Índice

1. C++ Cheat Sheet

```cpp
/* ========= */
// Template  //
/* ========= */
#include <bits/stdc++.h> // add almost everything in one shot
#include <tr1/unordered_map>
#include <tr1/unordered_set>
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 tr1::unordered_map
#define uset tr1::unordered_set

// typedefs
typedef vector<int> vi;
typedef vector<vi> vii;
typedef long long int ll;
typedef pair<int,int> pii;

int main() {
  setvbuf(stdout, NULL, _IONBF, 0);  //debugging
  return 0;
}

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

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

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

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

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

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

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

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

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

/* ====================== */
/* Peek last char of stdin */
/* ====================== */
bool peekAndCheck(char c) {
```

```
  char c2 = getchar();
  ungetc(c2, stdin); // return char to stdin
  return c == c2;
}

/* ================ */
/* Reading from cin */
/* ================ */
// reading a line of unknown length
string line;
getline (cine, name);

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


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

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

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

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

// substrings
string subs = str.substr(pos, length);
string subs = str.substr(pos); // default: to the end of the string

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

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

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

```

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

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

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

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


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

/* ======================= */
/* VECTOR UTILITY FUNCTIONS */
/* ======================= */
std::vector<int> myvector;
myvector.push_back(100);
myvector.pop_back(); // remove last element
myvector.back(); // peek reference to last element
myvector.front(); // peek reference to first element
myvector.clear(); // remove all elements
// sorting a vector
vector<int> foo;
sort (foo.begin(), foo.end());
sort (foo.begin(), foo.end(), std::less<int>()); // increasing
sort (foo.begin(), foo.end(), std::greater<int>()); // decreasing

/* ==================== */
/* MAP UTILITY FUNCTIONS */
/* ==================== */
struct Point {int x, y; };
bool operator<(const Point& a, const Point& b) {
  return a.x < b.x || (a.x == b.x && a.y < b.y);
}
map<Point, int> pt2id;
// --------------------
// inserting into map

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

// method #2: .insert(key, value)
```

```cpp
// it returns a pair { iterator(key, value) , bool }
// if the key already exists, it doesn't overwrite the value
int tid = 0;
while (true) {
  int x,y; scanf("%d %d",&x,&y);
  auto res = pt2id.insert({x,y}, tid);
  int id = res.first->second;
  if (res->second) // insertion happened
    tid++;
}
// -------------------------
// 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;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

```
}

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

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

2. Data Structures

2.1. Fenwick Tree

```
#define LSOne(s) (s & (-s))
struct FenwickTree {
  vector<int> ft;
  FenwickTree(int n) { ft.assign(n+1, 0); }
  int rsq(int b) {
    int sum = 0;
    for (; b; b -= LSOne(b)) sum += ft[b];
    return sum;
  }
  int rsq(int a, int b) {
    return rsq(b) - (a == 1 ? 0 : rsq(a-1));
  }
  void adjust(int k, int v) {
    for (; k < ft.size(); k += LSOne(k)) ft[k] += v;
  }
  void range_adj(int i, int j, int v) {
    adjust(i, v);
    adjust(j+1, -v);
  }
};
```

2.2. Fenwick Tree 2D

```
template<class T> class FenwickTree2D {
  vector<vector<T> > t;
  int n, m;

public:
  FenwickTree2D() {}

  FenwickTree2D(int n, int m) {
    t.assign(n, vector<T>(m, 0));
    this->n = n; this->m = m;
  }

  void add(int r, int c, T value) {
    for (int i = r; i < n; i |= i + 1)
      for (int j = c; j < m; j |= j + 1)
        t[i][j] += value;
  }

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

  // sum[(r1, c1), (r2, c2)]
  T 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);
  }

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

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

2.3. Segment Tree

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

struct SegmentTree {         // the segment tree is stored like a heap array
  vi st, A;
  int n;
  int left (int p) { return p << 1; }     // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                          // O(n log n)
    if (L == R)                            // as L == R, either one is fine
      st[p] = L;                                          // store the index
    else {                                // recursively compute the values
      build(left(p) , L              , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R          );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
    }
```

```
  }

  int rmq(int p, int L, int R, int i, int j) {                  // O(log n)
    if (i >  R || j <  L) return -1; // current segment outside query range
    if (L >= i && R <= j) return st[p];               // inside query range

     // compute the min position in the left and right part of the interval
    int p1 = rmq(left(p) , L             , (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R          , i, j);

    if (p1 == -1) return p2;   // if we try to access segment outside query
    if (p2 == -1) return p1;                                // same as above
    return (A[p1] <= A[p2]) ? p1 : p2; }          // as as in build routine

  int update_point(int p, int L, int R, int idx, int new_value) {
    // this update code is still preliminary, i == j
    // must be able to update range in the future!
    int i = idx, j = idx;

    // if the current interval does not intersect
    // the update interval, return this st node value!
    if (i > R || j < L)
      return st[p];

    // if the current interval is included in the update range,
    // update that st[node]
    if (L == i && R == j) {
      A[i] = new_value; // update the underlying array
      return st[p] = L; // this index
    }

    // compute the minimum position in the
    // left and right part of the interval
    int p1, p2;
    p1 = update_point(left(p) , L             , (L + R) / 2, idx, new_value);
    p2 = update_point(right(p), (L + R) / 2 + 1, R          , idx, new_value);

