^2 + c^2 - a^2) /(2*b*c))
double get_angle(double a, double b, double c) {
    return acos((b*b + c*c - a*a)/(2*b*c));
}
```

7.3. Polygon Area

```
/* ====================================== */
/* Area of 2D non self intersecting Polygon */
/* ====================================== */
//based on Green's Theorem:
//http://math.blogoverflow.com/2014/06/04/greens-theorem-and-area-of-polygons/

#include <bits/stdc++.h>
int N = 1000;
struct Point { int x, y; };
Point P[N];

double area() {
    int A = 0;
    for (int i = N-1, j = 0; j < N; i=j++)
        A += (P[i].x + P[j].x) * (P[j].y - P[i].y);
    return fabs(A * 0.5);
}
```

7.4. Point Inside Polygon

```
/* ================ */
/* Point in Polygon */
/* ================ */

#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; ++i)
struct Point { double x, y; };

// cross product (b - a) x (c - a)
double cross(Point& a, Point& b, Point& c) {
    double dx0 = b.x - a.x, dy0 = b.y - a.y;
    double dx1 = c.x - a.x, dy1 = c.y - a.y;
    return dx0 * dy1 - dx1 * dy0;
}
int orientation(Point& a, Point& b, Point& c) {
    double tmp = cross(a,b,c);
    return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
}

// ------------------------------------------------
// General methods: for complex / simple polygons

/* Nonzero Rule (winding number) */
bool inPolygon_nonzero(Point p, vector<Point>& pts) {
    int wn = 0; // winding number
    Point prev = pts.back();
    rep (i, 0, (int)pts.size() - 1) {
        Point curr = pts[i];
        if (prev.y <= p.y) {
            if (p.y < curr.y && cross(prev, curr, p) > 0)
                ++ wn; // upward & left
        } else {
            if (p.y >= curr.y && cross(prev, curr, p) < 0)
                -- wn; // downward & right
        }
        prev = curr;
    }
    return wn != 0; // non-zero :)
}

/* EvenOdd Rule (ray casting - crossing number) */
bool inPolygon_evenodd(Point p, vector<Point>& pts) {
    int cn = 0; // crossing number
    Point prev = pts.back();
    rep (i, 0, (int)pts.size() - 1) {
        Point curr = pts[i];
        if (((prev.y <= p.y) && (p.y < curr.y)) // upward crossing
            || ((prev.y > p.y) && (p.y >= curr.y))) { // downward crossing
            // check intersect's x-coordinate to the right of p
            double t = (p.y - prev.y) / (curr.y - prev.y);
            if (p.x < prev.x + t * (curr.x - prev.x))
                ++cn;
        }
        prev = curr;
    }
    return (cn & 1); // odd -> in, even -> out
}

// ------------------------------------------------
// Convex Polygon method: check orientation changes
bool inConvexPolygon(Point& p, vector<Point>& pts) {
    int n = pts.size();
    int o_min = 0, o_max = 0;
    for (int i=0, j=n-1; i < n; j=i++) {
        int o = orientation(pts[j], pts[i], p);
        if (o == 1) o_max = 1;
        else if (o == -1) o_min = -1;
        if (o_max - o_min == 2) return false;
    }
    return true;
}
```

7.5. Convex Hull

```cpp
#include <bits/stdc++.h>
using namespace std;
#define rep(i,a,b) for(int i = a; i <= b; ++i)
#define invrep(i,b,a) for(int i = b; i >= a; --i)
typedef long long int ll;
// ------------------------------------------
// Convex Hull: Andrew's Montone Chain Algorithm
// ------------------------------------------

struct Point {
    ll x, y;
    bool operator<(const Point& p) {
        return x < p.x || (x == p.x && y < p.y);
    }
};

ll cross(Point& a, Point& b, Point& c) {
    ll dx0 = b.x - a.x, dy0 = b.y - a.y;
    ll dx1 = c.x - a.x, dy1 = c.y - a.y;
    return dx0 * dy1 - dx1 * dy0;
}

vector<Point> upper_hull(vector<Point>& P) {
    // sort points lexicographically
    int n = P.size(), k = 0;
    sort(P.begin(), P.end());

