Contents

1 C++ Template

```cpp
#pragma GCC optimize("Ofast")
#include <bits/stdc++.h>
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 long long int ull;
typedef long long int ll;
typedef vector<int> vi;
typedef pair<int,int> ii;
// -------------------------------
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
    return 0;
}
```

2 C++ Cheat Sheet

```cpp
/* ================================= */
/* Input/Output with C++: cin & cout */
/* ================================= */

// reading many lines of unknown length
string line;
while(getline(cin, line)) {}

// reading tokens from a line
string token;
stringstream ss(line);
while (ss >> token) { /* do something with token */}

// printing floating with fixed precision
cout << setprecision(6) << fixed;
cout << 12312.12312355;

/* ================================= */
/* CONVERTING FROM STRING TO NUMBERS */
/* ================================= */

// 1) stringstream
string s = "12345";
stringstream ss(s);
int x;
ss >> x; // x = 12345
ss << "12345678910";
long long y;
ss >> y; // y = 12345678910

// 2) stoi, stoll
string str_dec = "2001, A Space Odyssey";
string str_hex = "40c3";
string str_bin = "-10010110001";
string str_auto = "0x7f";
int sz;
int i_dec = stoi(str_dec,&sz);
int i_hex = stoi(str_hex,0,16);
int i_bin = stoi(str_bin,0,2);
int i_auto = stoi(str_auto,0,0);
cout << str_dec << ": " << i_dec << " and [" << str_dec.substr(sz) << "]\n";
cout << str_hex << ": " << i_hex << '\n';
cout << str_bin << ": " << i_bin << '\n';
cout << str_auto << ": " << i_auto << '\n';
// 2001, A Space Odyssey: 2001 and [, A Space Odyssey]
// 40c3:  16579
// -10010110001: -1201
// 0x7f: 127
string str = "8246821 0xffff 020";
int sz = 0;
while (!str.empty()) {
    long long ll = stoll(str,&sz,0);
    cout << str.substr(0,sz) << " interpreted as " << ll << '\n';
    str = str.substr(sz);
}
// 8246821 interpreted as 8246821
//  0xffff interpreted as 65535
//  020 interpreted as 16

/* ========================== */
/* 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 */
/* ============================ */

// 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";
size_t pos = str.find("ra");
size_t pos = str.find('m');
```

```cpp
if (pos == string::npos) // not found

// substrings
string subs = str.substr(pos, length);
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
string str1("green apple");
string str2("red apple");
if (str1.compare(str2) != 0)
    cout << str1 << " is not " << str2 << '\n';
if (str1.compare(6,5,"apple") == 0)
    cout << "still, " << str1 << " is an apple\n";
if (str2.compare(str2.size()-5,5,"apple") == 0)
    cout << "and " << str2 << " is also an apple\n";
if (str1.compare(6,5,str2,4,5) == 0)
    cout << "therefore, both are apples\n";
// green apple is not red apple
// still, green apple is an apple
// and red apple is also an apple
// therefore, both are apples

/* ===================== */
/*  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;

/* =============== */
/* 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));
}

/* ============== */
/* 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';
}

/* ================= */
/* CLIMITS CONSTANTS */
/* ================= */
INT_MIN INT_MAX UINT_MAX LONG_MIN LONG_MAX ULONG_MAX LLONG_MIN LLONG_MAX ULLONG_MAX
```

3 Data Structures

3.1 C++ STL

3.1.1 Pairs & Tuples

```
// references:
// https://www.geeksforgeeks.org/returning-multiple-values-from-a-function-using-tuple-and-pair-in-c/
// http://www.cplusplus.com/reference/utility/pair/
// http://www.cplusplus.com/reference/tuple/

//======
// PAIR
//======
// Example: pair of ints
typedef pair<int,int> ii; // use ii as abbreviation
// initialization
ii p(5,5); // option 1
ii p = make_pair(5,5) // option 2
ii p = {5, 5}; // option 3
// getting values
int x = p.first, y = p.second;
// modifying values
p.first++, p.second--; // p = {6, 4}

//=======
// TUPLE
//=======
// Example: tuples of 3 ints
typedef tuple<int,int,int> iii; // use iii as abbreviation
// initialization
iii t(5,5,5); // option 1
iii t = make_tuple(5,5,5); // option 2
iii t = {5, 5, 5}; // option 3
// getting values
int x,y,z;
x = get<0>(t), y = get<1>(t), z = get<2>(t); // option 1
tie(x,y,z) = t; // option 2
// modifying values
get<0>(t)++, get<1>(t)--, get<2>(t)+=2; // t = {6, 4, 7}
```

3.1.2 Array

```
//================
// declare arrays
//================
int arr[10];
int arr[10][10];
int arr[5] = {1, 2, 3, 4, 5};
int arr[4][2] = {{0,1}, {1,0}, {0,-1}, {-1,0}};

//============================
// fill array using std::fill
//============================
// http://www.cplusplus.com/reference/algorithm/fill/

// 1) arrays 1D
int arr[100];
fill(arr, arr+4, -5);
fill(arr, arr+N, val);
fill(arr + offset, arr + N, val);
double arr[100];
fill(arr, arr+7, 0.999);

// 2) arrays 2D or more
int arr[100][100];
fill(&arr[0][0], &arr[0][0] + sizeof(arr), -1231);
```

```

//========================
// fill array using memset
//========================
int arr[100][100];
memset(arr, -1, sizeof(arr));
memset(arr, 0, sizeof(arr));
// ** only works with 0 and -1 for arryas of ints/longs
//   because memset works on bytes (same value is written on each char)
//   sizeof(arr) returns the number of bytes in arr

// in the case of char arrays, we can set any value, since
// sizeof(char) = 1 (each char uses a single byte)
char char_arr[100][100];
memset(char_arr, 'k', sizeof(char_arr));

// filling with -1/0 the first N ints in arr
int arr[MAXN];
memset(arr, -1, sizeof(int) * N);
memset(arr, 0, sizeof(int) * N);

// interesting links:
// https://stackoverflow.com/questions/936687/how-do-i-declare-a-2d-array-in-c-using-new/
// https://stackoverflow.com/questions/8767166/passing-a-2d-array-to-a-c-function
```

3.1.3 Vector

```
// references:
// http://www.cplusplus.com/reference/vector/vector/
// https://www.geeksforgeeks.org/vector-in-cpp-stl/
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i=a; i<=b; i++)
using namespace std;

//=============================
// DECLARATION & INITIALIZATION
//=============================

// vector of ints
vector<int> v; // empty
vector<int> v(100); // size 100
vector<int> v(N); // size N, make sure N is defined
vector<int> v(N, 2); // size N filled with 2's
vector<int> v = { 1, 2, 3, 5, 6 }; // list initialization (since C++11)
v[0] = -8; // v = { -8, 2, 3, 5, 6 }
v[1] = 0; // v = { -8, 0, 3, 5, 6 }

// vector of vector of ints
// a matrix of R rows by C columns filled with -1
vector<vector<int>> matrix(R, vector<int>(C,-1));

//=====================================================
// MODIFYING A VECTOR (capacity, size, adding elements)
///=====================================================

vector<int> v; //  v = {}, v.size() == 0
v.reserve(1000); // reserve 1000 x sizeof(int) bytes of contiguous memory in advance
// ** we use v.reserve(MAXN) when we when we know the maximum memory we will ever
//    need to prevent unnecessary memory reallocations

rep(i, 1, 10) v.push_back(i); // v = { 1, 2, 3, ..., 10 }, v.size() == 10
int x = v.front(); // x = 1
int y = v.back(); // y = 10
v.pop_back(); // remove last element -> v = { 1, 2, 3, ..., 9 }, v.size() == 9

// clearing
v.clear(); // v = {}, v.size() == 0

//========
// RESIZE
//========
rep(i,1,10) v.push_back(i); // v = { 1, 2, ..., 10 }
v.resize(5); // v = { 1, 2, 3, 4, 5 }
v.resize(8,100); // v = { 1, 2, 3, 4, 5, 100, 100, 100 }
v.resize(12); // v = { 1, 2, 3, 4, 5, 100, 100, 100, 0, 0, 0, 0 }

//========
// ASSIGN
//========
v.assign(N, 4); // v = { 4, 4, ..., 4 } (N times)

vector<int> v2;
v2.assign(v.begin(), v.end()); // v2 = v
v2.assign(v.begin() + 1, v.end() - 1); // v2 = v[1:-1]

int arr[5] = {1, 2, 3, 4, 5};
v2.assign(arr, arr + 5); // v2 = {1, 2, 3, 4, 5}
v2.assign(arr, arr + 3); // v2 = {1, 2, 3}

//==========================
// EMPLACE_BACK VS PUSH_BACK
//==========================
struct CustomData {
    int x; double y; string z;
    CustomData(int x, double y, string z) : x(x), y(y), z(z) {}
};
vector<CustomData> v;
// option 1: with push_back() -> object is created and then copied
v.push_back(CustomData(1,2.32,"foo")); // using constructor
v.push_back({1, 2.32,"bar"}); // c++11: using curly braces
// option 2: with emplace_back() -> object is created in its final location ;)
v.emplace_back(1, 2.32, "foo");
// ** NOTE: for emplace_back() make sure your custom struct/class has a constructor
//          for push_back(), no need to define a constructor


//======================
// ITERATING OVER VECTORS
```

```cpp
//=======================
// reference:
// https://stackoverflow.com/questions/15176104/c11-range-based-loop-get-item-by-value-or-reference-to-const

// 1) foward direction

vector<CustomData> v(100); // vector of custom type
// option 1: iterate over element copies (slower)
for (auto x : v) { /* do something */ }
// option 2: iterate over references (faster)
for (auto& x : v) { /* do something */ }
// option 3: iterate over const references (equally fast)
// * the const keyword is just to prevent unintended modifications
for (const auto& x : v) { /* do something */ }

vector<int> v {1, 2, 3, 4, 5, 6}; // vector of ints
for (int x : v) { /* do something */ }
for (int& x : v) { /* do something */ }
for (const int& x : v) { /* do something */ }

// using iterators
for (auto it = v.begin(); it != v.end(); ++it) {
    const auto& x = *it; // use *it to access original element pointed by it
    /* do something with x */
}

// 2) backward direction
for (auto it = v.rbegin(); it != v.rend(); ++it) {
    const auto& x = *it;
}

//===================
// SWAPPING 2 VECTORS
//===================
vector<int> v1 = {1, 1, 1, 1};
vector<int> v2 = {2, 2, 2};
v1.swap(v2); // v1 = {2, 2, 2}, v2 = {1, 1, 1, 1}
```