    // return the position where the overall minimum is
    return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  }

  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();              // copy content for local usage
    st.assign(4 * n, 0);            // create large enough vector of zeroes
    build(1, 0, n - 1);                                 // recursive build
  }

  int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }   // overloading

  int update_point(int idx, int new_value) {
    return update_point(1, 0, n - 1, idx, new_value); }
};

// usage
```

```
int main() {
  vi A = { 18, 17, 13, 19, 15, 11, 20 };
  SegmentTree st(A);
  st.rmq(1,3);
  st.update_point(5, 100);
  return 0;
}
```

2.4. Segment Tree Lazy

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

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

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

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

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

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

```
    }

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

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

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

};

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

2.5. Wavelet Tree

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

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

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

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

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

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

  // counts numbers in [x,y] in positions [a,b)
```

```
int range(int x, int y, int a, int b) {
  if (y < x or b <= a)
    return 0;
  q = x; w = y;
  return range(a, b, 0, s-1, 1);
}

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

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

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

//swap arr[i] with arr[i+1], i in [0,n-1)
void swap_adj(int i) {
  int &x = arrCopy[i], &y = arrCopy[i+1];
  int l = 0, r = s-1, u = 1;
  while (l != r) {
    int m = (l+r)/2, p = (x<=m), q = (y<=m);
    if (p != q) {
      r0[u][i+1] ^= r0[u][i] ^ r0[u][i+2];
      break;
    }
    u*=2; if (p) r = m; else l = m+1, ++u;
  }
  swap(x, y);
 }
};
```

2.6. Union-Find

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

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

3. General Algorithms

3.1. Binary Search

```
// Find the index of the first item that satisfies a predicate.
// If no such index exists, retuns -1
// Pseudo-code:
function binsearch(array, i, j) {
  while (i < j) {
    m = (i+j)/2
    if (predicate(array[m]))
      j = m
    else
      i = m + 1
  }
```

```
  return (predicate(array[i]) ? i : -1)
}

// ----------------------------
// 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 a[i] >= key ? i : -1;
}

// ----------------------------
// 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 a[i] > key ? i : -1;
}

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

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

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

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

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

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

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

  return 0;
}

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

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

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

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

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


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

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

3.2. Ternary Search

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

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

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

3.3. Brute Force

3.3.1. Generate all combinations

```
/* ======================== */
/*  Try all 2^n combinations */
/* ======================== */

void all_combs(vector<int> items) {
  int n = vals.size();
  int times = (1 << n);
  vector<int> comb(n, 0)

  while(times-- > 0) {

    do_something(comb)

    // generate next combination
    int i = 0, carry = 1;
    while (i < n) {
      in[i] += carry;
      if (in[i] <= 1)
        carry = 0;
      else
        in[i] = 0;
      // do something with i'th item
      i++;
    }
  }
}
```

4. Dynamic Programming

4.1. 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
}
```

5. Graphs

5.1. Breadth-First Search

```
/* ========================== */
/* BFS (Breadth First Search) */
/* ========================== */

#include <queue>
#include <stack>
#include <vector>
#define MAXN 1000

typedef vector<int> vi;
vector<vi> g; // graph
vi depth; // bfs depth per node
int N; // num of nodes

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


// ==========================
// Find Tree's Diameter Ends
// ==========================

#include <cstring>
#include <queue>
#include <vector>
using namespace std;

int dist[MAXN];
vector<vi> g;

int farthestFrom(int s) {
```

```
  int farthest = s;
  int maxd = 0;
  memset(dist, -1, sizeof(dist[0]) * n);
  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] > maxd) {
          maxd = dist[v];
          farthest = v;
        }
      }
    }
  }
  return farthest;
}

void findDiameter(int& e1, int& e2) {
  e1 = farthestFrom(0);
  e2 = farthestFrom(e1);
}
```

5.2. Depth-First Search

```
#include <queue>
#include <stack>
#include <vector>
#define MAXN 1000
vector<int> adjList[MAXN];
bool visited[MAXN];


//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 i = 0; i < adjList[u].size(); ++i) {
      int v = adjList[u][i];
      if (visited[v])
        continue;
      visited[u] = true;
      s.push(v);
    }
  }
}

```

```
//recursive
void dfs(int u) {
  visited[u] = true;
  for(int i = 0; i < adjList[i].size(); ++i) {
    int v = adjList[u][i];
    if(!visited[v])
      dfs(v);
  }
}

//------------------------------
// Finding connected components
//------------------------------
int numCC = 0;
memset(visited,false,sizeof visited)
for (int i = 0; i < V; i++)
  if (!visited[i])
    printf("Component %d:", ++numCC), dfs(i), printf("\n");   // 3 lines here!
printf("There are %d connected components\n", numCC);

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

//explicit graph
#define DFS_WHITE (-1)
vector<int> dfs_num(DFS_WHITE,n);
void floodfill(int u, int color) {
  dfs_num[u] = color;                              // not just a generic DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    int v = AdjList[u][j];
    if (dfs_num[v] == DFS_WHITE)
      floodfill(v, color);
  }
}