    // build upper hull
    vector<Point> uh(n);
    invrep (i, n-1, 0) {
        while (k >= 2 && cross(uh[k-2], uh[k-1], P[i]) <= 0) k--;
        uh[k++] = P[i];
    }
    uh.resize(k);
    return uh;
}

vector<Point> lower_hull(vector<Point>& P) {
    // sort points lexicographically
    int n = P.size(), k = 0;
    sort(P.begin(), P.end());

    // collect lower hull
    vector<Point> lh(n);
    rep (i, 0, n-1) {
        while (k >= 2 && cross(lh[k-2], lh[k-1], P[i]) <= 0) k--;
        lh[k++] = P[i];
    }
    lh.resize(k);
    return lh;
}

vector<Point> convex_hull(vector<Point>& P) {
    int n = P.size(), k = 0;

    // set initial capacity
    vector<Point> H(2*n);

    // Sort points lexicographically
    sort(P.begin(), P.end());

    // Build lower hull
    for (int i = 0; i < n; ++i) {
        while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    // Build upper hull
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    // remove extra space
    H.resize(k-1);
    return H;
}
```

7.6. Green's Theorem

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

struct Point { double x, y; };

// Computes the line integral of the vector field <0,x> over the arc of the circle with radius 'r'
// and x-coordinate 'x' from angle 'a' to angle 'b'. The 'y' goes away in the integral so it
// it doesn't matter.
// This can be done using a parameterization of the arc in polar coordinates:
// x(t) = x + r * cos(t)
// y(t) = y + r * sin(t)
// a <= t <= b
// The final integral can be seen here:
// https://www.wolframalpha.com/input/?i=integral((x+%2B+r*cos(t))+**+derivative(y+%2B+r*sin(t))+**+dt,+t%3Da..b)
double arc_integral(double x, double r, double a, double b) {
    return x * r * (sin(b) - sin(a)) + r * r * 0.5 * (0.5 * (sin(2*b) - sin(2*a)) + b - a);
}

// Computes the line integral of the vector field <0, x> over the directed segment a -> b
// This can be done using the parameterization:
// x(t) = a.x + (b.x - a.x) * t
```

```cpp
// y(t) = a.y + (b.y - a.y) * t
// 0 <= t <= 1
double segment_integral(Point& a, Point& b) {
    return 0.5 * (a.x + b.x) * (b.y - a.y);
}
```

