Contents

1 Info About Memory and Time Limits

$O(f(n))$	Limite
$O(n!)$	$10, \ldots, 11$
$O(2^n n^2)$	$15, \ldots, 18$
$O(2^n n)$	$18, \ldots, 21$
$O(n^4)$	100
$O(n^3)$	500[1]
$O(n^2 \log^2 n)$	1000
$O(n^2 \log n)$	2000
$O(n^2)$	1e4[2]
$O(n \log^2 n)$	3e5
$O(n \log n)$	1e6
$O(n)$	1e8[3]

2 C++ Cheat Sheet

2.1 Headers

```
#pragma GCC optimize("Ofast")
#include <bits/stdc++.h> //Import all

using namespace std; //Less verbose code

typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> ii;
typedef tuple<int, int, int> iii;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef vector<ii> vii;

typedef vector<vi> graph;
typedef vector<vii> wgraph;

#ifndef declaraciones_h
#define declaraciones_h

// Reps are inclusive exclusive (i.e. range is [a,b))
#define rep(i, n) for (int i = 0; i < (int)n; i++)
#define repx(i, a, b) for (int i = a; i < (int)b; i++)
#define invrep(i, a, b) for (int i = b; i-- > (int)a;)

#define pb push_back
#define eb emplace_back
#define ppb pop_back

// Base two log for ints and for ll
#define lg(x) (31 - __builtin_clz(x))
#define lgg(x) (63 - __buitlin_clzll(x))
#define gcd __gcd

#define umap unordered_map
#define uset unordered_set

//Debugs single variables (e.g. int, string)
#define debugx(x) cerr << #x << ": " << x << endl
//Debugs Iterables (e.g. vi, uset<int>)
#define debugv(v)         \
    cerr << #v << ":";    \
    for (auto e : v)      \
    {                     \
        cerr << " " << e; \
    }                     \
    cerr << endl
//Debugs Iterables of Iterables (e.g. graph, umap<int,umap<int, int>)
#define debugm(m)             \
    cerr << #m << endl;       \
    for (auto v : m)          \
    {                         \
        for (auto e : v)      \
            cerr << " " << e; \
        cerr << endl;         \
    }
#define print(x) copy(x.begin(), x.end(), ostream_iterator<int>(cout, "")), cout << endl

//Outputs generic pairs through streams (including cerr and cout)
template <typename T1, typename T2>
ostream &operator<<(ostream &os, const pair<T1, T2> &p)
{
    os << '(' << p.first << ',' << p.second << ')';
    return os;
}

#endif
```

2.2 Cheat Sheet

```
/* ================== */
/* Reading from stdin */
/* ================== */
// With scanf
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
```

[1] Este caso esta justo en el limite de tiempo, además en 256 MB cabe a los una matriz de 400^3 ints

[2] En general solo funciona hasta 6e3

[3] En general solo funciona hasta 4e7

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

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

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

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

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

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

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

// Optimizations with cin/cout
ios::sync_with_stdio(0);
cin.tie(0);
cout.tie(0);

// Fix precision on cout
cout.setf(ios::fixed);
cout.precision(4); // e.g. 1.000

/* ===================== */
/* USING PAIRS AND TUPLES */
/* ===================== */
// ii = pair<int,int>
ii p(5, 5);
```

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

/* ================================ */
/* CONVERTING FROM STRING TO NUMBERS */
/* ================================ */
//------------------------
// string to int
// option #1:
int atoi(const char *str);
// option #2:
sscanf(string, "%d", &i);
//--------------------------
// string to long int:
// option #1:
long int strtol(const char *str, char **endptr, int base);
// it only works skipping whitespaces, so make sure your numbers
// are surrounded by whitespaces only
// Example:
char szNumbers[] = "2001 60c0c0 -1101110100110100100000 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);

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

```cpp
/*  OPERATOR OVERLOADING */
/* ==================== */

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

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

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

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

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

/* ===================== */
/* SET UTILITY FUNCTIONS */
/* ===================== */
set<int> myset;
myset.begin();  // iterator to first elemnt
myset.end();    // iterator to after last element
myset.rbegin(); // iterator to last element
myset.rend();   // iterator to before first element
for (auto it = myset.begin(); it != myset.end(); ++it) {
  do_something(*it);
} // left -> right
for (auto it = myset.rbegin(); it != myset.rend(); ++it) {
  do_something(*it);
} // right -> left
for (auto &i : myset) {
```