//implicit graph
int dr[] = {1,1,0,-1,-1,-1, 0, 1}; // trick to explore an implicit 2D grid
int dc[] = {0,1,1, 1, 0,-1,-1,-1}; // S,SE,E,NE,N,NW,W,SW neighbors
int floodfill(int r, int c, char c1, char c2) { // returns the size of CC
  if (r < 0 || r >= R || c < 0 || c >= C) return 0; // outside grid
  if (grid[r][c] != c1) return 0; // does not have color c1
  int ans = 1; // adds 1 to ans because vertex (r, c) has c1 as its color
  grid[r][c] = c2; // now recolors vertex (r, c) to c2 to avoid cycling!
  for (int d = 0; d < 8; d++)
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans; // the code is neat due to dr[] and dc[]
}


//------------------------------
// Topo Sort
//------------------------------

```

```
//option 1: tarjan's algorithm

vector<int> topoSort;
void dfs2(int u) {
  visited[u] = true;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    int v = AdjList[u][j];
    if (!visited[v])
      dfs2(v);
  }
  topoSort.push_back(u); //only change with respect to dfs()
}
//in main
topoSort.clear();
memset(visited, false, sizeof visited);
for (int i = 0; i < V; i++)            // this part is the same as finding CCs
  if (!visited[i])
      dfs2(i);
for (int i = topoSort.size()-1; i >= 0; i--)       // we need to print in reverse order
  printf(" %d", topoSort[i]);


//option 2: Kahn's algorithm

//pseudo-code
// L <- Empty list that will contain the sorted elements
// S <- Set of all nodes with no incoming edges
// while S is non-empty do
//     remove a node n from S
//     add n to tail of L
//     for each node m with an edge e from n to m do
//         remove edge e from the graph
//         if m has no other incoming edges then
//             insert m into S
// if graph has edges then
//     return error (graph has at least one cycle)
// else
//     return L (a topologically sorted order)

//c++ version
//Input : adj_list ->Adjacency list; indegree : indegrees of all nodes .....
void topoSort(vii & adj_list,  vi &indegree) {

  queue<int> tsort_queue;
  vector<int> sorted;

  for(int i = 0; i < (signed)indegree.size(); i++)
    if(indegree[i] == 0)
      tsort_queue.push(i);

  while(!tsort_queue.empty()) {
    int u = tsort_queue.front();
    tsort_queue.pop();
    sorted.push_back(u);
    for(int i = 0; i <  adj_list[u].size(); ++i) {
      int v = adj_list[u][i];
      if(--indegree[v] == 0)
        tsort_queue.push(v);
    }
  }

  printf("Top Sorted Order : ");
  for(int i = 0; i < (signed)sorted.size(); i++)
    printf("%d ",sorted[i]);
  printf("\n");
}

/* ============================== */
/* Articulation Points & Cut Edges */
/* ============================== */
vi depth(N,-1);
vi low(N);
vii graph(N,vi());
int rootChildren = 0;

void dfs(int u, int p, int d) { // (node, parent, depth)
  depth[u] = d;
  low[u] = d;
  for(int v : graph[u]) {
    if (v == p) continue; // direct edge to parent is not back edge
    if (depth[v] == -1) {
      if (p == -1 && ++rootChildren > 1) // root
          printf("root = %d is articulation point\n", root);

      dfs(v, u, d + 1);

      if (low[v] >= depth[u] && p != -1)
        printf("u = %d is articulation point\n",  u);

      if (low[v] > depth[u])
        printf("(u,v) = (%d, %d) is cut edge\n", u, v);

      if (low[v] < low[u]) low[u] = low[v];

    } else if (depth[v] < low[u]) {
      low[u] = depth[v];
    }
  }
}
```

5.3. Dijkstra

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

vector<vector<pii>> g; // graph
```

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

void dijkstra(int source) {
  parent.assign(N, -1);
  mindist.assign(N, INT_MAX);
  mindist[source] = 0;
  priority_queue<pii, vector<pii>, greater<pii>> q;
  q.push(pii(0, source));
  while (!q.empty()) {
    pii p = q.front(); q.pop();
    int u = p.second, dist = p.first;
    if (mindist[u] < dist) continue; // skip outdated improvements
    for (pii& e : g[u]) {
      int v = e.second, w = e.first;
      if (mindist[v] > dist + w) {
        mindist[v] = dist + w;
        parent[v] = u;
        q.push(v);
      }
    }
  }
}
```

5.4. Max Flow : Dinic

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

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

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

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

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

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

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

  void add_edge(int u, int v, ll cap) {
    edge a = {v, (int)g[v].size(), 0, cap};
    edge b = {u, (int)g[u].size(), 0, cap};
    g[u].push_back(a);
    g[v].push_back(b);
  }

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

```
    return ans;
  }
};

// usage
int main() {
  Dinic din(2);
  din.add_edge(0,1,10);
  ll mf = din.max_flow(0,1);
}
```

5.5. Max Flow : EdmondsKarp

```
/* ============== */
/*  Edmonds Karp  */
/* ============== */
// complexity: |V| * |E|^2
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;

#define INF 1000000000
#define MAX_V 40

int res[MAX_V][MAX_V]; //residual capacities
int mf, f, s, t;
vi p;
vector<vi> AdjList;

void augment(int v, int minEdge) {     // traverse BFS spanning tree from s to t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global variable f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); // recursive
                         res[p[v]][v] -= f; res[v][p[v]] += f; }       // update
}

int main() {
  int V, k, vertex, weight;
  scanf("%d %d %d", &V, &s, &t);