8. Strings

8.1. Rolling Hashing

```cpp
#include <bits/stdc++.h>
using namespace std;
#define rep(i,a,b) for(int i = a; i <= b; ++i)
typedef unsigned long long int ull;

const int MAXLEN = 1e6;

// -----------------------------------
// rolling hashing using a single prime

struct RH_single { // rolling hashing
    static const ull B = 127; // base
    static const ull P = 1e9 + 7; // prime
    static ull pow[MAXLEN];

    static ull add(ull x, ull y) { return (x + y) % P; }
    static ull mul(ull x, ull y) { return (x * y) % P; }

    static void init() {
        pow[0] = 1;
        rep(i, 1, MAXLEN-1) pow[i] = mul(B, pow[i-1]);
    }

    vector<ull> h;
    int len;
    RH(string& s) {
        len = s.size();
        h.resize(len);
        h[0] = s[0] - 'a';
        rep(i,1,len-1) h[i] = add(mul(h[i-1], B),s[i] - 'a');
    }

    ull hash(int i, int j) {
        if (i == 0) return h[j];
        return add(h[j], P - mul(h[i-1], pow[j - i + 1]));
    }
    ull hash() { return h[len-1]; }
};
ull RH::pow[MAXLEN]; // necessary for the code to compile

// -------------------------------------------------
// rolling hashing using 2 primes (for extra safety)

struct RH_double { // rolling hashing
    static const ull B = 127; // base
    static const ull P[2]; // primes
    static ull pow[2][MAXLEN];

    static ull add(ull x, ull y, int a) { return (x + y) % P[a]; }
    static ull mul(ull x, ull y, int a) { return (x * y) % P[a]; }

    static void init(int a) {
        pow[a][0] = 1;
        rep(i, 1, MAXLEN-1) pow[a][i] =  mul(B, pow[a][i-1], a);
    }
    static void init() { init(0); init(1); }

    vector<ull> h[2];
    int len;
    RH(string& s) {
        len = s.size();
        rep(a,0,1) {
            h[a].resize(len);
            h[a][0] = s[0] - 'a';
            rep(i,1,len-1) h[a][i] = add(mul(h[a][i-1], B, a),s[i] - 'a', a);
        }
    }

    ull hash(int i, int j, int a) {
        if (i == 0) return h[a][j];
        return add(h[a][j], P[a] - mul(h[a][i-1], pow[a][j-i+1], a), a);
    }
    ull hash(int i, int j) {
        return hash(i,j,0) << 32 | hash(i,j,1);
    }
    ull hash() { return hash(0, len-1); }
};
// these lines are necessary for the code to compile
const ull RH::P[2] = {(int)1e9+7, (int)1e9+9};
ull RH::pow[2][MAXLEN];


// ----- usage & testing
int main() {
    RH_single::init();
    while (true) {
        string s1, s2; cin >> s1 >> s2;
        if (s1.size() == s2.size()) {
            ull h1 = RH_single(s1).hash(0, s1.size()-1);
            ull h2 = RH_single(s2).hash(0, s2.size()-1);
            if (s1 == s2 ? h1 == h2 : h1 != h2) {
                cout << "test passed!" << endl;
            } else {
                cout << "test failed :(" << endl;
            }
        }
    }
```

```
}
```

8.2. Suffix Array

```cpp
// ======================================================
// Suffix Array Construction : Prefix Doubling + Radix Sort
// ======================================================
// Complexity: O(N*log(N))
// reference: https://www.cs.helsinki.fi/u/tpkarkka/opetus/10s/spa/lecture11.pdf
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; ++i)
#define invrep(i,b,a) for(int i = b; i >= a; --i)
using namespace std;

namespace SA {
    const int MAXN = (int)1e6;
    int n;
    int rank[MAXN], rank_tmp[MAXN];
    int sa[MAXN], sa_tmp[MAXN];
    int inline get_rank(int i) { return i < n ? rank[i]: 0; }

    // stable sort suffix array based on the ranking function:
    // sa[i] -> get_rank(sa[i] + k)
    void counting_sort(int maxv, int k) {
        static int counts[MAXN];
        memset(counts, 0, sizeof(int) * (maxv+1));
        rep(i,0,n-1) counts[get_rank(i+k)]++;
        rep(i,1,maxv) counts[i] += counts[i-1];
        invrep(i,n-1,0) sa_tmp[--counts[get_rank(sa[i]+k)]] = sa[i];
        swap(sa, sa_tmp);
    }

    // word: sequence of values
    // maxv: maximum value in 'word'
    // suffix_indexes: suffix indexes to be sorted
    // ** 1 <= word[i] <= maxv
    void sort_suffix_indexes(vector<int>& word, int maxv, vector<int>& suffix_indexes) {
        n = word.size();
        rep(i,0,n-1) {
            sa[i] = i; rank[i] = word[i];
            assert(word[i] >= 1 and word[i] <= maxv);
        }
        for (int h=1; h < n; h <<= 1) {
            // two counting sort passes to achieve O(n) sorting complexity
            // (i.e. 2-pass radix sort)
            counting_sort(maxv, h);
            counting_sort(maxv, 0);
            // re-compute ranking
            int r = 1;
            rank_tmp[sa[0]] = r;
            rep(i,1,n-1) {
                if (rank[sa[i]] != rank[sa[i-1]] or
                    get_rank(sa[i]+h) != get_rank(sa[i-1]+h)) r++;
                rank_tmp[sa[i]] = r;
            }
            swap(rank, rank_tmp);
            maxv = r;
        }
        // copy suffix indexes in their final order
        suffix_indexes.resize(n);
        rep(i,0,n-1) suffix_indexes[i] = sa[i];
    }

    void sort_suffix_indexes(vector<int>& word, vector<int>& suffix_indexes) {
        int maxv = 0;
        for (int v : word) if (maxv < v) maxv = v;
        sort_suffix_indexes(word, maxv, suffix_indexes);
    }
}

int main() { // testing
    string testword; cin >> testword;
    vector<int> w;
    for (char c : testword) w.push_back(c - 'a' + 1);
    vector<int> suffix_indexes;
    SA::sort_suffix_indexes(w, suffix_indexes);
    for (int i : suffix_indexes) cout << i << ":\t" << testword.substr(i) << endl;
}
```