```cpp
  do_something(i);
} // left->right shortcut
auto ret = myset.insert(
    5); // ret.first = iterator, ret.second = boolean (inserted / not
         inserted)
int count = mysert.erase(5); // count = how many items were erased
if (!myset.empty()) {
}
// custom comparator 1: functor
struct cmp {
  bool operator()(int i, int j) { return i > j; }
};
set<int, cmp> myset;
// custom comparator 2: function
bool cmp(int i, int j) { return i > j; }
set<int, bool (*)(int, int)> myset(cmp);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 General Algorithms

3.1 Search

3.1.1 Binary Search

```
// On iterables v use lower_bound(v.begin(),v.begin()+delta,key) and upper_bound(v.begin(), v.begin()+delta,key)

int val;
vi vals;
bool discreteP(int x) { return x > val; }

int bin(int start, int end)
{
    int left = start, right = end, mid;
    while (left < right)
    {
        mid = (left + right) / 2;
        if (discreteP(vals[mid]))
            right = mid;
        else
            left = mid + 1;
    }
    return left;
}

double approx;
bool continuousP(double x) { return x > approx; }

double bin(double start, double end)
{
    double left = start, right = end;
    int reps = 80; //Safe numbers check if viable for problem
    double mid;
    rep(_, reps)
    {
        mid = (left + right) / 2;
        if (continuousP(mid))
            right = mid;
        else
            left = mid;
```

```
    }
    return mid;
}
```

3.1.2 Ternary Search

```

double f(double x)
{
    return -x * x;
}

bool compare(double x, double y) { return f(x) < f(y); }

double maxTer(double start, double end)//Searches maximum of f in range
     [start, end]
{
    double left = start, right = end;
    double mid1, mid2;
    int reps = 80;
    rep(_, reps)
    {
        mid1 = left + (right - left) / 3, mid2 = right - (right - left)
             / 3;
        if (compare(mid1, mid2))
            left = mid1;
        else
            right = mid2;
    }
    return (mid1 + mid2) / 2; // * Can return -0!
    // Tends to the right
}

double minTer(double start, double end)//Searches minimum of f in range
     [start,end]
{
    double left = start, right = end;
    double mid1, mid2;
    int reps = 80;
    rep(_, reps)
    {
        mid1 = left + (right - left) / 3, mid2 = right - (right - left)
             / 3;
        if (not compare(mid1, mid2))
            left = mid1;
        else
            right = mid2;
    }
    return (mid1 + mid2) / 2;
    // Tends to the left
}
```

3.2 Brute Force

4 Data Structures

4.1 Segment Tree

4.1.1 Lazy

```
struct RSQ // Range sum query
{
  static ll const neutro = 0;
  static ll op(ll x, ll y) { return x + y; }
  static ll lazy_op(int i, int j, ll x) { return (j - i + 1) * x; }
};

struct RMinQ // Range minimun query
{
  static ll const neutro = 1e18;
  static ll op(ll x, ll y) { return min(x, y); }
  static ll lazy_op(int i, int j, ll x) { return x; }
};

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::op(st[l], st[r]);
  }

  void propagate(int u, int i, int j, ll x) {
    // nota, las operaciones pueden ser un and, or, ..., etc.
    st[u] += t::lazy_op(i, j, x); // incrementar el valor (+)
    // st[u] = t::lazy_op(i, j, x); // setear el valor
    if (i != j) {
      // incrementar el valor
      lazy[u * 2 + 1] += x;
      lazy[u * 2 + 2] += x;
      // setear el valor
      // lazy[u * 2 + 1] = x;
      // 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)
```

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

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

  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); }
};
```

4.1.2 Iterative

```cpp
// It requires a struct for a node (e.g. prodsgn)
// A node must have three constructors
//      Arity 0: Constructs the identity of the operation (e.g. 1 for prodsgn)
//      Arity 1: Constructs a leaf node from the input
//      Arity 2: Constructs a node from its children
//
// Building the Segment Tree:
//      Create a vector of nodes (use constructor of arity 1).
//      ST<miStructNode> mySegmentTree(vectorOfNodes);
// Update:
//      mySegmentTree.set_points(index, myStructNode(input));
// Query:
//      mySegmentTree.query(l, r); (It searches on the range [l,r), and returns
//      a node.)