  //initialize AdjList and res
  AdjList.assign(V, vi());
  for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
      scanf("%d %d", &vertex, &weight);
      //forward residual capacity
      AdjList[i].push_back(vertex);
      res[i][vertex] = weight;
      //backward residual capacity
      AdjList[vertex].push_back(i);
      res[vertex][i] = 0;
    }
  }

  mf = 0;
  while (1) {

    //run BFS to find aumenting path
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1); //reset parents
    while (!q.empty()) {
      int u = q.front(); q.pop();
      //if (u == t) break; //not necesary, check goto below
      for (int j = 0; j < (int)AdjList[u].size(); j++) {  // we use AdjList here!
        int v = AdjList[u][j];
        if (res[u][v] > 0 && !vis[v]) {
          vis[v] = true, q.push(v), p[v] = u;
          if(v == t) { //target found!!
            goto end_bfs;
          }
        }
      }
    }
    end_bfs:
    augment(t, INF);
    if (f == 0) break;
    mf += f;
  }

  printf("%d\n", mf);                                  // this is the max flow value

  return 0;
}
```

5.6. Minimum Spanning Tree : Kruskal

```
/* ====================================== */
/* KRUSKAL ALGORITHM : Minimum Spanning Tree */
/* ====================================== */

typedef pair<int,int> pii;

// edge list
vector<pair<int,pii>> edge_list; // (weight, (u, v))
// num of nodes
int N;

struct UnionFind {
  vi p, rank;
  int numSets;
  UnionFind(int n) {
    numSets = n;
    rank.assign(n,0);
    p.resize(n);
    rep(i,0,n-1) p[i] = i;
  }
```

```
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {
      numSets--;
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) {
        p[y] = x;
      } else {
        p[x] = y;
        if (rank[x] == rank[y]) rank[y]++;
      }
    }
  }
};

int mst_cost() {
  sort(edge_list.begin(), edge_list.end());
  UnionFind uf(N);
  int cost = 0;
  for (auto& edge : edge_list) {
    int w = edge.first;
    int u = edge.second.first;
    int v = edge.second.second;
    if (!uf.isSameSet(u,v)) {
      cost += w;
      uf.unionSet(u, v);
    }
  }
  return cost;
}
```

5.7. Lowest Common Ancestor

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

// ---------------------------------
// METHOD 1: SPARSE TABLE - EULER TOUR
// ---------------------------------
// construction: O(2|V| log 2|V|) = O(|V| log |V|)
// query: O(1)
// cannot be updated :(


#define MAXN 10000
#define MAXLOG 14

int E[2 * MAXN]; // records sequence of visited nodes
int L[2 * MAXN]; // records level of each visited node
int H[MAXN]; // records index of first ocurrence of node u in E
int idx; // tracks node ocurrences
int rmq[2 * MAXN][MAXLOG + 1];

int N; // number of nodes
vector<vi> g; // tree graph

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

void dfs(int u, int depth) {
  H[u] = idx; // index of first u's ocurrence
  E[idx] = u; // record node ocurrence
  L[idx++] = depth; // record depth
  for (int v : g[u])
    if (H[v] == -1) {
      dfs(v, depth + 1); // backtrack
      E[idx] = u; // new ocurrence of u
      L[idx++] = depth;
    }
}

void lca_init() {
  idx = 0;
  memset(H, -1, sizeof(H[0]) * N);
  dfs(0, 0); // euler tour to initialize H, E, L
  int nn = idx; // <-- make sure you use the correct number
  int m = log2(nn);

  rep(i, 0, nn - 1)
    rmq[i][0] = i; // base case
  rep(j, 1, m) {
    rep(i, 0, nn - (1 << j)) {
      // i ... i + 2 ^ (j-1) - 1
      int i1 = rmq[i][j-1];
      // i + 2 ^ (j-1) ... i + 2 ^ j  - 1
      int i2 = rmq[i + (1 << (j-1))][j-1];
      // choose index with minimum level
      rmq[i][j] = (L[i1] < L[i2] ? i1 : i2);
    }
  }
}

int LCA(int u, int v) {
  // get ocurrence indexes in increasing order
  int l, r;
  if (H[u] < H[v])
    l = H[u], r = H[v];
  else
    l = H[v], r = H[u];
  // get node with minimum level within [l .. r] in O(1)
  int len = r -  l + 1;
```

```
  int m = log2(len);
  int i1 = rmq[l][m];
  int i2 = rmq[r - ((1 << m) - 1)][m];
  return L[i1] < L[i2] ? E[i1] : E[i2];
}

inline 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 L
  return L[H[u]] + L[H[v]] - 2 * L[H[LCA(u,v)]];
}

// ---------------------------------------
// METHOD 2: SPARSE TABLE - JUMP POINTERS
// ---------------------------------------
// construction: O(|V| log |V|)
// query: O(log|V|)
// can be updated: tree can receive new nodes :)

#define MAXN 1000000
#define MAXLOG 20

int P[MAXN][MAXLOG+1]; // level ancestor table
int L[MAXN]; // levels
int N; // num of nodes
vector<vi> g; // tree graph
int root; // root of the tree

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

void init() {
  memset(P, -1, sizeof P);
  memset(L, -1, sizeof L);
  dfs(root, -1, 0);
  rep(j, 1, MAXLOG) {
    rep (i, 0, N-1) {
      // i's 2^j th ancestor is
      // i's 2^(j-1) th ancestor's 2^(j-1) th ancestor
      int p = P[i][j-1];
      if (p != -1) P[i][j] = P[a][j-1];
    }
  }
}

int log2(int x) {
  int i = 0;
  while (x) x>>=1, i++;
  return i-1;
}

int LCA(int u, int v) {
  if (level[u] < level[v]) swap(u, v);
  // raise lowest to same level
  int diff = level[u] - level[v];
  while (diff) {
    int j = log2(diff);
    u = P[u][j];
    diff -= (1 << j);
  }
  if (u == v) return u; // same node, we are done
  // raise u and v to their highest ancestors below
  // the LCA
  invrep (j, MAXLOG, 0)
    // if there are 2^j th ancestors for u and v
    // and they are not the same,
    // then they can be raised and still be below the LCA
    if (P[u][j] != -1 && P[u][j] != P[v][j])
      u = P[u][j], v = P[v][j];
  // the direct parent of u (or v) is lca(u,v)
  return P[u][0];
}

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

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

5.8. Level Ancestor