3.1.4 Queue & Stack

```cpp
// references:
// http://www.cplusplus.com/reference/queue/queue/
// https://www.geeksforgeeks.org/queue-cpp-stl/
// http://www.cplusplus.com/reference/stack/stack/
// https://www.geeksforgeeks.org/stack-in-cpp-stl/

//===== QUEUE =====
queue<int> q;
// adding to queue
rep(i,1,5) q.push(i); // q = {1, 2, 3, 4, 5}
// OR
rep(i,1,5) q.emplace(i); // q = {1, 2, 3, 4, 5}
// removing from queue
while (!q.empty()) {
    cout << q.front() << ' ';
    q.pop();
} // output: 1 2 3 4 5

//===== STACK =====
stack<int> s;
// adding to stack
rep(i,1,5) s.push(i); // s = {1, 2, 3, 4, 5}
// OR
rep(i,1,5) s.emplace(i); // s = {1, 2, 3, 4, 5}
// removing from stack
while (!s.empty()) {
    cout << s.top() << ' ';
    s.pop();
} // output: 5 4 3 2 1
```

3.1.5 Priority Queue

```cpp
// references:
// http://www.cplusplus.com/reference/queue/priority_queue/
// https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/

//===================
// 1) MAXHEAP of ints
//===================
priority_queue<int> q;
q.push(30);
q.push(100);
q.push(25);
q.push(40);
cout << "Popping out elements...";
while (!q.empty()) {
    cout << ' ' << q.top();
    q.pop();
}
cout << '\n';
// Popping out elements... 100 40 30 25

//===================
// 2) MINHEAP of ints
//===================
priority_queue<int, vector<int>, greater<int>> q;
q.push(30);
q.push(100);
q.push(25);
q.push(40);
cout << "Popping out elements...";
while (!q.empty()) {
    cout << ' ' << q.top();
    q.pop();
}
cout << '\n';
// Popping out elements... 25 30 40 100
```

```
//====================================
// 3) custom data + custom comparator
//====================================

// option 1: overload operator< inside your struct/class
struct Event {
    double time; string name;
    Event (double t, string n) : time(t), name(n) {}
    bool overload<(const Event& rhs) const {
        // define your < operator however you want
        return time > rhs.time;
    }
};
priority_queue<Event> q;

// option 2: use a functor
struct Event {
    double time; string name;
    Event (double t, string n) : time(t), name(n) {}
};
struct EventCmp {
    bool operator()(const Event& lhs, const Event& rhs) {
        return lhs.time > rhs.time;
    }
};
priority_queue<Event, vector<Event>, EventCmp> q;

// option 3: use a lambda function
struct Event {
    double time; string name;
    Event (double t, string n) : time(t), name(n) {}
};
auto cmp = [](const Event& lhs const Event& rhs) {
    return lhs.time > rhs.time;
}
priority_queue<Event, vector<Event>, decltype(cmp)> q(cmp);

// usage example
q.emplace(10.2, "Peter");
q.emplace(2.7, "Mary");
q.emplace(5.3, "John");
q.emplace(0.3, "Bob");
cout << "Events:";
while (!q.empty()) {
    Event& e = q.top();
    cout << " (" << e.time << ',' << e.name << ")";
    q.pop();
}
// Events: (0.3,Bob) (2.7,Mary) (5.3,John) (10.2,Peter)
```

3.1.6 Set & Multiset

```
// references:
// http://www.cplusplus.com/reference/set/set/
// http://www.cplusplus.com/reference/set/multiset/
#define rep(i,a,b) for(int i=a; i<=b; i++)

//================
// INITIALIZATION
//================
// set
set<int> s{1, 2, 3, 4, 4, 5, 5, 5, 2, 2, 2};
for (int x : s) cout << x; // 12345
// multiset
multiset<int> ms{1, 2, 3, 4, 4, 5, 5, 5, 2, 2, 2};
for (int x : s) cout << x; // 12222344555

//========
// INSERT
//========
// set
set<int> s;
rep(i,1,5) s.insert(i*10);    // 10 20 30 40 50
auto ret = s.insert(20);  // no new element inserted
auto it = ret.first;
if (it.second) cout << "20 inserted for the first time\n";
else cout << "20 already in set\n";
int myints[]= {5,10,15};     // 10 already in set, not inserted
s.insert(myints,myints+3);
cout << "s contains:";
for (int x : s) cout << ' ' << x;
cout << '\n'; // 5 10 15 20 30 40 50
// multiset
// ** same as set, but allows duplicates, so insert returns an iterator
//    not a pair

//=======
// ERASE
//=======
// -- set
set<int> s;
rep(i,1,9) s.insert(i*10);  // 10 20 30 40 50 60 70 80 90
auto it = s.begin();
++it; // "it" points now to 20
s.erase(it); // erase by pointer
s.erase(40); // erase by value
it = s.find(60); // iterator pointing to 60
s.erase(it, s.end()); // erase everything in range [it, s.end())
// s = 10 30 50
// -- multiset
multiset<int> ms;
ms.insert (40);                // 40
rep(i,1,6) ms.insert(i*10);    // 10 20 30 40 40 50 60
auto it=ms.begin();
it++;                          //    ^
ms.erase(it);                 // 10 30 40 40 50 60
ms.erase(40);                 // 10 30 50 60
```

```
it=ms.find(50);
ms.erase(it, ms.end());      // 10 30

//=======
// FIND
//=======
// -- set
set<int> s;
rep(i,1,5) s.insert(i*10);  // 10 20 30 40 50
auto it=s.find(20);
s.erase(it);                // 10 30 40 50
s.erase(s.find(40));        // 10 30 50
// -- multiset
// ** same as set

//================================
// lower_bound() & upper_bound()
//================================
// -- set
set<int> s;
rep(i,1,9) s.insert(i*10);    // 10 20 30 40 50 60 70 80 90
auto itlow=s.lower_bound(30); //       ^
auto itup=s.upper_bound(60);  //                   ^
s.erase(itlow,itup);          // 10 20 70 80 90
// -- multiset
multiset<int> ms{30, 10, 10, 40, 30, 90}; // 10 10 30 30 40 90
auto itlow = ms.lower_bound(30);          //        ^
auto itup = ms.upper_bound(40);           //                 ^
ms.erase(itlow,itup);                     // 10 20 90

//=======================
// multiset::equal_range
//=======================
int myints[] = {77,30,16,2,30,30};
multiset<int> ms(myints, myints+6);  // 2 16 30 30 30 77
auto ret = ms.equal_range(30);       //       ^        ^
// ret.first -> first 30 (same as ms.lower_bound(30))
// ret.second -> 77 (same as ms.upper_bound(30))
ms.erase(ret.first, ret.second);     // 2 16 77

//=======
// COUNT
//=======
// --- set
set<int> s{3, 6, 9, 12};
rep(i,0,9) {
    cout << i;
    if (s.count(i) > 0) cout << " is an element of s.\n";
    else cout << " is not an element of s.\n";
}
// --- multiset
multiset<int> ms{10,73,12,22,73,73,12};
cout << ms.count(73); // 3

//=============================
// SET/MULTISET of Custom Data
//=============================
struct CustomData {
    int x; string name;
    CustomData(int x, string n) : x(x), name(n) {}
    // define operator <
    bool operator<(const CustomData& rhs) const {
        return x < rhs.x;
    }
};
set<CustomData> s;
multiset<CustomData> ms;
s.emplace(1, "foo");
s.emplace(2, "bar");
ms.emplace(-12, "bla");
```

3.1.7 Map & Multimap

```
// references:
// http://www.cplusplus.com/reference/map/map/
// http://www.cplusplus.com/reference/map/multimap/
// ** SUMMARY **
//  same as set and multiset, except that for each key
//  now there is a value associated to it (if we only consider
//  the keys is the same as set/multiset)

//================
// INITIALIZATION
//================
// --- map
map<string,float> m {{"a",1.50}, {"b",2.10}, {"c",1.40}};
// or
map<string,float> m;
m.emplace("a", 1.50);
m.emplace("b", 2.10);
m.emplace("c", 1.40);
// --- multimap
// ** same as map

//=========
// INSERT
//=========
// --- map
map<char,int> m;
// first insert function version (single parameter):
m.insert( pair<char,int>('a',100) );
m.insert( pair<char,int>('z',200) );
auto ret = m.insert ( pair<char,int>('z',500) );
if (ret.second==false) {
    cout << "element 'z' already existed";
    cout << " with a value of " << ret.first->second << '\n';
}
// second insert function version (with hint position):
```

```cpp
auto it = m.begin();
m.insert(it, pair<char,int>('b',300));  // max efficiency inserting
m.insert(it, pair<char,int>('c',400));  // no max efficiency inserting
// third insert function version (range insertion):
map<char,int> m2;
m2.insert(m.begin(), m.find('c'));
// showing contents:
cout << "m contains:\n";
for (auto& kv : m) cout << kv.first << " => " << kv.second << '\n';
cout << "m2 contains:\n";
for (auto& kv : m2) cout << kv.first << " => " << kv.second << '\n';
/*
element 'z' already existed with a value of 200
m contains:
a => 100
b => 300
c => 400
z => 200
m2 contains:
a => 100
b => 300
 */
// --- multimap
// ** same as map

//=================
// map::operator[]
//=================
map<char,string> m;
m['a']="an element";
m['b']="another element";
m['c']=m['b'];
cout << "m['a'] is " << m['a'] << '\n';
cout << "m['b'] is " << m['b'] << '\n';
cout << "m['c'] is " << m['c'] << '\n';
cout << "m['d'] is " << m['d'] << '\n'; // ('d' -> "") is created by default
cout << "m now contains " << m.size() << " elements.\n";
/*
m['a'] is an element
m['b'] is another element
m['c'] is another element
m['d'] is
m now contains 4 elements.
*/

//=================
// map::operator=
//=================
map<char,int> first;
map<char,int> second;
first['x']=8;
first['y']=16;
first['z']=32;
second=first;           // second now contains 3 ints
first=map<char,int>();  // and first is now empty
cout << "Size of first: " << first.size() << '\n';
cout << "Size of second: " << second.size() << '\n';

//=========================
// 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;
}
```