// Logic And Query
struct ANDQ {
  ll value;
  ANDQ() { value = -1ll; }
  ANDQ(ll x) { value = x; }
  ANDQ(const ANDQ &a, const ANDQ &b) { value = a.value & b.value; }
};

// 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); }
};

template <class node> class ST {
  vector<node> t;
  int n;

public:
```

```
  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);
  }
};
```

4.2 Fenwick Tree/BIT

4.2.1 1D

```

struct FenwickTree {
  vector<int> FT;
  FenwickTree(int N) { FT.resize(N + 1, 0); }

  int query(int i) {
    int ans = 0;
    for (; i; i -= i & (-i))
      ans += FT[i];
    return ans;
  }

  int query(int i, int j) { return query(j) - query(i - 1); }

  void update(int i, int v) {
    int s = query(i, i); // Sets range to v?
    for (; i < FT.size(); i += i & (-i))
      FT[i] += v - s;
  }

  // Queries puntuales, Updates por rango
  void update(int i, int j, int v) {
    update(i, v);
    update(j + 1, -v);
  }
};
```

4.2.2 2D

4.3 Wavelet Tree

```

class WaveTree {
  typedef vector<int>::iterator iter;
  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);
  }

public:
  // 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);
  }
};
```

5 Dynamic Programming

5.1 Knapsack

```
vector<vector<ll>> DP;
vector<ll> Weights;
vector<ll> Values;

ll Knapsack(int w, int i) {
  if (w == 0 or i == -1)
    return 0;
  if (DP[w][i] != -1)
    return DP[w][i];
  if (Weights[i] > w)
    return DP[w][i] = Knapsack(w, i - 1);
  return DP[w][i] = max(Values[i] + Knapsack(w - Weights[i], i - 1),
                        Knapsack(w, i - 1));
}
```

5.2 Matrix Chain Multiplication

```
vector<vector<ii>> DP; // Pair value, op result
int n;                 // Size of DP (i.e. i,j<n)
ii op(ii a, ii b) {
  return {
      a.first + b.first + a.second * b.second,
      (a.second + b.second) %
          100}; // Second part MUST be associative, first part is cost
                function
}

ii MCM(int i, int j) {
  if (DP[i][j].first != -1)
    return DP[i][j];
  int ans = 1e9; // INF
  int res;
  repx(k, i + 1, j) {
    ii temp = op(MCM(i, k), MCM(k, j));
    ans = min(ans, temp.first);
    res = temp.second;
  }
  return DP[i][j] = {ans, res};
}

void fill() {
  DP.assign(n, vector<ii>(n, {-1, 0}));
  rep(i, n - 1) {
    DP[i][i + 1].first = 1;
  } // Pair op identity, cost (cost must be from input)
}
```

5.3 Longest Increasing Subsequence

```
vi L;
vi vals;
int maxl = 1;

// Bottom up approach O(nlogn)
int lis(int n) {
  L.assign(n, -1);
  L[0] = vals[0];
  repx(i, 1, n) {
    auto it = lower_bound(L.begin(), L.begin() + maxl, vals[i]);
    if (it == L.begin() + maxl) {
      L[maxl] = vals[i];
      maxl++;
    } else
      *it = vals[i];
  }
  return maxl;
}
```

6 Graphs

6.1 Graph Traversal

6.1.1 Breadth First Search

```
void bfs(graph &g, int start) {
  int n = g.size();
  vi visited(n, 1);
  queue<int> q;

  q.emplace(start);
  visited[start] = 0;
  while (not q.empty()) {
    int u = q.front();
    q.pop();

    for (int v : g[u]) {
      if (visited[v]) {
        q.emplace(v);
        visited[v] = 0;
      }
    }
  }
}
```

6.1.2 Recursive Depth First Search

```
// Recursive (create visited filled with 1s)
void dfs_r(graph &g, vi &visited, int u) {
  cout << u << '\n';
  visited[u] = 0;

  for (int v : g[u])
    if (visited[v])
      dfs_r(g, visited, v);
}
```

6.1.3 Iterative Depth First Search