```
/* ========================== */
/* LA (Level Ancestor Problem) */
/* ========================== */
#include <vector>
using namespace std;
typedef vector<int> vi;

#define rep (i,a,b) for(int i=a; i<=b; ++i)
#define invrep(i,b,a) for(int i=b; i>=a; --i)
```

```

#define MAXN 10000
#define MAXLOG 16

int P[MAXN][MAXLOG + 1]; // level ancestor table
int L[MAXN]; // level array
vector<vi> g; // tree graph
int root; // root of the tree

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

inline int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }

void init_la(int n) {
  memset(P, -1, sizeof P);
  memset(L, -1, sizeof L);
  dfs(root, -1, 0);
  // fill sparse table
  int m = log2(n);
  rep(j, 1, m) {
    rep(i, 0, n - 1) {
      // 2^j th ancestor of i
      // = 2^(j-1) th ancestor of 2^(j-1) th ancestor of i
      int p = P[i][j-1];
      if (p != -1) P[i][j] = P[p][j-1];
    }
  }
}

int lev_anc(int u, int k) {
  if (k == 0) return u; // trivial case
  if (L[u] < k) return -1; // check ancestor exists
  invrep(j, log2(k), 0) {
    if (k >= (1 << j)) {
      u = P[u][j]; // u = 2^j th ancestor of u
      k -= (1 << j); // only k - 2^j steps left
      if (k == 0) break; // target reached
    }
  }
  return u;
}
```

5.9. Centroid Decomposition

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

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

#include <vector>
#include <queue>
#include <cstring>
using namespace std;

#define MAXN 100000
typedef vector<int> vi;

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

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

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

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

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

```
        c = centroid(v);
        cpar[c] = u; // set parent of c to u
        cg[u].push_back(c); // add edge (u -> c)
        removed[c] = true;
        q.push(c);
      }
    }
  }
}
```

6. Geometry

6.1. Convex Hull

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

double isLeft(Point o, Point a, Point b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

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 && isLeft(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 && isLeft(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 && isLeft(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 && isLeft(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
  }

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

6.2. Geometry 2D Utils

```
/* ========================== */
/* Example of Point Definition */
/* ========================== */

struct Point {
  double x, y;
  bool operator==(const Point& p) const { return x==p.x && y == p.y; }
  Point operator+(const Point& p) const { return {x+p.x, y+p.y}; }
  Point operator-(const Point& p) const { return {x-p.x, y-p.y}; }
  Point operator*(double d) const { return {x*d, y*d}; }
  double norm2() const { return x*x + y*y; }
  double norm() const { return sqrt(norm2()); }
  double dot(const Point& p) const { return x*p.x + y*p.y; }
  Point unit() const {
    double d = norm();
    return {x/d,y/d};
  }
};

/* ================ */
/* Angle Comparison */
/* ================ */

// --------------
```

```cpp
// method 1: atan2()
#include <cmath>
const double PI = atan(1) * 4;
double angle(double x, double y) {
  double a = atan2(y, x);
  return (a < 0) ? (a + 2 * PI) : a;
}
int cmpAngles(double x1, double y1, double x2, double y2) {
  double a1 = angle(x1,y1);
  double a2 = angle(x2,y2);
  return (a1 < a2) ? -1 : (a1 == a2) ? 0 : 1;
}

// ----------------------------
// method 2: quadrants + slopes
// this is the prefered method when coordinates
// are given as integers
#include <cmath>
enum Quadrant { UpRight, Up, UpLeft, DownLeft, Down, DownRight };
int getQuadrant(int x, int y) {
  if (x > 0) return (y >= 0) ? UpRight : DownRight;
  if (x < 0) return (y >= 0) ? UpLeft : DownLeft;
  return (y >= 0) ? Up : Down;
}
int cmpAngles(int x1, int y1, int x2, int y2) {
  int q1 = getQuadrant(x1,y1);
  int q2 = getQuadrant(x2,y2);
  if (q1 > q2) return 1;
  if (q1 < q2) return -1;
  int m1 = abs(y1 * x2);
  int m2 = abs(y2 * x1);
  switch (q1) {
    case UpRight:
    case DownLeft:
      return (m1 > m2) ? 1 : (m1 < m2) ? -1 : 0;
    case UpLeft:
    case DownRight:
      return (m1 > m2) ? -1 : (m1 < m2) ? 1 : 0;
    default: return 0;
  }
}