3.1.8 Unordered Set & Multiset

```cpp
// references:
// http://www.cplusplus.com/reference/unordered_set/unordered_set/
// http://www.cplusplus.com/reference/unordered_set/unordered_multiset/
// ** unordered_multiset is basically the same as unordered_set
//    except that unordered_multiset allows duplicate elements

//=========
// RESERVE
//=========
unordered_set<string> s;
s.reserve(5);
s.insert("office");
s.insert("house");
s.insert("gym");
s.insert("parking");
s.insert("highway");
cout << "s contains:";
for (const string& x: s) cout << " " << x;
cout << '\n'; // s contains: highway house office gym parking
// By calling reserve with the size we expected for the unordered_set
// container we avoided the multiple rehashes that the increases in container
// size could have produced and optimized the size of the hash table.

//========
// INSERT
//========
unordered_set<string> s = {"yellow","green","blue"};
array<string,2> arr = {"black","white"};
string mystring = "red";
s.insert(mystring);                 // copy insertion
s.insert(mystring+"dish");          // move insertion
s.insert(arr.begin(), arr.end());   // range insertion
s.insert( {"purple","orange"} );    // initializer list insertion
cout << "s contains:";
for (const string& x: s) cout << " " << x;
cout << '\n';
```

```
//s contains: green blue reddish white yellow black red orange purple

//=======
// ERASE
//=======
unordered_set<string> s =
{"USA","Canada","France","UK","Japan","Germany","Italy"};
s.erase( s.begin() );                 // erasing by iterator
s.erase( "France" );                  // erasing by key
s.erase( s.find("Japan"), s.end() ); // erasing by range
cout << "s contains:";
for ( const string& x: s ) cout << " " << x;
cout << '\n'; // s contains: Canada USA Italy

//======
// FIND
//======
unordered_set<string> s{"red","green","blue"};
auto it = s.find("black");
assert (it == s.end());
assert (s.find("red") != s.end());

//=======
// COUNT
//=======
unordered_set<string> s { "hat", "umbrella", "suit" };
for (auto& x: {"hat","sunglasses","suit","t-shirt"}) {
    if (s.count(x) > 0) cout << "s has " << x << '\n';
    else cout << "s has no " << x << '\n';
} /*
s has hat
s has no sunglasses
s has suit
s has no t-shirt */
```

3.1.9 Unordered Map & Multimap

```
// references:
// http://www.cplusplus.com/reference/unordered_map/unordered_map/
// http://www.cplusplus.com/reference/unordered_map/unordered_multimap/
// ** SUMMARY **
//  same as unordered_set and unordered_multiset, except that for each key
//  now there is a value associated to it (if we only consider
//  the keys is the same as unordered_set/unordered_multiset)

//================
// INITIALIZATION
//================
// --- unordered_map
unordered_map<string,float> m {{"a",1.50}, {"b",2.10}, {"c",1.40}};
// or
unordered_map<string,float> m;
m.emplace("a", 1.50);
m.emplace("b", 2.10);
m.emplace("c", 1.40);
// --- unordered_multimap
// ** same as unordered_map

//=========
// INSERT
//=========
// --- unordered_map
unordered_map<string,double>
            myrecipe,
            mypantry = {{"milk",2.0},{"flour",1.5}};
pair<string,double> myshopping("baking powder",0.3);
myrecipe.insert(myshopping);                        // copy insertion
myrecipe.insert(make_pair("eggs",6.0));             // move insertion
myrecipe.insert(mypantry.begin(), mypantry.end()); // range insertion
myrecipe.insert( {{"sugar",0.8},{"salt",0.1}} );   // initializer list insertion
cout << "myrecipe contains:" << '\n';
for (auto& x: myrecipe) cout << x.first << ": " << x.second << '\n';
cout << '\n';/*
myrecipe contains:
salt: 0.1
eggs: 6
sugar: 0.8
baking powder: 0.3
flour: 1.5
milk: 2 */
// --- unordered_multimap
// ** same as unordered_map

//==========================
// unordered_map::operator[]
//===========================
unordered_map<string,string> m;
m["Bakery"]="Barbara";  // new element inserted
m["Seafood"]="Lisa";    // new element inserted
m["Produce"]="John";    // new element inserted
string name = m["Bakery"];   // existing element accessed (read)
m["Seafood"] = name;         // existing element accessed (written)
m["Bakery"] = m["Produce"];  // existing elements accessed (read/written)
name = m["Deli"];      // non-existing element: new element "Deli" inserted!
m["Produce"] = m["Gifts"];    // new element "Gifts" inserted, "Produce" written
for (auto& x: m) cout << x.first << ": " << x.second << '\n';
/*
Seafood: Barbara
Deli:
Bakery: John
Gifts:
Produce:
*/

//==========================
// unordered_map::operator=
//==========================
typedef unordered_map<string,string> stringmap;
```

```cpp
stringmap merge (stringmap a,stringmap b) {
    stringmap temp(a); temp.insert(b.begin(),b.end()); return temp;
}
int main() {
    stringmap first, second, third;
    first = {{"AAPL","Apple"},{"MSFT","Microsoft"}};  // init list
    second = {{"GOOG","Google"},{"ORCL","Oracle"}};   // init list
    third = merge(first,second);                      // move
    first = third;                                    // copy
    cout << "first contains:";
    for (auto& x: first) cout << " " << x.first << ":" << x.second;
    cout << '\n';
    return 0;
}
// first contains: MSFT:Microsoft AAPL:Apple GOOG:Google ORCL:Oracle
```

3.1.10 Deque

```cpp
// references:
//   http://www.cplusplus.com/reference/deque/deque/
//   https://www.geeksforgeeks.org/deque-cpp-stl/
// SUMMARY: deque can do the same things as vector
//  + push_front() + emplace_front()
//  - contiguous memory allocation is not guaranteed
//  (elements may be stored in fragmented chunks of memory)
deque<int> dq = { 1, 2, 3 };
dq.push_back(8); // { 1, 2, 3, 8 }
dq.push_front(100); // { 100, 1, 2, 3, 8 }
dq.pop_back(); // { 100, 1, 2, 3 }
dq.pop_front(); // { 1, 2, 3}
```

3.1.11 List

```cpp
// full documentation:
// http://www.cplusplus.com/reference/list/list/
// https://www.geeksforgeeks.org/list-cpp-stl/

//========
// INSERT
//========
// http://www.cplusplus.com/reference/list/list/insert/

list<int> mylist;
list<int>::iterator it;
// set some initial values:
rep(i,1,5) mylist.push_back(i);               // 1 2 3 4 5
it = mylist.begin();
++it;       // it points now to number 2           ^
mylist.insert(it,10);                         // 1 10 2 3 4 5
// "it" still points to number 2                      ^
mylist.insert (it,2,20);                      // 1 10 20 20 2 3 4 5
--it;       // it points now to the second 20            ^
vector<int> myvector (2,30);
mylist.insert (it,myvector.begin(),myvector.end());
                                            // 1 10 20 30 30 20 2 3 4 5
                                            //                ^
cout << "mylist contains:";
for (int x : mylist) cout << ' ' << x;
cout << '\n';
// mylist contains: 1 10 20 30 30 20 2 3 4 5

//=======
// ERASE
//=======
// http://www.cplusplus.com/reference/list/list/erase/

list<int> mylist;
list<int>::iterator it1,it2;
// set some values:
rep(i,1,9) mylist.push_back(i*10);
                            // 10 20 30 40 50 60 70 80 90
it1 = it2 = mylist.begin(); // ^^
advance (it2,6);            // ^                 ^
++it1;                      //    ^              ^

it1 = mylist.erase(it1);    // 10 30 40 50 60 70 80 90
                            //    ^           ^

it2 = mylist.erase(it2);    // 10 30 40 50 60 80 90
                            //    ^           ^
++it1;                      //       ^        ^
--it2;                      //       ^     ^
mylist.erase(it1,it2);      // 10 30 60 80 90
                            //       ^
cout << "mylist contains:";
for (int x : mylist) cout << ' ' << x;
cout << '\n';
// mylist contains: 10 30 60 80 90
```

3.1.12 Policy based Data Structures: Ordered Set

```cpp
// references:
// https://www.geeksforgeeks.org/ordered-set-gnu-c-pbds/
// https://www.geeksforgeeks.org/policy-based-data-structures-g/
// https://codeforces.com/blog/entry/11080
#include <bits/stdc++.h>
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

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

```cpp

int main() {
    ordered_set o_set;
    o_set.insert(5);
    o_set.insert(1);
    o_set.insert(2);
    // Finding the second smallest element
    // in the set using * because
    //  find_by_order returns an iterator
    cout << *(o_set.find_by_order(1)) << '\n';
    // Finding the number of elements
    // strictly less than k=4
    cout << o_set.order_of_key(4) << '\n';
    // Finding the count of elements less
    // than or equal to 4 i.e. strictly less
    // than 5 if integers are present
    cout << o_set.order_of_key(5) << '\n';
    // Deleting 2 from the set if it exists
    if (o_set.find(2) != o_set.end())
        o_set.erase(o_set.find(2));
    // Now after deleting 2 from the set
    // Finding the second smallest element in the set
    cout << *(o_set.find_by_order(1)) << '\n';
    // Finding the number of
    // elements strictly less than k=4
    cout << o_set.order_of_key(4) << '\n';
    return 0;
}
```

3.1.13 Bitset

```cpp
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
```

3.2 Sparse Tables

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

// time complexity:
//  - filling DP table: O(N log N)
//  - answering queries: O(1) / O(log N)

struct SparseTable {
    int n;
    vector<int> memo;
    vector<int>* arr;
    SparseTable(vector<int>& _arr) {
        arr = &_arr;
        n = arr->size();
        int maxlog = 31 - __builtin_clz(n);
        memo.assign(n * (maxlog + 1), -1);
    }
    // dp(i,e) = min { arr[j] } for j in {i, i+1, ..., i+2^e-1}
    int dp(int i, int e) {
        int& ans = memo[e * n + i];
        if (ans != -1) return ans;
        if (e == 0) return ans = (*arr)[i];
        return ans = min(dp(i, e-1), dp(i+(1<<(e-1)), e-1));
    }

    // ---- RMQ = Range Minimun Query ----
    // rmq(l,r) = min { arr[j] } for j in {l, l+1, ..., r}

    // option 1: complexity O(1)
    int rmq_O1(int l, int r) {
        int e = 31 - __builtin_clz(r - l + 1);
        return min(dp(l,e), dp(r - (1 << e) + 1, e));
    }

    // option 2: complexity O(log N)
    int rmq_Ologn(int l, int r) {
        int ans = INT_MAX;
        int d = r-l+1;
        for (int e = 0; d; e++, d>>=1) {
            if (d & 1) {
                ans = min(ans, dp(l, e));
                l += 1 << e;
            }
        }
        return ans;
    }
};

// example of usage
int main() {
    vector<int> arr = {1, 3, 4, 3, 1, 6, 7, 4, 8, 9};
    SparseTable st(arr);
    while (true) {
        int l, r; cin >> l >> r; // read query
        cout << st.rmq_O1(l,r) << '\n'; // print minimum
    }
    return 0;
}
```

3.3 Fenwick Tree

```cpp
struct BIT { // BIT = binary indexed tree (a.k.a. Fenwick Tree)
    vector<int> bit;
    BIT(int n) { bit.assign(n+1, 0); }
    // prefix sum query (sum in range 1 .. k)
    int psq(int k) {
        int sum = 0;
```

```
        for (; k; k -= (k & -k)) sum += bit[k];
        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 < bit.size(); k += (k & -k)) bit[k] += v;
    }
};
```