```
// Iterative
void dfs_i(graph &g, int start) {
  int n = g.size();
  vi visited(n, 1);
  stack<int> s;

  s.emplace(start);
  visited[start] = 0;
  while (not s.empty()) {
    int u = s.top();
    s.pop();

    for (int v : g[u]) {
      if (visited[v]) {
```

```
        s.emplace(v);
        visited[v] = 0;
      }
    }
  }
}
```

6.2 Shortest Path Algorithms

6.2.1 Dijsktra

All edges have non-negative values

```
// g has vectors of pairs of the form (w, index)
int dijsktra(wgraph g, int start, int end) {
  int n = g.size();
  vi cost(n, 1e9); //~INT_MAX/2
  priority_queue<ii, greater<ii>> q;

  q.emplace(0, start);
  cost[start] = 0;
  while (not q.empty()) {
    int u = q.top().second, w = q.top().first;
    q.pop();

    // we skip all nodes in the q that we have discovered before at a
        lower cost
    if (cost[u] < w)
      continue;

    for (auto v : g[u]) {
      if (cost[v.second] > v.first + w) {
        cost[v.second] = v.first + w;
        q.emplace(cost[v.second], v.second);
      }
    }
  }
  return cost[end];
}
```

6.2.2 Bellman Ford

Edges can be negative, and it detects negative cycles

```
bool bellman_ford(wgraph &g, int start) {
  int n = g.size();
  vector<int> dist(n, 1e9); //~INT_MAX/2
  dist[start] = 0;
  rep(i, n - 1) rep(u, n) for (ii p : g[u]) {
    int v = p.first, w = p.second;
    dist[v] = min(dist[v], dist[u] + w);
  }

  bool hayCicloNegativo = false;
  rep(u, n) for (ii p : g[u]) {
    int v = p.first, w = p.second;
    if (dist[v] > dist[u] + w)
      hayCicloNegativo = true;
  }

  return hayCicloNegativo;
}
```

6.2.3 Floyd Warshall

Shortest path from every node to every other node

```
/*
Floyd Warshall implemenation, note that g is using an adjacency matrix
     and not
an adjacency list
*/
static const int INF = 1e9;
graph floydWarshall(const graph g) {
  int n = g.size();
  graph dist(n, vi(n, -1));

  rep(i, n) rep(j, n) dist[i][j] = g[i][j];

  rep(k, n) rep(i, n) rep(j, n) if (dist[i][k] + dist[k][j] < dist[i][j]
       &&
                                    dist[i][k] != INF && dist[k][j] !=
                                         INF)
      dist[i][j] = dist[i][k] + dist[k][j];

  return dist;
}
```

6.3 Minimum Spanning Tree (MST)

6.3.1 Kruskal

```
struct edge {
  int u, v;
  ll w;
  edge(int u, int v, ll w) : u(u), v(v), w(w) {}

  bool operator<(const edge &o) const { return w < o.w; }
};

class Kruskal {
private:
  ll sum;
  vi p, rank;

public:
  // Amount of Nodes n, and unordered vector of Edges E
  Kruskal(int n, vector<edge> E) {
```

```cpp
    sum = 0;
    p.resize(n);
    rank.assign(n, 0);
    rep(i, n) p[i] = i;
    sort(E.begin(), E.end());
    for (auto &e : E)
      UnionSet(e.u, e.v, e.w);
  }
  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, ll w) {
    if (not 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]++;

      sum += w;
    }
  }
  ll mst_val() { return sum; }
};
```

6.4 Lowest Common Ancestor (LCA)

Supports multiple trees

```cpp
class LcaForest {
  int n;
  vi parent;
  vi level;
  vi root;
  graph P;