/* ================================== */
/* Straight Line Hashing (integer coords) */
/* ================================== */

struct Point {int x, y; };
struct Line { int a, b, c; };

int gcd(int a, int b) {
  a = abs(a);
  b = abs(b);
  while(b) {
    int c = a;
    a = b;
    b = c % b;
  }
  return a;
}

// Line = {a,b,c} such that a*x + b*y + c = 0
Line getLine(Point p1, Point p2) {
  int a = p1.y - p2.y;
  int b = p2.x - p1.x;
  int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
  int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
  int f = gcd(a, gcd(b, c)) * sgn;
  a /= f;
  b /= f;
  c /= f;
  return {a, b, c};
}

/* ======================= */
/* Point - Segment distance */
/* ======================= */

// get distance between p and truncated projection of p on segment s -> e
double point_segment_dist(const Point& p, const Point& s, const Point& e) {
  if (s==e) return (p-s).norm(); // segment is a single point
  double t = min(1.0, max(0.0, (p-s).dot(e-s) / (e-s).norm2()));
  return (s+(e-s)*t-p).norm();
}

/* ==================== */
/* Point - Line distance */
/* ==================== */

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

6.3. Point Inside Polygon

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

#include <vector>
struct Point { float x, y; };

/* signed area of p0 with respect to (p1 -> p2) */
float isLeft(Point p0, Point p1, Point p2) {
  return (p1.x - p0.x) * (p2.y - p0.y)
    - (p2.x - p0.x) * (p1.y - p0.y);
}
```

```cpp

// ------------------------------------------
// 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 && isLeft(p, prev, curr) > 0)
        ++ wn; // upward & left
    } else {
      if (p.y >= curr.y && isLeft(p, prev, curr) < 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
      float t = (p.y - prev.y) / (curr.y - prev.y);
      if (p.x < prev.x + t * (curr.x - prev.x))
        ++cn;
    }
    prev = curr;
  }
  return (cn & 1); // odd -> in, even -> out
}

// ------------------------------------------------
// Convex Polygon method: check orientation changes
bool inConvexPolygon(Point p, vector<Point>& pts) {
  Point prev_p = pts.back();
  Point curr_p;
  float prev_orient = 0;
  float curr_orient;
  rep (i, 0, (int)pts.size() - 1) {
    curr_p = pts[i];
    curr_orient = isLeft(p, prev, curr);
    if ((prev_orient < 0 && curr_orient > 0)
      || (prev_orient > 0 && curr_orient < 0))
      return false;
    prev_p = curr_p;
    prev_orient = curr_orient;
  }
  return true;
}
```

6.4. Polygon Area

```cpp
/* ===================================== */
/* Area of 2D non self intersecting Polygon */
/* ===================================== */
//based on Green Theorem

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

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

7. Mathematics

7.1. Modular Arithmetics

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

/* ============================ */
/* GCD (greatest common divisor) */
/* ============================ */
// euclid algorithm
int gcd (int a, int b) {
  while (b) {
    int aux = a;
    a = b;
    b = aux % b;
  }
```

```
  return a;
}

/* ============ */
/* GCD extended */
/* ============ */
// extended euclid algorithm
// a * x + b * y = d = gcd(a, b)
// x = x0 + n * (b/d)
// y = y0 - n * (a/d)
void gcdext(int a, int b, int& d, int& x, int& y) {
  if (b == 0) { x = 1; y = 0; d = a; return; }
  gcdext(b, a % b, d, x, y);
  int x1 = y;
  int y1 = x - y * (a / b);
  x = x1;
  y = y1;
}

/* ================== */
/* Integer Root Square */
/* ================== */

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

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

7.2. Modular Fibonacci

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

typedef unsigned long long ull;
const ull MOD = 1000000000;

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

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

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

7.3. Prime Numbers

```
//=========================================
// Sieve of Eratosthenes (all primes up to N)
//=========================================
void collect_primes_up_to(int n, vector<int>& primes) {
  vector<bool> isPrime(n + 1, true);
  int limit = (int) floor(sqrt(n));
  for (int i = 2; i <= limit; ++i)
    if (isPrime[i])
      for (int j = i * i; j <= n; j += i)
        isPrime[j] = false;
  for (int i = 2; i <= n; ++i)
    if (isPrime[i])
      primes.push_back(i);
```

```
}


//====================================
// Prime Factorization of Factorials
//====================================
// source: http://mathforum.org/library/drmath/view/67291.html
int N = 9999;
int pcount[N];
vector<int> primes;
collect_primes_up_to(N,primes);
int number = 12312; //the number we want the prime factorization of
for (int i = 0; i < (int)primes.size() && primes[i] <= N; ++i) {
  int p = primes[i];
  pcount[p] = 0;
  int n = number;
  while ((n /= p) > 0)
    pcount[p] += n;
}
```