3.4 Fenwick Tree 2D

```
struct BIT2D { // BIT = binary indexed tree (a.k.a. Fenwick Tree)
    vector<int> bit;
    int R, C;
    BIT2D(int _R, int _C) : R(_R+1), C(_C+1) {
        bit.assign(R*C, 0);
    }
    void add(int r, int c, int value) {
        for (int i = r; i < R; i += (i&-i))
            for (int j = c; j < C; j += (j&-j))
                bit[i * C + 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 += bit[i * C + 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);
    }
    // get value at cell (r,c)
    int get(int r, int c) {
        return sum(r, c, r, c);
    }
    // set value to cell (r,c)
    int set(int r, int c, int value) {
        add(r, c, -get(r, c) + value);
    }
};
```

3.5 Segment Tree

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

//=============================
// 1) Segment Tree - ITERATIVE
//=============================
// source: https://docs.google.com/document/d/1rcex_saP4tExbbU62qGUjR3eenxOh-50
    i9Y45WtHkc4/
/*
Se requiere un struct para el nodo (ej: prodsgn).
Un nodo debe tener tres constructores:
    Aridad 0: Construye el neutro de la operacion
    Aridad 1: Construye un nodo hoja a partir del input
    Aridad 2: Construye un nodo segun sus dos hijos

Construccion del segment tree:
    Hacer un arreglo de nodos (usar ctor de aridad 1).
    ST<miStructNodo> miSegmentTree(arregloDeNodos);
Update:
    miSegmentTree.set_point(indice, miStructNodo(input));
Query:
    miSegmentTree.query(l, r) es inclusivo exclusivo y da un nodo. Usar la info del nodo
        para obtener la respuesta.
 */
template<class node> struct ST {
    vector<node> t; int n;
    ST(vector<node> &arr) {
        n = arr.size();
        t.resize(n*2);
        copy(arr.begin(), arr.end(), t.begin() + n);
        for (int i = n-1; i > 0; --i)
            t[i] = node(t[i<<1], t[i<<1|1]);
    }
    // 0-indexed
    void set_point(int p, const node &value) {
        for (t[p += n] = value; p > 1; p >>= 1)
            t[p>>1] = node(t[p], t[p^1]);
    }
    // inclusive exclusive, 0-indexed
    node query(int l, int r) {
        node ansl, ansr;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l&1) ansl = node(ansl, t[l++]);
            if (r&1) ansr = node(t[--r], ansr);
        }
        return node(ansl, ansr);
    }
};

// Interval Product (LiveArchive)
struct prodsgn {
    int sgn;
    prodsgn() {sgn = 1;}
    prodsgn(int x) { sgn = (x > 0) - (x < 0); }
    prodsgn(const prodsgn &a, const prodsgn &b) { sgn = a.sgn*b.sgn; }
};

```

```
// Maximum Sum (SPOJ)
struct maxsum {
    int first, second;
    maxsum() {first = second = -1;}
    maxsum(int x) { first = x; second = -1; }
    maxsum(const maxsum &a, const maxsum &b) {
        if (a.first > b.first) {
            first = a.first;
            second = max(a.second, b.first);
        } else {
            first = b.first; second = max(a.first, b.second);
        }
    }
    int answer() { return first + second; }
};

// Range Minimum Query
struct rminq {
    int value;
    rminq() {value = INT_MAX;}
    rminq(int x) {value = x;}
    rminq(const rminq &a, const rminq &b) {
        value = min(a.value, b.value);
    }
};

//=============================
// 2) Segment Tree - RECURSIVE
//=============================

template<class t> class ST {
    vector<ll> *arr, st; int n;

    void build(int u, int i, int j) {
        if (i == j) {
            st[u] = (*arr)[i];
            return;
        }
        int m = (i+j)/2, l = u*2+1, r = u*2+2;
        build(l, i, m);
        build(r, m+1, j);
        st[u] = t::merge_op(st[l], st[r]);
    }

    ll query(int a, int b, int u, int i, int j) {
        if (j < a or b < i) return t::neutro;
        if (a <= i and j <= b) return st[u];
        int m = (i+j)/2, l = u*2+1, r = u*2+2;
        ll x = query(a, b, l, i, m);
        ll y = query(a, b, r, m+1, j);
        return t::merge_op(x, y);
    }

    void update(int a, ll value, int u, int i, int j) {
        if (j < a or a < i) return;
        if (i == j) st[a] += value;
        else {
            int m = (i+j)/2, l = u*2+1, r = u*2+2;
            update(a, value, l, i, m);
            update(a, value, r, m+1, j);
            st[u] = t::merge_op(st[l], st[r]);
        }
    }

public:
    ST(vector<ll>& v) {
        arr = &v;
        n = v.size();
        st.resize(n*4+5);
        build(0, 0, n-1);
    }

    ll query(int a, int b) {
        return query(a, b, 0, 0, n-1);
    }

    void update(int a, ll value) {
        update(a, value, 0, 0, n-1);
    }
};

struct RSQ { // range sum query
    static ll const neutro = 0;
    static ll merge_op(ll x, ll y) { return x + y; }
};

struct RMinQ { // range minimun query
    static ll const neutro = LLONG_MAX;
    static ll merge_op(ll x, ll y) { return min(x, y); }
};

struct RMaxQ { // range maximum query
    static ll const neutro = LLONG_MIN;
    static ll merge_op(ll x, ll y) { return max(x, y); }
};

// usage
int main() {
    vector<int> A = { 18, 17, 13, 19, 15, 11, 20 };
    ST<RSQ> stl(A);
    stl.update(2, 100);
    stl.query(1, 3);
    return 0;
}
```

3.6 Segment Tree Lazy

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

```
using namespace std;
typedef long long int ll;

template<class t> class SegTreeLazy {
    vector<ll> *arr, st, lazy; int n;

    void build(int u, int i, int j) {
        if (i == j) {
            st[u] = (*arr)[i];
            return;
        }
        int m = (i+j)/2, l = u*2+1, r = u*2+2;
        build(l, i, m);
        build(r, m+1, j);
        st[u] = t::merge_op(st[l], st[r]);
    }

    void propagate(int u, int i, int j, ll x) {
        st[u] = t::range_op(st[u], i, j, x);
        if (i != j) {
            lazy[u*2+1] = t::prop_left_op(lazy[u*2+1], x);
            lazy[u*2+2] = t::prop_right_op(lazy[u*2+2], x);
        }
        lazy[u] = 0;
    }

    ll query(int a, int b, int u, int i, int j) {
        if (j < a or b < i) return t::neutro;
        if (lazy[u]) propagate(u, i, j, lazy[u]);
        if (a <= i and j <= b) return st[u];
        int m = (i+j)/2, l = u*2+1, r = u*2+2;
        ll x = query(a, b, l, i, m);
        ll y = query(a, b, r, m+1, j);
        return t::merge_op(x, y);
    }

    void update(int a, int b, ll value, int u, int i, int j) {
        if (lazy[u]) propagate(u, i, j, lazy[u]);
        if (a <= i and j <= b) propagate(u, i, j, value);
        else if (j < a or b < i) return; else {
            int m = (i+j)/2, l = u*2+1, r = u*2+2;
            update(a, b, value, l, i, m);
            update(a, b, value, r, m+1, j);
            st[u] = t::merge_op(st[l], st[r]);
        }
    }

public:
    SegTreeLazy(vector<ll>& v) {
        arr = &v;
        n = v.size();
        st.resize(n*4+5);
        lazy.assign(n*4+5, 0);
        build(0, 0, n-1);
    }

    SegTreeLazy(int64_t n) {
      arr = new vector<ll>(4 * n);
      this->n = n;
      st.resize(n*4+5);
      lazy.assign(n*4+5, 0);
      build(0, 0, n-1);
    }

    ll query(int a, int b) {
        return query(a, b, 0, 0, n-1);
    }

    void update(int a, int b, ll value) {
        update(a, b, value, 0, 0, n-1);
    }
};

struct RSQ { // range sum query
    static ll const neutro = 0;
    static ll merge_op(ll x, ll y) { return x + y; }
    static ll range_op(ll st_u, int i, int j, ll x) { return st_u + (j - i + 1) * x; }
    static ll prop_left_op(ll left_child, ll x) { return left_child + x; }
    static ll prop_right_op(ll right_child, ll x) { return right_child + x; }
};

struct RMinQ { // range minimum query
    static ll const neutro = LLONG_MAX;
    static ll merge_op(ll x, ll y) { return min(x, y); }
    static ll range_op(ll st_u, int a, int b, ll x) { return st_u + x; }
    static ll prop_left_op(ll left_child, ll x) { return left_child + x; }
    static ll prop_right_op(ll right_child, ll x) { return right_child + x; }
};

struct RMaxQ { // range maximum query
    static ll const neutro = LLONG_MIN;
    static ll merge_op(ll x, ll y) { return max(x, y); }
    static ll range_op(ll st_u, int a, int b, ll x) { return st_u + x; }
    static ll prop_left_op(ll left_child, ll x) { return left_child + x; }
    static ll prop_right_op(ll right_child, ll x) { return right_child + x; }
};

// usage
int main() {
    vector<ll> A = { 18, 17, 13, 19, 15, 11, 20 };
    SegTreeLazy<RSQ> stl(A);
    stl.update(1, 5, 100);
    stl.query(1, 3);
    return 0;
}
```

3.7 Union-Find

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

struct UnionFind {
    vector<int> 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)]; }
};
```

4 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; // m = (i+j) / 2;
        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
}

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

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

#include <bits/stdc++.h>

int main () {
    vector<int> v{10,20,30,30,20,10,10,20};          // 10 20 30 30 20 10 10 20
    sort (v.begin(), v.end());                       // 10 10 10 20 20 20 30 30
    auto low = lower_bound (v.begin(), v.end(), 20); //          ^
    auto up  = upper_bound (v.begin(), v.end(), 20); //                   ^
    cout << "lower_bound at position " << (low- v.begin()) << '\n';
    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);

//=====================
// std::binary_search()
```

```
//======================
bool myfunction (int i,int j) { return (i<j); }
std::vector<int> v{1,2,3,4,5,4,3,2,1};
sort(v.begin(), v.end());
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;
}
```

5 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;
```