public:
  LcaForest(int n) {
    this->n = n;
    parent.assign(n, -1);
    level.assign(n, -1);
    P.assign(n, vi(lg(n) + 1, -1));
    root.assign(n, -1);
  }
  void addLeaf(int index, int par) {
    parent[index] = par;
    level[index] = level[par] + 1;
    P[index][0] = par;
    root[index] = root[par];
    for (int j = 1; (1 << j) < n; ++j) {
      if (P[index][j - 1] != -1)
        P[index][j] = P[P[index][j - 1]][j - 1];
    }
  }
  void addRoot(int index) {
    parent[index] = index;
    level[index] = 0;
    root[index] = index;
  }
  int lca(int u, int v) {
    if (root[u] != root[v] || root[u] == -1)
      return -1;
    if (level[u] < level[v])
      swap(u, v);
    int dist = level[u] - level[v];
    while (dist != 0) {
      int raise = lg(dist);
      u = P[u][raise];
      dist -= (1 << raise);
    }
    if (u == v)
      return u;
    for (int j = lg(n); j >= 0; --j) {
      if (P[u][j] != -1 && P[u][j] != P[v][j]) {
        u = P[u][j];
        v = P[v][j];
      }
    }
    return parent[u];
  }
};
```

6.5 Max Flow

```cpp
class 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;
```

```cpp
        }
      }
    }
    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;
  }

public:
  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].pb(a);
    g[v].pb(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;
  }
};
```

6.6 Others

6.6.1 Diameter of a tree

```cpp
graph Tree;
vi dist;

// Finds a diameter node
int bfs1() {
  int n = Tree.size();
  queue<int> q;

  q.emplace(0);
  dist[0] = 0;
  int u;
  while (not q.empty()) {
    u = q.front();
    q.pop();

    for (int v : Tree[u]) {
      if (dist[v] == -1) {
        q.emplace(v);
        dist[v] = dist[u] + 1;
      }
    }
  }
  return u;
}

// Fills the distances from one diameter node and finds another diameter node
int bfs2() {
  int n = Tree.size();
  vi visited(n, 1);
  queue<int> q;
  int start = bfs1();
  q.emplace(start);
  visited[start] = 0;
  int u;
  while (not q.empty()) {
    u = q.front();
    q.pop();

    for (int v : Tree[u]) {
      if (visited[v]) {
        q.emplace(v);
        visited[v] = 0;
        dist[v] = max(dist[v], dist[u] + 1);
      }
    }
  }
  return u;
}

// Finds the diameter
int bfs3() {
  int n = Tree.size();
  vi visited(n, 1);
  queue<int> q;
```

```cpp
  int start = bfs2();
  q.emplace(start);
  visited[start] = 0;
  int u;
  while (not q.empty()) {
    u = q.front();
    q.pop();

    for (int v : Tree[u]) {
      if (visited[v]) {
        q.emplace(v);
        visited[v] = 0;
        dist[v] = max(dist[v], dist[u] + 1);
      }
    }
  }
  return dist[u];
}
```

7 Mathematics

7.1 Useful Data

n	Primes less than n	Maximal Prime Gap	$\max_{0<i<n}(d(i))$
1e2	25	8	12
1e3	168	20	32
1e4	1229	36	64
1e5	9592	72	128
1e6	78.498	114	240
1e7	664.579	154	448
1e8	5.761.455	220	768
1e9	50.487.534	282	1344

7.2 Modular Arithmetic

7.2.1 Chinese Remainder Theorem

```cpp

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

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

```cpp
// 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];
  repx(i, 1, n) {
    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, n) assert (r1 % m[i] == r[i]);
  return {r1, m1};
}
```