8. Strings

8.1. Suffix Array

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

typedef pair<int, int> ii;

#define MAX_N 100010                          // second approach: O(n log n)
char T[MAX_N];                      // the input string, up to 100K characters
int n;                                        // the length of input string
int RA[MAX_N], tempRA[MAX_N];        // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];    // suffix array and temporary suffix array
int c[MAX_N];                                 // for counting/radix sort

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }      // compare

void constructSA_slow() {               // cannot go beyond 1000 characters
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
  sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) = O(n^2 log n)
}

void countingSort(int k) {                                      // O(n)
  int i, sum, maxi = max(300, n);   // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);                         // clear frequency table
  for (i = 0; i < n; i++)       // count the frequency of each integer rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t;
  }
  for (i = 0; i < n; i++)          // shuffle the suffix array if necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++)                     // update the suffix array SA
    SA[i] = tempSA[i];
}

void constructSA() {         // this version can go up to 100000 characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                // initial rankings
  for (i = 0; i < n; i++) SA[i] = i;     // initial SA: {0, 1, 2, ..., n-1}
  for (k = 1; k < n; k <<= 1) {       // repeat sorting process log n times
    countingSort(k);  // actually radix sort: sort based on the second item
    countingSort(0);          // then (stable) sort based on the first item
    tempRA[SA[0]] = r = 0;             // re-ranking; start from rank r = 0
    for (i = 1; i < n; i++)                     // compare adjacent suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)                      // update the rank array RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;               // nice optimization trick
} }
```

9. Parsers

9.1. Arithmetic Parser

```
/**
 * Implementation of LL(1), recursive-descent Parser
 * for Arithmetic Expressions
 */
#include <cstdio>
#include <iostream>
#include <string>
#include <stack>
#include <vector>
#include <cstdlib>
#include <stdexcept>
#include <cmath>
using namespace std;

#define rep(i,a,b) for(int i=a; i<=b; ++i)

char errorBuffer[200];

enum Terminal { NUMBER, MINUS, PLUS, DIV, MULT, SQRT, OPEN_PAREN, CLOSE_PAREN, END };

const char * terminal2String(Terminal t) {
  switch (t) {
  case NUMBER: return "NUMBER";
  case MINUS: return "MINUS";
  case PLUS: return "PLUS";
  case DIV: return "DIV";
  case MULT: return "MULT";
```

```
  case SQRT: return "SQRT";
  case OPEN_PAREN: return "OPEN_PAREN";
  case CLOSE_PAREN: return "CLOSE_PAREN";
  default: return "END";
  }
}

struct Token {
  Terminal terminal;
  Token(Terminal t) :
      terminal(t) {
  }
};

struct NumberToken: Token {
  double value;
  NumberToken(double value) :
      Token(NUMBER), value(value) {
  }
};

typedef vector<Token*> vtp;


void skipWhitespace(const char* buffer, int& offset) {
  while (true) {
    char c = buffer[offset];
    if (c == ' ' || c == '\t') offset++;
    else break;
  }
}

bool isDigit(char c) {
  return '0' <= c && c <= '9';
}

Token* getNextToken(const char* buffer, int& offset) {
  skipWhitespace(buffer, offset);
  char c = buffer[offset];
  switch (c) {
  case '(':
    offset++;
    return new Token(OPEN_PAREN);
  case ')':
    offset++;
    return new Token(CLOSE_PAREN);
  case '*':
    offset++;
    return new Token(MULT);
  case '/':
    offset++;
    return new Token(DIV);
  case '+':
    offset++;
    return new Token(PLUS);
  case '-':
    offset++;
    return new Token(MINUS);
  case '\0':
    return new Token(END);
  case 's':{
    rep(i,0,3) if (buffer[offset + i] != "sqrt"[i]) {
      sprintf(errorBuffer, "unexpected char '%c' at position %d\n", buffer[offset + i],
          offset + i);
      throw std::runtime_error(errorBuffer);
    }
    offset += 4;
    return new Token(SQRT);
  }
  default:
    if (isDigit(c)) {
      char* endp;
      double num = strtod(buffer + offset, &endp);
      offset = endp - buffer;
      return new NumberToken(num);
    }
    sprintf(errorBuffer, "unexpected char '%c' at position %d\n", c, offset);
    throw std::runtime_error(errorBuffer);
  }
}

struct Node {
  virtual ~Node() {};
  virtual double eval() = 0;
};
struct DoubleOpNode: Node {
  Node* left;
  Node* right;
  DoubleOpNode(Node* left, Node* right): left(left), right(right) {}
  ~DoubleOpNode() { delete left; delete right; }
};
struct SingleOpNode: Node {
  Node* child;
  SingleOpNode(Node* child): child(child) {}
  ~SingleOpNode() { delete child; }
};
struct AddNode : DoubleOpNode {
  AddNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
  double eval() { return left->eval() + right->eval(); }
};
struct SubNode : DoubleOpNode {
  SubNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
  double eval() { return left->eval() - right->eval(); }
};
struct MultNode : DoubleOpNode {
  MultNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
  double eval() { return left->eval() * right->eval(); }
};
```