6 Dynamic Programming

6.1 Longest Increasing Subsequence

```
// ====================================
// LIS (Longest Increasing Subsequence)
// ====================================
// references:
// https://stackoverflow.com/questions/2631726/how-to-determine-the-longest-increasing-
    subsequence-using-dynamic-programming
const int MAXLEN = 1000000;
// return the length of the longest increasing (non-decreasing)
// subsequence in values
int LIS(vector<int>& values) {
    static int q[MAXLEN+1];
    int len = 0;
    q[0] = -INT_MAX; // make sure it's strictly smallest
    for (int val : values) {
        if (q[len] < val) { // use <= if non-decreasing
            q[++len] = val;
        } else {
            int l=1, r=len;
            while (l<r) {
                int m = (l+r)>>1;
                if (q[m] >= val) { // use > if non-decreasing
                    r = m;
                } else {
                    l = m+1;
                }
            }
            q[l] = val;
        }
    }
    return len;
}
```

6.2 Travelling Salesman Problem

```
// ---------------------------------
// Travelling Salesman Problem (TSP)
// ---------------------------------
// complexity: O(2^N * N)

const int MAXN = 14; // maximum number of nodes in the problem statement
int cost[MAXN][MAXN]; // cost[i][j]: cost to travel from node i to node j
// make sure cost[i][j] >= 0
int start_index; // OPTIONAL: if you need to remember the start node
```

```

// dp(mask, i): find the minimum cost of visiting all nodes indicated by 'mask'
// starting from node 'i'.
//  *** OPTIONAL VARIANT: include the cost of returning back to the start node at the end
//
//   * mask: an int whose bits indicate the nodes we want to visit
//      ** if j-th bit in mask is 1, the j-th node should be visited
//         else, the j-th node should be ignored
//
//   * i: node we are starting the travel from (i'th bit should be 1 in mask)
int memo[1 << MAXN][MAXN]; // 2^MAXN x MAXN
int dp(int mask, int i) {
    // base case 1: problem already solved
    int& ans = memo[mask][i];
    if (ans != -1) return ans;

    // mark i-th node as visited
    int mask2 = mask & ~(1 << i);

    // base case 2: nothing else to visit
    if (mask2 == 0) return ans = 0;
    // if (mask2 == 0) return ans = cost[i][start_index]; // <--- if returning back to
         start

    // general case: try all possible next nodes
    ans = INT_MAX;
    for (int j = 0, tmp=mask2; tmp; ++j, tmp>>=1) {
        if (tmp & 1) ans = min(ans, cost[i][j] + dp(mask2, j));
    }

    // return answer
    return ans;
}

int main() { // usage
    memset(memo, -1, sizeof memo);
    start_index = 0;
    cout << dp((1 << N)-1, start_index); // <-- mincost of visiting all N nodes starting
         from 0
}
```

6.3 Knapsack

```
/* ==================== */
/* Knapsack problem : DP */
/* ==================== */

// --------------------------------------
// VARIANT 1: without reposition of items
// --------------------------------------

// ------------------------------
// TOP-DOWN RECURSION (pseudo-code)

function DP(i, c)
    if i == first
        if c >= weight[i] && value[i] > 0 // enough space and worth it
            return value[i]
        else
            return 0
    else
        ans = DP(i-1, c)
        if c >= weight[i] && value[i] > 0 // enough space and worth it
            ans = max(ans, value[i] + DP(i-1, c - weight[i]))
        return ans

// -----------
// BOTTOM-UP

#define MAXN 1000 // max num items
#define MAXC 500 // max capacity
int value[MAXN];
int weight[MAXN];
int memo[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[c] = value[0];
        }
    }
    // other items (i = 1 .. N-1)
    rep (i, 1, N-1) {
        if (value[i] > 0) { // worth it
            invrep(c, C, weight[i]) { // <--- REVERSE ORDER !!
                memo[c] = max(memo[c], value[i] + memo[c - weight[i]]);
            }
        }
    }
    return memo[C];
}

// -----------------------------------
// VARIANT 2: with reposition of items
// -----------------------------------

// ------------------------------
// TOP-DOWN RECURSION (pseudo-code)

function DP(i, c)
    if i == first
        if c >= weight[i] && value[i] > 0 // enough space and worth it
            return value[i]
        else
            return 0
```

```
    else
        ans = DP(i-1, c)
        if c >= weight[i] && value[i] > 0 // enough space and worth it
            ans = max(ans, value[i] + DP(i, c - weight[i])) // << i instead of i-1
        return ans

// -----------
// 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
}
```

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

7 Graphs

7.1 BFS

```
const int MAXN = 1000;
vector<int> g[MAXN]; // graph
int depth[MAXN]; // bfs depth per node
int n; // number of nodes

void bfs(int s) {
    memset(depth, -1, sizeof(int) * n); // init depth with -1
    queue<int> q; q.push(s); // init queue and add 's' (starting node)
    depth[s] = 0; // s will have depth 0
    while (!q.empty()) { // while there are nodes in the queue
        int u = q.front(); q.pop(); // extract the first node 'u' from the queue
        for (int v : g[u]) { // for each neighbor 'v' of 'u'
            if (depth[v] == -1) { // if 'v' has not been visited yet -> visit it
                depth[v] = depth[u] + 1;
                q.push(v);
            }
        }
    }
}

//-----------------------------
// Finding connected components
//-----------------------------

int count_cc() {
    static bool visited[MAXN];
    int count = 0;
    memset(visited, 0, sizeof(bool)*n);
    queue<int> q;
    rep(i,0,n-1) {
        if (!visited[i]) {
            count++;
            visited[i] = true;
            q.push(i);
            while (!q.empty()) {
                int u = q.front(); q.pop();
                for (int v : g[u]) {
                    if (!visited[v]) {
                        visited[v] = true;
                        q.push(v);
                    }
                }
            }
        }
    }
    return count;
}
```

7.2 DFS

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

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

//recursive, using depth
int depth[MAXN];
void dfs(int u, int d) {
    depth[u] = d;
    for(int v : g[u]) {
        if(depth[v] == -1) { // not visited yet
            dfs(v, d+1);
        }
    }
}

//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);
            }
        }
    }
}

//-----------------------------
// Finding connected components
```

```cpp
//-----------------------------
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
const int 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;
}
```

7.3 TopoSort

```cpp
typedef vector<int> vi;

// ---------------------------
// option 1: tarjan's algorithm
// ---------------------------
// Note: nodes are sorted in reversed order

vector<vi> g; // graph
int n; // num of nodes
bool visited[MAXN]; // track visited nodes
vi sorted;

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

void topo_sort() {
    memset(visited, false, sizeof(bool) * n);
    sorted.clear();
    rep(i,0,n-1)
        if (!visited[i])
            dfs(i);
}

// --------------------------
// option 2: Kahn's algorithm
// --------------------------

vector<vi> g;
int n;
vi indegree;
vi sorted;

void compute_indegree() {
    indegree.assign(n, 0);
    rep(u,0,n-1)
        rep(int v : g[u])
            indegree[v]++;
}

void topoSort() {
    sorted.clear();
    compute_indegree();

    queue<int> q;
    rep(i,0,n-1)
        if (indegree[i] == 0)
            q.push(i);

    while(!q.empty()) {
        int u = q.front(); q.pop();
        sorted.push_back(u);
        for (int v : g[u]) {
            if(--indegree[v] == 0)
                q.push(v);
        }
    }
}
```

7.4 Dijkstra

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

vector<vector<ii>> 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<ii, vector<ii>, greater<ii>> q; // minheap
    q.emplace(0, source);
    while (!q.empty()) {
        ii p = q.top(); q.pop();
        int u = p.second, dist = p.first; // u = node, dist = mindist from source to u
        if (mindist[u] < dist) continue; // skip outdated improvements
        for (ii& e : g[u]) {
            int v = e.second, w = e.first;
            if (mindist[v] > dist + w) {
                mindist[v] = dist + w;
                parent[v] = u;
                q.emplace(mindist[v], v);
            }
        }
    }
}
```

7.5 Minimum Spanning Tree (Kruskal & Prim)

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

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

struct Edge {
    int u, v, cost;
    bool operator<(const Edge& o) const {
        return cost < o.cost;
    }
};
namespace Kruskal {
    struct UnionFind {
        vector<int> p, rank;
        UnionFind(int 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)) {
                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;
    }
}
```

7.6 Lowest Commen Ancestor (LCA)

```
/* =========================== */
/* LCA (Lowest Common Ancestor) */
/* =========================== */
#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)

// 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:
//   - the lca query can be modified to compute querys over the path between 2 nodes
//   - it's possible to append new leaf nodes to the tree

struct LCA {
    vector<int> A, D; // ancestors, depths
    vector<vector<int>> *g; // pointer to graph
    int n, maxe; // num nodes, max exponent
    int& anc(int u, int e) { return A[e * n + u]; }
    int inline log2(int x) { return 31 - __builtin_clz(x); }

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

    LCA(vector<vector<int>>& _g, int root) {
        g = &_g;
        n = _g.size();
        maxe = log2(n);
        D.assign(n, -1);
        A.resize(n * (maxe + 1));
        dfs(root, -1, 0);
        rep(e, 1, maxe) {
            rep (u, 0, n-1) {
                // u's 2^e th ancestor is
                // u's 2^(e-1) th ancestor's 2^(e-1) th ancestor
                int a = anc(u,e-1);
                anc(u,e) = (a == -1 ? -1 : anc(a,e-1));
            }
        }
    }

    // move node u "k" levels up towards the root
    // i.e. find the k-th ancestor of u
    int raise(int u, int k) {
        for (int e = 0; k; e++, k>>=1) if (k&1) u = anc(u,e);
        return u;
    }

    int 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 (e, maxe, 0) {
            // greedily take the biggest 2^e jump possible as long as
            // u and v still remain BELOW the LCA
            if (anc(u,e) != anc(v,e)) {
                u = anc(u,e), v = anc(v,e);
            }
        }
        // the direct parent of u (or v) is lca(u,v)
        return anc(u,0);
    }

    // distance between 'u' and 'v'
    int dist(int u, int v) {
        return D[u] + D[v] - 2 * D[lca(u,v)];
```

```
    }
    // optimized version (in case you already computed lca(u,v))
    int dist(int u, int v, int lca_uv) {
        return D[u] + D[v] - 2 * D[lca_uv];
    }
    // get the node located k steps from 'u' walking towards 'v'
    int kth_node_in_path(int u, int v, int k) {
        int lca_uv = lca(u,v);
        if (D[u] - D[lca_uv] >= k) return raise(u, k);
        return raise(v, dist(u,v,lca_uv) - k);
    }

    int add_child(int p, int u) { // optional
        // add to graph
        (*g)[p].push_back(u);
        // update depth
        D[u] = D[p] + 1;
        // update ancestors
        anc(u,0) = p;
        rep (e, 1, maxe){
            p = anc(p,e-1);
            if (p == -1) break;
            anc(u,e) = 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)


struct LCA {
    vector<int> E, D, H; // E = euler tour, D = depth, H = first index of node in euler
         tour
    vector<int> DP // memo for range minimun query
    vector<vector<int>> *g; // pointer to graph
    int idx; // tracks node ocurrences
    int n; // number of nodes

    int& rmq(int i, int e) { return DP[e * idx + i]; }
    inline int log2(int x) { return 31 - __builtin_clz(x); }