8.3. KMP (Knuth Morris Pratt)

```cpp
#include <bits/stdc++.h>
using namespace std;
#define rep(i,a,b) for(int i=a; i<=b; ++i)

// Build longest proper prefix/suffix array (lps) for pattern
// lps[i] = length of the longest proper prefix which is also suffix in pattern[0 .. i]
void init_lps(string& pattern, int lps[]) {
    int n = pattern.size();
    lps[0] = 0; // base case: no proper prefix/suffix for pattern[0 .. 0] (length 1)
    rep(j, 1, n-1) { // for each pattern[0 .. j]
        int i = lps[j-1]; // i points to the char next to lps of previous iteration
        while (pattern[i] != pattern[j] and i > 0) i = lps[i-1];
        lps[j] = pattern[i] == pattern[j] ? i+1 : 0;
    }
}

// Count number of matches of pattern string in target string using KMP algorithm
int count_matches(string& pattern, string& target) {
    int n = pattern.size(), m = target.size();
    int lps[n];
    init_lps(pattern, lps); // build lps array
    int matches = 0;
    int i = 0; // i tracks current char in pattern to compare
    rep(j, 0, m-1) { // j tracks each char in target to compare
        // try to keep prefix before i as long as possible while ensuring i matches j
        while (pattern[i] != target[j] and i > 0) i = lps[i-1];
        if (pattern[i] == target[j]) {
```

```
            if (++i == n) { // we matched the whole pattern
                i = lps[n-1]; // shift the pattern so that the longest proper prefix/
                    suffix pair is aligned
                matches++;
            }
        }
    }
    return matches;
}

int main() {
    string target, pattern;
    while (true) {
        cin >> target >> pattern;
        cout << count_matches(pattern, target) << " matches" << endl;
    }
    return 0;
}
```

8.4. Shortest Repeating Cycle

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

int shortest_repeating_cycle(string& seq) {
    // KMP : lps step
    int n = seq.size();
    int lps[n];
    lps[0] = 0;
    int i = 0, j = 1;
    while (j < n) {
        while (i > 0 and seq[i] != seq[j])
            i = lps[i-1];
        if (seq[i] == seq[j])
            lps[j] = ++i;
        else
            lps[j] = 0;
        j++;
    }
    int len = n - lps[n-1];
    return (n % len) ? n : len;
}

// test
int main() {
    string line; cin >> line;
    int cycle = shortest_repeating_cycle(line);
    cout << line.substr(0, cycle) << endl;
    return 0;
}
```

8.5. Trie

```
const int MAX_NODES = 1000;
int tree[MAX_NODES][26];
int freqs[MAX_NODES];
int nodes;

void init_trie() {
    memset(tree, -1, sizeof tree);
    memset(freqs, 0, sizeof freqs);
    nodes = 1;
}

void add_word(string& word) {
    int cur = 0;
    freqs[0]++;
    for (char c : word) {
        int& nxt = tree[cur][c-'a'];
        if (nxt == -1) nxt = nodes++;
        cur = nxt;
        freqs[cur]++; // update node frequency
    }
}
```