```
struct DivNode : DoubleOpNode {
  DivNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
  double eval() { return left->eval() / right->eval(); }
};
struct NegNode : SingleOpNode {
  NegNode(Node* child) : SingleOpNode(child) {}
  double eval() { return -child->eval(); }
};
struct SqrtNode : SingleOpNode {
  SqrtNode(Node* child) : SingleOpNode(child) {}
  double eval() { return sqrt(child->eval()); }
};
struct IntegerNode : Node {
  double value;
  IntegerNode(double value) : value(value) {}
  double eval() { return value; }
};

/**
 * Context Free Grammar:
 *  Root    ->  AddSum1 END
 *  AddSum1   ->  MultDiv1 AddSum2
 *  AddSum2   ->  + MultDiv1 AddSum2 | - MultDiv1 AddSum2 | epsilon
 *  MultDiv1  ->  Term MultDiv2
 *  MultDiv2  ->  * Term MultDiv2 | / Term MultiDiv2 | epsilon
 *  Term    ->  - Term | (AddSum1) | SQRT(AddSum1) | NUMBER
 */

vector<Token*> tokens;
int offset;
stack<Node*> nodes;

void throwUnexpectedTerminalException(Terminal terminal, int offset);
void matchAndConsume(Terminal terminal);
void parseTerm();
void parseMultDiv1();
void parseMultDiv2();
void parseAddSub1();
void parseAddSub2();
void parseRoot();

template<typename T>
void swap2for1() {
  Node* r = nodes.top(); nodes.pop();
  Node* l = nodes.top(); nodes.pop();
  nodes.push(new T(l,r));
}

template<typename T>
void swap1for1() {
  Node* n = nodes.top(); nodes.pop();
  nodes.push(new T(n));
}

void throwUnexpectedTerminalException(Terminal terminal, int offset) {
  sprintf(errorBuffer, "unexpected terminal %s at position %d\n", terminal2String(
      terminal), offset);
  throw std::runtime_error(errorBuffer);
}
void matchAndConsume(Terminal terminal) {
  if (tokens[offset]->terminal != terminal) {
    sprintf(errorBuffer, "expected terminal %s but found %s\n",
      terminal2String(terminal),
      terminal2String(tokens[offset]->terminal));
    throw std::runtime_error(errorBuffer);
  }
  offset++;
}
void parseTerm() {
  Token* t = tokens[offset];
  switch (t->terminal) {
    case MINUS: {
      offset++;
      parseTerm();
      // generate node
      swap1for1<NegNode>();
      break;
    }
    case OPEN_PAREN: {
      offset++;
      parseAddSub1();
      matchAndConsume(CLOSE_PAREN);
      break;
    }
    case SQRT: {
      offset++;
      matchAndConsume(OPEN_PAREN);
      parseAddSub1();
      matchAndConsume(CLOSE_PAREN);
      swap1for1<SqrtNode>();
      break;
    }
    case NUMBER: {
      offset++;
      // generate node
      double value = static_cast<NumberToken*>(t)->value;
      nodes.push(new IntegerNode(value));
      break;
    }
    default:
      throwUnexpectedTerminalException(t->terminal, offset);
      break;
  }
}
void parseMultDiv1() {
  parseTerm();
  parseMultDiv2();
}
```

```
void parseMultDiv2() {
  Token* t = tokens[offset];
  switch (t->terminal) {
    case MULT: {
      offset++;
      parseTerm();
      // generate node
      swap2for1<MultNode>();
      // resume parsing
      parseMultDiv2();
      break;
    }
    case DIV: {
      offset++;
      parseTerm();
      // generate node
      swap2for1<DivNode>();
      // resume parsing
      parseMultDiv2();
      break;
    }
    // follow set
    case PLUS: case MINUS: case END: case CLOSE_PAREN:
      break;
    default:
      throwUnexpectedTerminalException(t->terminal, offset);
      break;
  }
}
void parseAddSub2() {
  Token* t = tokens[offset];
  switch (t->terminal) {
    case PLUS: {
      offset++;
      parseMultDiv1();
      // generate node
      swap2for1<AddNode>();
      // resume parsing
      parseAddSub2();
      break;
    }
    case MINUS: {
      offset++;
      parseMultDiv1();
      // generate node
      swap2for1<SubNode>();
      // resume parsing
      parseAddSub2();
      break;
    }
    // follow set
    case END: case CLOSE_PAREN:
      break;
    default:
      throwUnexpectedTerminalException(t->terminal, offset);
      break;
  }
}
void parseAddSub1() {
  parseMultDiv1();
  parseAddSub2();
}
void parseRoot() {
  parseAddSub1();
  matchAndConsume(END);
}


int main() {
  string line;
  while (true) {
    /*  read input */
    getline(cin, line);
    if (line == "exit") break;

    /* get tokens */
    int index = 0;
    while(true) {
      Token* t = getNextToken(line.c_str(), index);
      tokens.push_back(t);
      if (t->terminal == END) break;
    }

    /* parse tokens to generate AST */
    parseRoot();
    Node* root = nodes.top();

    /* print result */
    printf("==> %lf\n", root->eval());

    /* clean memory */
    delete root;
    for (int i = 0, l = tokens.size(); i < l; ++i) delete tokens[i];
    tokens.clear();
    nodes.pop();
    offset = 0;
  }
  return 0;
}
```