    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;
            }
        }
    }

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

    int 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 e = log2(len);
        int i1 = rmq(l,e);
        int i2 = rmq(r - ((1 << e) - 1), e);
        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[lca(u,v)]];
    }
}

// -----------------
// EXAMPLE OF USAGE
// -----------------
int main() {
    // build graph
```

```
    int n, m;
    scanf("%d%d", &n, &m);
    vector<vector<int>> g(n);
    while (m--) {
        int u, v; scanf("%d%d", &u, &v);
        g[u].push_back(v);
        g[v].push_back(u);
    }
    // init LCA
    LCA lca(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, lca.lca(u,v));
        printf("dist(%d,%d) = %d\n", u, v, lca.dist(u,v));
    }
};
```

7.7 Diameter of a Tree

```
// ==========================
// 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);
}
```

7.8 Articulation Points, Cut Edges, Biconnected Components

```
// -------------------
// Tarjan's Algorithm
// -------------------
//references:
//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
        }
    }
}
```

7.9 Strongly Connected Components

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i=a; i<=b; ++i)
using namespace std;
// ----------------------------------------
// implementation of Tarjan's SCC algorithm
struct tarjanSCC {
    vector<int> _stack, ids, low;
    vector<bool> instack;
    vector<vector<int>>* g;
    int n, ID;
    void dfs(int u) {
        ids[u] = low[u] = ID++;
        instack[u] = true;
        _stack.push_back(u);
        for (int v : (*g)[u]) {
            if (ids[v] == -1) {
                dfs(v);
                low[u] = min(low[v], low[u]);
            } else if (instack[v]) {
                low[u] = min(low[v], low[u]);
            }
        }
        if (low[u] == ids[u]) {
            // u is the root of a SCC
            // ** here you can do whatever you want
            // with the SCC just found
            cout << "SCC found!\n";
            // remove SCC from top of the stack
            while (1) {
                int x = _stack.back(); _stack.pop_back();
                instack[x] = false;
                if (x == u) break;
            }
        }
    }
    tarjanSCC(vector<vector<int>>& _g) {
        g = &_g;
        n = _g.size();
        _stack.reserve(n);
        ids.assign(n, -1);
        low.resize(n);
        instack.assign(n, 0);
        ID = 0;
        rep(u, 0, n-1) if (ids[u] == -1) dfs(u);
    }
};

// example of usage
int main() {
    // read and build graph from standard input
    int n, m; cin >> n >> m;
    vector<vector<int>> g(n);
    while(m--) {
        int u, v; cin >> u >> v; u--, v--;
        g[u].push_back(v);
    }
    // find SCCs
    tarjanSCC tscc(g);
    return 0;
}
```

7.10 Max Flow : Dinic

```
// Time Complexity:
// - general worst case: O (|E| * |V|^2)
// - unit capacities: O( min(V^(2/3), sqrt(E)) * 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);
}
```

8 Mathematics

8.1 Euclidean Algorithm

```
typedef long long int ll;

inline ll mod(ll x, ll m) { return ((x %= m) < 0) ? x+m : x; }

/* ============================ */
/* GCD (greatest common divisor) */
/* ============================ */
// OPTION 1: using C++ builtin function __gcd
__gcd(a,b)
// OPTION 2: manually usings euclid's algorithm
int gcd (ll a, ll b) {
    while (b) { a %= b; swap(a,b); }
    return a;
}

/* ============ */
/* extended GCD */
/* ============ */
// extended euclid's algorithm: find g, x, y such that
// a * x + b * y = g = gcd(a, b)
// The algorithm finds a solution (x0,y0) but there are infinite more:
//   x = x0 + n * (b/g)
//   y = y0 - n * (a/g)
// where n is integer, are the set of all solutions

// --- version 1: iterative
ll gcdext(ll a, ll b, ll& x, ll& y) {
    ll r2, x2, y2, r1, x1, y1, r0, x0, y0, q;
    r2 = a, x2 = 1, y2 = 0;
    r1 = b, x1 = 0, y1 = 1;
    while (r1) {
        q = r2 / r1;
        r0 = r2 % r1;
        x0 = x2 - q * x1;
        y0 = y2 - q * y1;
        r2 = r1, x2 = x1, y2 = y1;
        r1 = r0, x1 = x0, y1 = y0;
    }
    ll g = r2; x = x2, y = y2;
    if (g < 0) g = -g, x = -x, y = -y; // make sure g > 0
    // for debugging (in case you think you might have bugs)
    // assert (g == a * x + b * y);
    // assert (g == __gcd(abs(a),abs(b)));
```

```
    return g;
}

// --- version 2: recursive
ll gcdext(ll a, ll b, ll& x, ll& y) {
    if (a == 0) {
        x = 0, y = 1;
        return b;
    }
    ll x1, y1;
    ll g = gcdext(b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return g;
}

/* ====================== */
/* multiplicative inverse */
/* ====================== */
// find x such that a * x = 1 (mod m)
// this is the same as finding x, y such that
// a * x + m * y = 1, which can be done with gcdext
// and then returning x (mod m)
ll mulinv(ll a, ll m) {
    ll x, y;
    if (gcdext(a, m, x, y) == 1) return mod(x, m); // make sure 0 <= x < m
    return -1; // no inverse exists
}

/* =========================== */
/* Linear Diophantine Equation */
/* =========================== */
// recommended readings:
// http://gauss.math.luc.edu/greicius/Math201/Fall2012/Lectures/linear-diophantine.article.pdf
// http://mathonline.wikidot.com/solutions-to-linear-diophantine-equations

// find intengers x and y such that a * x + b * y = c
bool lindiopeq(ll a, ll b, ll c, ll& x, ll& y) {
    if (a == 0 and b == 0) { // special case
        if (c == 0) { x = y = 0; return true; }
        return false;
    }
    // general case
    ll s, t;
    ll g = gcdext(a,b,s,t);
    if (c % g == 0) {
        x = s*(c/g), y = t*(c/g);
        return true;
    }
    return false;
}

/* ========================== */
/* Linear Congruence Equation */
/* ========================== */
// recommended reading:
// http://gauss.math.luc.edu/greicius/Math201/Fall2012/Lectures/linear-congruences.article.pdf

// find smallest integer x (mod m) that solves the equation
//   a * x = b (mod m)
bool lincongeq(ll a, ll b, ll m, ll& x) {
    assert (m > 0);
    a = mod(a,m);
    b = mod(b,m);
    ll s, t;
    ll g = gcdext(a,m,s,t);
    if (b % g == 0) {
        ll bb = b/g;
        ll mm = m/g;
        ll n = -s*bb/mm;
        x = s*bb + n*mm;
        if (x < 0) x += mm;
        // for debugging
        // assert (0 <= x and x < m);
        // assert (mod(a*x,m) == b);
        return true;
    }
    return false;
}
```

8.2 Primality Test

```
// ==============
// trial division
//================
// complexity: ~O( sqrt(x) )
bool isPrime(int x) {
    for (int d = 2; d * d <= x; d++) {
        if (x % d == 0)
            return false;
    }
    return true;
}

// ====================================
// trial division with precomputed primes
// ====================================
// complexity: ~O( sqrt(x)/log(sqrt(x)) )
//   + time of precomputing primes
bool isPrime(int x, vector<int>& primes) {
    for (int p : primes) {
        if (p*p > x) break;
        if (p % x == 0)
            return false;
    }
    return true;
```

```
}


// =============
// Miller-Rabin
// =============
// complexity: O (k * log^3(n))
// references:
// https://cp-algorithms.com/algebra/primality_tests.html
// https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Complexity
using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};

bool MillerRabin(u64 n) { // returns true if n is probably prime, else returns false.
    if (n < 4)
        return n == 2 || n == 3;

    int s = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        s++;
    }

    for (int i = 0; i < iter; i++) {
        int a = 2 + rand() % (n - 3);
        if (check_composite(n, a, d, s))
            return false;
    }
    return true;
}

bool MillerRabin(u64 n) { // returns true if n is prime, else returns false.
    if (n < 2)
        return false;

    int r = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

8.3 Prime Factorization

```
//=====================
// Prime Factorization
//=====================
// reference: https://cp-algorithms.com/algebra/factorization.html

// method 1: trial division
// complexity: ~ O( sqrt(n) + log_2(n) )
vector<int> trial_division(int n) {
  vector<int> factors;
  for (int d = 2; d*d <= n; d++) {
    while (n % d == 0) {
      factors.push_back(d);
      if ((n /= d) == 1) return factors;
    }
  }
  if (n > 1) factors.push_back(n);
  return factors;
}

// method 2: precomputed primes
// complexity: ~ O( sqrt(n) / log(sqrt(n)) + log_2(n) )
//   + time of precomputing primes
vector<int> trial_division_precomp(int n, vector<int>& primes) {
  vector<int> factors;
  for (int d : primes) {
    if (d*d > n) break;
    while (n % d == 0) {
      factors.push_back(d);
      if ((n /= d) == 1) return factors;
    }
```

```
  }
  if (n > 1) factors.push_back(n);
  return factors;
}

//====================================
// Prime Factorization of Factorials
//====================================
// references:
// http://mathforum.org/library/drmath/view/67291.html
// https://janmr.com/blog/2010/10/prime-factors-of-factorial-numbers/
#define umap unordered_map
umap<int,int> factorial_prime_factorization(int n, vector<int>& primes) {
  umap<int,int> prime2exp;
  for (int p : primes) {
    if (p > n) break;
    int e = 0;
    int tmp = n;
    while ((tmp /= p) > 0) e += tmp;
    if (e > 0) prime2exp[p] = e;
  }
  return prime2exp;
}
```

8.4 Binary modular exponentiation

```
// compute a^b (mod m)
int binary_exp(int a, int b, int m) {
    a %= m;
    int res = 1;
    while (b > 0) {
        if (b&1) res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}
```

8.5 Modular Binomial Coefficient

```
#define rep(i,a,b) for(int i = a; i <= b; ++i)
typedef long long int ll;
const ll MOD = 1000000007ll; // 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

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 choose_mod(int n, int k) {
    if (choose_memo[n][k] != -1) return choose_memo[n][k];
    return choose_memo[n][k] = mul(fac[n], mulinv(mul(fac[k], fac[n-k])));
}
```

8.6 Modular Multinomial Coefficient