7.2.2 Binomial Coefficients mod m

```cpp
#include "../CRT/CRT.cpp"
#include "../modularArithmetic/modularArithmetic.cpp"
#include "../primalityChecks/millerRabin/millerRabin.cpp"
#include "../primalityChecks/sieveEratosthenes/sieve.cpp"

// Modular computation of nCr using lucas theorem, granville theorem and CRT

ll num;            // Set num to the corresponding mod for the nCr calculations
umap<ll, int> MOD; // MOD[P]=V_p(mod)
umap<ll, vector<ll>> FMOD; // n! mod p if MOD[p]=1 else the product of all i mod
                           // P^MOD[P], where 1<=i<=n and (i,p)=1
umap<ll, vector<ll>> invFMOD; // the inverse of FMOD[n] in the corresponding MOD

void preCompute() {
  // Factor mod->MOD
  vi primes = sieve(num);
  ll m = num;
  for (auto p : primes) {
    if (p * p > m)
      break;
    while (m % p == 0) {
      MOD[p]++;
      if ((m /= p) == 1)
        goto next;
    }
  }
  if (m > 1)
    MOD[m] = 1;
next:
  // Compute FMOD and invFMOD
  for (auto p : MOD) {
    int m = pow(p.first, p.second); // p^V_p(n)
    FMOD[p.first].assign(m, 1);
    invFMOD[p.first].assign(m, 1);
    repx(i, 2, FMOD[p.first].size()) {
      if (i % p.first == 0 and p.second > 1)
        FMOD[p.first][i] = FMOD[p.first][i - 1];
      else
        FMOD[p.first][i] = mul(FMOD[p.first][i - 1], i, FMOD[p.first].size());

      // Compute using Euler's theorem i.e. a^phi(m)=1 mod m with (a.m)=1
      invFMOD[p.first][i] = modularInverse(FMOD[p.first][i], m);
    }
  }
}

// Compute nCr using Granville's theorem (prime powers)
// Auxiliary functions

// V_p(n!) using Legendre's theorem
int V(ll n, int p) {
  int e = 0;
  while ((n /= p) > 0)
    e += n;
  return e;
}

//
ll f(ll n, ll p) {
  ll m = pow(p, MOD[p]);
  int e = n / m;
  return mul(fastPow(FMOD[p][m - 1], e, m), FMOD[p][n % m], m);
}
ll F(ll n, ll p) {
  ll m = pow(p, MOD[p]);
  ll ans = 1;
```

```
  do {
    ans = mul(ans, f(n, p), m);
  } while ((n /= p) > 0);
  return ans;
}
// Granville theorem
ll granville(ll n, ll r, int p) {
  int e = V(n, p) - V(n - r, p) - V(r, p);
  ll m = pow(p, MOD[p]);
  if (e >= MOD[p])
    return 0;
  ll ans = fastPow(p, e, m);
  ans = mul(ans, F(n, p), m);
  ans = mul(ans, modularInverse(F(r, p), m), m);
  ans = mul(ans, modularInverse(F(n - r, p), m), m);
  return ans;
}

// Compute nCr using Lucas theorem (primes)
ll lucas(ll n, ll r, int p) {
  // Trivial cases
  if (r > n or r < 0)
    return 0;
  if (r == 0 or n == r)
    return 1;
  if (r == 1 or r == n - 1)
    return n % p;
  // Base case
  if (n < p and r < p) {
    ll ans = mul(invFMOD[p][r], invFMOD[p][n - r], p); // 1/(r!(n-r)!)
        mod p
    ans = mul(ans, FMOD[p][n], p);                     // n!/(r!(n-r!))
        mod p
    return ans;
  }
  ll ans = lucas(n / p, r / p, p);           // Recursion
  ans = mul(ans, lucas(n % p, r % p, p), p); // False recursion
  return ans;
}

// Given the prime decomposition of mod;
ll nCr(ll n, ll r) {
  // Trivial cases
  if (n < r or r < 0)
    return 0;
  if (r == 0 or r == n)
    return 1;
  if (r == 1 or r == n - 1)
    return (n % num);
  // Non-trivial cases
  ll ans = 0;
  ll mod = 1;
  for (auto p : MOD) {
    ll temp = pow(p.first, p.second);
    if (p.second > 1) {
      ans = CRT(ans, mod, granville(n, r, p.first), temp).first;
    } else {
      ans = CRT(ans, mod, lucas(n, r, p.first), temp).first;
    }
    mod *= temp;
  }
  return ans;
}
```

7.3 Primality Checks

7.3.1 Miller Rabin

```

ll mulmod(ull a, ull b, ull c) {
  ull x = 0, y = a % c;
  while (b) {
    if (b & 1)
      x = (x + y) % c;
    y = (y << 1) % c;
    b >>= 1;
  }
  return x % c;
}

ll fastPow(ll x, ll n, ll MOD) {
  ll ret = 1;
  while (n) {
    if (n & 1)
      ret = mulmod(ret, x, MOD);
    x = mulmod(x, x, MOD);
    n >>= 1;
  }
  return ret;
}

bool isPrime(ll n) {
  vi a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

  if (binary_search(a.begin(), a.end(), n))
    return true;

  if ((n & 1) == 0)
    return false;

  int s = 0;
  for (ll m = n - 1; !(m & 1); ++s, m >>= 1)
    ;

  int d = (n - 1) / (1 << s);

  for (int i = 0; i < 7; i++) {
    ll fp = fastPow(a[i], d, n);
    bool comp = (fp != 1);
    if (comp)
```

```
      for (int j = 0; j < s; j++) {
        if (fp == n - 1) {
          comp = false;
          break;
        }

        fp = mulmod(fp, fp, n);
      }
      if (comp)
        return false;
    }
    return true;
}
```