```
typedef long long int ll;
const ll MOD = 1000000007ll; // 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;
}
```

8.7 Chinese Remainder Theorem (CRT)

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

inline ll mod(ll x, ll m) { return ((x %= m) < 0) ? x+m : x; }
inline ll mul(ll x, ll y, ll m) { return (x * y) % m; }
inline ll add(ll x, ll y, ll m) { return (x + y) % m; }

// extended euclidean algorithm
// finds g, x, y such that
//    a * x + b * y = g = GCD(a,b)
ll gcdext(ll a, ll b, ll& x, ll& y) {
    ll r2, x2, y2, r1, x1, y1, r0, x0, y0, q;
    r2 = a, x2 = 1, y2 = 0;
    r1 = b, x1 = 0, y1 = 1;
    while (r1) {
        q = r2 / r1;
        r0 = r2 % r1;
        x0 = x2 - q * x1;
        y0 = y2 - q * y1;
        r2 = r1, x2 = x1, y2 = y1;
        r1 = r0, x1 = x0, y1 = y0;
    }
    ll g = r2; x = x2, y = y2;
    if (g < 0) g = -g, x = -x, y = -y; // make sure g > 0
    // for debugging (in case you think you might have bugs)
    // assert (g == a * x + b * y);
    // assert (g == __gcd(abs(a),abs(b)));
    return g;
}

// ==============================================
// CRT for a system of 2 modular linear equations
// ==============================================
// We want to find X such that:
//   1) x = r1 (mod m1)
//   2) x = r2 (mod m2)
// The solution is given by:
//    sol = r1 + m1 * (r2-r1)/g * x' (mod LCM(m1,m2))
// where x' comes from
//   m1 * x' + m2 * y' = g = GCD(m1,m2)
//   where x' and y' are the values found by extended euclidean algorithm (gcdext)
// Useful references:
//   https://codeforces.com/blog/entry/61290
//   https://forthright48.com/chinese-remainder-theorem-part-1-coprime-moduli
//   https://forthright48.com/chinese-remainder-theorem-part-2-non-coprime-moduli
// ** Note: this solution works if lcm(m1,m2) fits in a long long (64 bits)
pair<ll,ll> CRT(ll r1, ll m1, ll r2, ll m2) {
    ll g, x, y; g = gcdext(m1, m2, x, y);
    if ((r1 - r2) % g != 0) return {-1, -1}; // no solution
    ll z = m2/g;
    ll lcm = m1 * z;
    ll sol = add(mod(r1, lcm), m1*mul(mod(x,z),mod((r2-r1)/g,z),z), lcm);
    // for debugging (in case you think you might have bugs)
    // assert (0 <= sol and sol < lcm);
    // assert (sol % m1 == r1 % m1);
    // assert (sol % m2 == r2 % m2);
    return {sol, lcm}; // solution + lcm(m1,m2)
}

// ==============================================
// CRT for a system of N modular linear equations
// ==============================================
//  Args:
//      r = array of remainders
//      m = array of modules
//      n = length of both arrays
//  Output:
//      a pair {X, lcm} where X is the solution of the sytemm
//          X = r[i] (mod m[i]) for i = 0 ... n-1
//      and lcm = LCM(m[0], m[1], ..., m[n-1])
//      if there is no solution, the output is {-1, -1}
// ** Note: this solution works if LCM(m[0],...,m[n-1]) fits in a long long (64 bits)
pair<ll,ll> CRT(ll* r, ll* m, int n) {
    ll r1 = r[0], m1 = m[0];
    rep(i,1,n-1) {
        ll r2 = r[i], m2 = m[i];
        ll g, x, y; g = gcdext(m1, m2, x, y);
        if ((r1 - r2) % g != 0) return {-1, -1}; // no solution
        ll z = m2/g;
        ll lcm = m1 * z;
        ll sol = add(mod(r1, lcm), m1*mul(mod(x,z),mod((r2-r1)/g,z),z), lcm);
        r1 = sol;
        m1 = lcm;
    }
    // for debugging (in case you think you might have bugs)
    // assert (0 <= r1 and r1 < m1);
    // rep(i,0,n-1) assert (r1 % m[i] == r[i]);
    return {r1, m1};
}
```

8.8 Theorems

8.8.1 Pick's Theorem

$A = I + \frac{P}{2} - 1$

9 Geometry

9.1 Geometry 2D Utils

```cpp
#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 */
/* =========================== */
template<typename T>
struct Point<T> { // 2D
    T x, y;
    bool operator==(const Point<T>& p) const { return x==p.x && y == p.y; }
    Point<T> operator+(const Point<T>& p) const { return {x+p.x, y+p.y}; }
    Point<T> operator-(const Point<T>& p) const { return {x-p.x, y-p.y}; }
    Point<T> operator*(T d) const { return {x*d, y*d}; }
    Point<double> cast() { return {(double)x, (double)y}; }
    T norm2() { return x*x + y*y; }
    double norm() { return sqrt(norm2()); }
    T dot(const Point<T>& p) const { return x*p.x + y*p.y; }
    T cross(const Point<T>& p) const { return x*p.y - y*p.x; }
    double angle() {
        double angle = atan2(y, x);
        if (angle < 0) angle += 2 * PI;
        return angle;
    }
    Point<double> 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);
}

/* ========================= */
/* 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;
}

/* ======================== */
/* Line - Line Intersection */
/* ======================== */
ll det(Point& a, Point& b) { return a.x * b.y - a.y * b.x; }
// return whether straight lines <-a1-b1-> and <-a2-b2-> intersect each other
// if they intersect, we assign values to t1 and t2 such that
//    a1 + (b1 - a1) * t1 == a2 + (b2 - a2) * t2
bool find_line_line_intersection(Point& a1, Point& b1, Point& a2, Point& b2,
        double& t1, double& t2) {
    Point d1 = b1 - a1;
    Point d2 = b2 - a2;
    Point _d2 = d2 * -1;
    ll detA = det(d1, _d2);
    if (detA == 0) return false; // parallel lines
    Point b = a2 - a1;
    t1 = (double)det(b, _d2)/(double)detA;
    t2 = (double)det(d1, b)/(double)detA;
    return true;
}


/* =================== */
/* 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 / Line Segment distance */
/* ==================================== */
// reference: https://stackoverflow.com/questions/849211/shortest-distance-between-a-
    point-and-a-line-segment

// 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();
}

// 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 { ll x, y; };
tuple<ll,ll,ll> hash_line(const Point& p1, const Point& p2) {
    ll a = p1.y - p2.y;
    ll b = p2.x - p1.x;
    ll c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
    ll sgn = (a < 0 or (a == 0 and b < 0)) ? -1 : 1;
    ll g = __gcd(abs(a), __gcd(abs(b), abs(c))) * sgn;
    return make_tuple(a/g, b/g, c/g);
}
// task: given 2 points p1 and p2 with integer coords, return a pair {a, b}
// which is unique for all straight lines having the same slope as
// the straight line that goes through p1 and p2
pair<ll,ll> hash_slope(const Point& p1, const Point& p2) {
    ll dx = p2.x - p1.x;
    ll dy = p2.y - p1.y;
    ll sgn = (dx < 0 or (dx == 0 and dy < 0)) ? -1 : 1;
    ll g = __gcd(abs(dx), abs(dy)) * sgn;
    return {dx/g, dy/g};
}
```

9.2 Geometry 3D Utils

```cpp
/* ========================== */
/* Example of Point Definition */
/* ========================== */
struct Point { // 3D
    double x, y, z;
    bool operator==(const Point& p) const { return x==p.x and y==p.y and z==p.z; }
    Point operator+(const Point& p) const { return {x+p.x, y+p.y, z+p.z}; }
    Point operator-(const Point& p) const { return {x-p.x, y-p.y, z-p.z}; }
    Point operator*(double d) const { return {x*d, y*d, z*d}; }
    double norm2() { return x*x + y*y + z*z; }
    double norm() { return sqrt(norm2()); }
    double dot(const Point& p) { return x*p.x + y*p.y + z*p.z; }
    Point cross(Point& p) {
        return {
            y*p.z - z*p.y,
            z*p.x - x*p.z,
            x*p.y - y*p.x
        };
    }
    Point unit() {
        double d = norm();
        return {x/d, y/d, z/d};
    }
    static Point from_sphere_coords(double r, double u, double v) {
        return {
            r*cos(u)*cos(v),
            r*cos(u)*sin(v),
            r*sin(u)
        };
    }
};
// compute angle (0 <= angle <= PI) between vectors a and b
// ** for better performance, the norms can be precomputed
//   or norms can be ommited altogether if a and b are unit vectors
double angle_between(Point& a, Point& b) {
    return acos(a.dot(b)/(a.norm() * b.norm()));
}
// check if point p belongs to the sphere arc from a to b.
// ** this assumes that a and b are points on a sphere centered at (0,0,0),
// and the sphere arc from a to b is the shortest path on the sphere connecting them
const double EPS = 1e-8;
bool point_in_arc(Point& a, Point& b, Point& p) {
    double angle_ab = angle_between(a, b);
    double angle_ap = angle_between(a, p);
    if (angle_ap > angle_ab) return false;
    Point n = a.cross(b);
    Point c_hat = n.cross(a).unit();
    double R = a.norm();
    Point a_hat = a * (1./R);
    Point a_rotated = (a_hat * cos(angle_ap) + c_hat * sin(angle_ap)) * R;
    return (p - a_rotated).norm() < EPS;
}
```

9.3 Polygon Algorithms

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

// ----- Utils ------
const double EPS = 1e-8;
struct Point {
    ll x, 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}; }
    ll cross(const Point& p) const { return x*p.y - y*p.x; }
    ll dot(const Point& p) const { return 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;
}
int orientation(Point& a, Point& b, Point& c) {
    ll tmp = cross(a,b,c);
    return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
}

/* ======================================== */
/* 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/
// ** points must be sorted ccw or cw
double polygon_area(vector<Point>& pol) {
    int n = pol.size()
    double area = 0;
    for (int i = n-1, j = 0; j < n; i = j++) {
        area += (pol[i].x + pol[j].x) * (pol[j].y - pol[i].y);
    }
    return area * 0.5; // use abs(area * 0.5) if points are cw
}

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

// ----------------------------
// 1) Convex Polygons

// 1.1) O(N) method
bool point_in_convexhull(Point& p, vector<Point>& ch) {
    int n = ch.size();
    for (int i=n-1, j=0; j<n; i=j++) {
        if (cross(ch[i], ch[j], p) < 0) return false;
    }
    return true;
}
```

```
// 1.2) O(log N) method
bool point_in_triangle(Point& a, Point& b, Point& c, Point& x) {
    return cross(a, b, x) >= 0 and cross(b, c, x) >= 0 and cross(c, a, x) >= 0;
}
bool point_in_convexhull(Point& p, vector<Point>& ch) {
    if (cross(ch[0], ch[1], p) < 0) return false;
    if (cross(ch[0], ch.back(), p) > 0) return false;
    int l = 2, r = ch.size() - 1;
    while (l < r) {
        int m = (l+r) >> 1;
        if (cross(ch[0], ch[m], p) <= 0) r = m;
        else l = m+1;
    }
    return point_in_triangle(ch[0], ch[l-1], ch[l], p);
}