7.3.2 Sieve of Eratosthenes

```

// O(n log log n)
vi sieve(int n) {
  vi primes;

  vector<bool> is_prime(n + 1, true);
  int limit = (int)floor(sqrt(n));
  repx(i, 2, limit + 1) if (is_prime[i]) for (int j = i * i; j <= n; j
      += i)
      is_prime[j] = false;

  repx(i, 2, n + 1) if (is_prime[i]) primes.eb(i);

  return primes;
}
```

7.3.3 trialDivision

```

// O(sqrt(n)/log(sqrt(n))+log(n))
vi trialDivision(int n, vi &primes) {
  vi factors;
  for (auto p : primes) {
    if (p * p > n)
      break;
    while (n % p == 0) {
      primes.pb(p);
      if ((n /= p) == 1)
        return factors;
    }
  }
  if (n > 1)
    factors.pb(n);

  return factors;
}
```

7.4 Others

7.4.1 Polynomials

```

template <class T> class Pol {
private:
  vector<T> cofs;
  int n;

public:
  Pol(vector<T> cofs) : cofs(cofs) { this->n = cofs.size() - 1; }

  Pol<T> operator+(const Pol<T> &o) {
    vector<T> n_cofs;
    if (n > o.n) {
      n_cofs = cofs;
      rep(i, o.n + 1) { n_cofs[i] += o.cofs[i]; }
    } else {
      n_cofs = o.cofs;
      rep(i, n + 1) { n_cofs[i] += cofs[i]; }
    }
    return Pol(n_cofs);
  }

  Pol<T> operator-(const Pol<T> &o) {
    vector<T> n_cofs;
    if (n > o.n) {
      n_cofs = cofs;
      rep(i, o.n + 1) { n_cofs[i] -= o.cofs[i]; }
    } else {
      n_cofs = o.cofs;
      rep(i, n + 1) {
        n_cofs[i] *= -1;
        n_cofs[i] += cofs[i];
      }
    }
    return Pol(n_cofs);
  }

  Pol<T>
  operator*(const Pol<T> &o) // Use Fast Fourier Transform when we
      implement it
  {
    vector<T> n_cofs(n + o.n + 1);
    rep(i, n + 1) {
      rep(j, o.n + 1) { n_cofs[i + j] += cofs[i] * o.cofs[j]; }
    }
    return Pol(n_cofs);
  }

  Pol<T> operator*(const T &o) {
    vector<T> n_cofs = cofs;
    for (auto &cof : n_cofs) {
      cof *= o;
```

```cpp
    }
    return Pol(n_cofs);
  }

  double operator()(double x) {
    double ans = 0;
    double temp = 1;
    for (auto cof : cofs) {
      ans += (double)cof * temp;
      temp *= x;
    }
    return ans;
  }

  Pol<T> integrate() {
    vector<T> n_cofs(n + 2);
    repx(i, 1, n_cofs.size()) { n_cofs[i] = cofs[i - 1] / T(i); }
    return Pol<T>(n_cofs);
  }

  double integrate(T a, T b) {
    Pol<T> temp = integrate();
    return temp(b) - temp(a);
  }

  friend ostream &operator<<(ostream &str, const Pol &a);
};

ostream &operator<<(ostream &strm, const Pol<double> &a) {
  bool flag = false;
  rep(i, a.n + 1) {
    if (a.cofs[i] == 0)
      continue;

    if (flag)
      if (a.cofs[i] > 0)
        strm << " + ";
      else
        strm << " - ";
    else
      flag = true;
    if (i > 1) {
      if (abs(a.cofs[i]) != 1)
        strm << abs(a.cofs[i]);
      strm << "x^" << i;
    } else if (i == 1) {
      if (abs(a.cofs[i]) != 1)
        strm << abs(a.cofs[i]);
      strm << "x";
    } else {
      strm << a.cofs[i];
    }
  }
  return strm;
}
```