// ----------------------------------------------
// 2) 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
}

/* ================================ */
/* Find extreme point in Convex Hull */
/* ================================ */
// given two points a and b defining a vector a -> b, and given a convex hull with points
// sorted ccw, find the index in the convex hull of the extreme point.
//  ** the extreme point is the "leftmost" point in the convex hull with respect to the
//     vector a -> b (if there are 2 leftmost points, pick anyone)
int extreme_point_index(Point &a, Point &b, vector<Point> &ch) {
    int n = ch.size();
    Point v = b - a;
    v = Point(-v.y, v.x); // to find the leftmost point
    if (v.dot(ch[0]) >= v.dot(ch[1]) && v.dot(ch[0]) >= v.dot(ch[n - 1])) {
        return 0;
    }
    int l = 0, r = n;
    while (true) {
        int m = (l + r) / 2;
        if (v.dot(ch[m]) >= v.dot(ch[m + 1]) && v.dot(ch[m]) >= v.dot(ch[m - 1])) {
            return m;
        }
        int d1 = v.dot(ch[l + 1] - ch[l]) > 0;
        int d2 = v.dot(ch[m + 1] - ch[m]) > 0;
        int a = v.dot(ch[m]) > v.dot(ch[l]);
        if (d1) { if (d2 && a) l = m; else r = m; }
        else { if (!d2 && a) r = m; else l = m; }
    }
}

/* ======================================== */
/* Line Segment and Convex Hull Intersection */
/* ======================================== */
pair<int,int> find_crossing_edge(Point& a, Point& b, vector<Point>& ch, int start, int
     end) {
    int o_ref = orientation(a, b, ch[start]);
    int n = ch.size();
    int l = start, r = start + ((end - start + n) % n);
    while (l < r) {
        int m = (l+r) >> 1;
        if (orientation(a, b, ch[m % n]) != o_ref) r = m;
        else l = m+1;
    }
    return {(l-1+n) % n, l%n};
}
void find_segment_convexhull_intersection(Point& a, Point& b, vector<Point>& ch) {
    // find rightmost and leftmost points in convex hull wrt vector a -> b
    int i1 = extreme_point_index(a, b, ch);
    int i2 = extreme_point_index(b, a, ch);
    // make sure the extremes are not to the same side
    int o1 = orientation(a, b, ch[i1]);
    int o2 = orientation(a, b, ch[i2]);
    if (o1 == o2) return; // all points are to the right (left) of a -> b (no
         intersection)
```

```
    // find 2 edges in the convex hull intersected by the straight line <- a - b ->
    pair<int,int> e1 = find_crossing_edge(a, b, ch, i1, i2); // binsearch from i1 to i2
        ccw
    pair<int,int> e2 = find_crossing_edge(a, b, ch, i2, i1); // binsearch from i2 to i1
        ccw
    // find exact intersection points
    double r1, s1, r2, s2;
    assert (find_line_line_intersection(a, b, ch[e1.first], ch[e1.second], r1, s1));
    assert (find_line_line_intersection(a, b, ch[e2.first], ch[e2.second], r2, s2));
    // make sure intersections are significant and within line segment range
    if (r1 > 1.0 - EPS and r2 > 1.0 - EPS) return; // intersections above line segment
    if (r1 < EPS and r2 < EPS) return; // intersections below line segment
    if (abs(r1 - r2) < EPS) return; // insignificant intersection in a single point
    if (r1 > r2) swap(r1, r2), swap(e1, e2), swap(s1, s2); // make sure r1 < r2
    // ** HERE DO WHATEVER YOU WANT WITH INTERSECTIONS FOUND
    //   1) a + (b-a) * max(r1, 0)  <--- first point of segment a -> b inside convex hull
    //       if r1 < 0, point a is strictly inside the convex hull
    //   2) a + (b-a) * min(r2, 1)  <--- last point of segment a -> b inside convex hull
    //       if r2 > 1, point b is strictly inside the convex hull
    cout << "(significant) intersection detected!\n";
}
```

9.4 Trigonometry

```
/* ================= */
/* 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));
}
```

9.5 Convex Hull

```
#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) const {
        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;
}
```

9.6 Green's Theorem

```
#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
// 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);
}
```

10 Strings

10.1 Suffix Array

```
// =========================================================
// Suffix Array Construction : Prefix Doubling + Radix Sort
// =========================================================
// Complexity: O(N*log(N))
// references: https://www.cs.helsinki.fi/u/tpkarkka/opetus/10s/spa/lecture11.pdf
// https://youtu.be/_TUeAdu-U_k
#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;

// - the input to the suffix array must be a vector of ints
// - all values in the vector must be >= 1 (because 0 is used
//   as a special value internally)
struct SuffixArray {
    int n;
    vector<int> counts, rank, rank_tmp, sa, sa_tmp;
    vector<int> lcp; // optional: only if lcp is needed
    inline int get_rank(int i) { return i < n ? rank[i]: 0; }
    void counting_sort(int maxv, int k) {
        counts.assign(maxv+1, 0);
        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];
        sa.swap(sa_tmp);
    }
    void compute_sa(vector<int>& s) {
        rep(i,0,n-1) sa[i] = i;
        sort(sa.begin(), sa.end(), [&s](int i, int j) { return s[i] < s[j]; });
        int r = rank[sa[0]] = 1;
        rep(i,1,n-1) rank[sa[i]] = (s[sa[i]] != s[sa[i-1]]) ? ++r : r;
        for (int h=1; h < n and r < n; h <<= 1) {
            counting_sort(r, h);
            counting_sort(r, 0);
            r = rank_tmp[sa[0]] = 1;
            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;
            }
            rank.swap(rank_tmp);
        }
    }
    // LCP construction in O(N) using Kasai's algorithm
    // reference: https://codeforces.com/blog/entry/12796?#comment-175287
    void compute_lcp(vector<int>& s) { // optional: only if lcp array is needed
        lcp.assign(n, 0);
        int k = 0;
        rep(i,0,n-1) {
            int r = rank[i]-1;
            if (r == n-1) { k = 0; continue; }
            int j = sa[r+1];
            while (i+k<n and j+k<n and s[i+k] == s[j+k]) k++;
            lcp[r] = k;
```

```
            if (k) k--;
        }
    }
    SuffixArray(vector<int>& s) {
        n = s.size();
        rank.resize(n); rank_tmp.resize(n);
        sa.resize(n); sa_tmp.resize(n);
        compute_sa(s);
        compute_lcp(s); // optional: only if lcp array is needed
    }
};

int main() { // how to use
    string test; cin >> test;
    vector<int> s;
    for (char c : test) s.push_back(c);
    SuffixArray sa(s);
    for (int i : sa.sa) cout << i << ":\t" << test.substr(i) << '\n';
    rep (i, 0, s.size() - 1) {
        printf("LCP between %d and %d is %d\n", i, i+1, sa.lcp[i]);
    }
}
```

10.2 Trie

```
#include <bits/stdc++.h>
using namespace std;
struct Trie {
    vector<vector<int>> g;
    vector<int> count;
    int vocab;
    Trie(int vocab, int maxdepth = 10000) : vocab(vocab) {
        g.reserve(maxdepth);
        g.emplace_back(vocab, -1);
        count.reserve(maxdepth);
        count.push_back(0);
    }
    int move_to(int u, int c) {
        assert (0 <= c and c < vocab);
        int& v = g[u][c];
        if (v == -1) {
            v = g.size();
            g.emplace_back(vocab, -1);
            count.push_back(0);
        }
        count[v]++;
        return v;
    }
    void insert(const string& s, char ref = 'a') {  // insert string
        int u = 0; for (char c : s) u = move_to(u, c - ref);
    }
    void insert(vector<int>& s) { // insert vector<int>
        int u = 0; for (int c : s) u = move_to(u, c);
    }
    int size() { return g.size(); }
};

// example of usage
int main() {
    Trie trie(26);
    for (string s : {"hell", "hello", "hellyeah", "helpzzzz", "abcdefg"}) {
        cout << "inserting " <<  s << '\n';
        trie.insert(s);
        cout << "\ttrie size = " << trie.size() << '\n';
    }
    return 0;
}
```

10.3 Rolling Hashing

```
#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: single hash

struct RH_single { // rolling hashing
    static const ull B = 131; // base
    static const ull P = 1e9 + 21; // 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;
    void init(vector<int>& s) {
        for (int x : s) assert (x >= 1); // DEBUGGING
        len = s.size();
        h.resize(len);
        h[0] = s[0];
        rep(i,1,len-1) h[i] = add(mul(h[i-1], B), s[i]);
    }
    RH_single(vector<int>& s) { init(s); } // from vector<int>
    RH_single(string& s, char ref) { // from string
        vector<int> tmp; for(char c : s) tmp.push_back(c - ref + 1);
        init(tmp);
    }
    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]; }
```

```cpp
};
ull RH_single::pow[MAXLEN]; // necessary for the code to compile

// ----------------------------------------------
// Rolling Hashing: double hash (extra safety)

struct RH_double { // rolling hashing
    static const ull B = 131; // 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;
    void init(vector<int>& s) {
        for (int x : s) assert (x >= 1); // DEBUGGING
        len = s.size();
        rep(a,0,1) {
            h[a].resize(len);
            h[a][0] = s[0];
            rep(i,1,len-1) h[a][i] = add(mul(h[a][i-1], B, a), s[i], a);
        }
    }
    RH_double(vector<int>& s) { init(s); } // from vector<int>
    RH_double(string& s, char ref) { // from string
        vector<int> tmp; for (char c : s) tmp.push_back(c - ref + 1);
        init(tmp);
    }
    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_double::P[2] = {(ull)1e9+21, (ull)1e9+9};
ull RH_double::pow[2][MAXLEN];

// ----- usage & testing
int main() {
    RH_double::init();
    while (true) {
        string s; cin >> s;
        int l1, r1, l2, r2; cin >> l1 >> r1 >> l2 >> r2;
        char cmin = *min_element(s.begin(), s.end());
        RH_double rh(s, cmin);
        ull h1 = rh.hash(l1, r1);
        ull h2 = rh.hash(l2, r2);
        string s1 = s.substr(l1, r1-l1+1);
        string s2 = s.substr(l2, r2-l2+1);
        printf("Strings s1=%s and s2=%s are %s\n", s1.c_str(), s2.c_str(),
            h1 == h2 ? "EQUAL" : "DISTINCT");
    }
}
```

10.4 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() { // usage
    string target, pattern;
    while (true) {
        cin >> target >> pattern;
        cout << count_matches(pattern, target) << " matches\n";
    }
```

```
    return 0;
}
```

10.5 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;
}
```