7.4.2 Factorial Factorization

```cpp

// O(n)
umap<ll, int> factorialFactorization(int n, vi &primes) {
  umap<ll, int> p2e;
  for (auto p : primes) {
    if (p > n)
      break;
    int e = 0;
    ll tmp = n;
    while ((tmp /= p) > 0)
      e += tmp;
    if (e > 0)
      p2e[p] = e;
  }
  return p2e;
}
```

8 Geometry

8.1 Vectors/Points

```cpp
const double PI = acos(-1);

struct Point {
  double x, y;

  Point &operator+=(const Point &o) {
    this->x += o.x;
    this->y += o.y;
    return *this;
  }
  Point &operator-=(const Point &o) {
    this->x -= o.x;
    this->y -= o.y;
    return *this;
  }
  Point operator+(const Point &o) { return {x + o.x, y + o.y}; }
  Point operator-(const Point &o) { return {x - o.x, y - o.y}; }
  Point operator*(const double &o) { return {x * o, y * o}; }
  bool operator==(const Point &o) { return x == o.x and y == o.y; }
  double norm2() { return x * x + y * y; }
  double norm() { return sqrt(norm2()); }
  double dot(const Point &o) { return x * o.x + y * o.y; }
  double cross(const Point &o) { return x * o.y - y * o.x; }
  double angle() {
    double angle = atan2(y, x);
    if (angle < 0)
      angle += 2 * PI;
    return angle;
  }

  Point Unit() { return {x / norm(), y / norm()}; }
```

```cpp
};
/* ========================================================= */
/* 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;
};
#define MAXN (int)1e6               // Example
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;
}
/* =================== */
/* Circle Intersection */
/* =================== */
struct Circle {
  double x, y, r;
};
bool is_fully_outside(double r1, double r2, double d_sqr) {
  double tmp = r1 + r2;
  return d_sqr > tmp * tmp;
}
bool is_fully_inside(double r1, double r2, double d_sqr) {
  if (r1 > r2)
    return false;
  double tmp = r2 - r1;
  return d_sqr < tmp * tmp;
}
bool do_circles_intersect(Circle &c1, Circle &c2) {
  double dx = c1.x - c2.x;
  double dy = c1.y - c2.y;
  double d_sqr = dx * dx + dy * dy;
  if (is_fully_inside(c1.r, c2.r, d_sqr))
    return false;
  if (is_fully_inside(c2.r, c1.r, d_sqr))
    return false;
  if (is_fully_outside(c1.r, c2.r, d_sqr))
```

```
    return false;
  return true;
}
/* ===================== */
/* Point - Line distance */
/* ===================== */
// get distance between p and projection of p on line <- a - b ->
double point_line_dist(Point &p, Point &a, Point &b) {
  Point d = b - a;
  double t = d.dot(p - a) / d.norm2();
  return (a + d * t - p).norm();
}
/* ======================== */
/* Point - Segment distance */
/* ======================== */
// get distance between p and truncated projection of p on segment a ->
    b
double point_segment_dist(Point &p, Point &a, Point &b) {
  if (a == b)
    return (p - a).norm(); // segment is a single Point
  Point d = b - a;      // direction
  double t = d.dot(p - a) / d.norm2();
  if (t <= 0)
    return (p - a).norm(); // truncate left
  if (t >= 1)
    return (p - b).norm(); // truncate right
  return (a + d * t - p).norm();
}
/* ====================================== */
/* Straight Line Hashing (integer coords) */
/* ====================================== */
// task: given 2 points p1, p2 with integer coordinates, output a unique
// representation {a,b,c} such that a*x + b*y + c = 0 is the equation
// of the straight line defined by p1, p2. This representation must be
// unique for each straight line, no matter which p1 and p2 are sampled.
struct Line {
  int a, b, c;
};
int gcd(int a, int b) { // greatest common divisor
  a = abs(a);
  b = abs(b);
  while (b) {
    int c = a;
    a = b;
    b = c % b;
  }
  return a;
}
Line getLine(Point p1, Point p2) {
  int a = p1.y - p2.y;
  int b = p2.x - p1.x;
  int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
  int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
  int f = gcd(a, gcd(b, c)) * sgn;
  a /= f;
  b /= f;
  c /= f;
  return {a, b, c};
}
```

8.2 Calculate Areas

8.2.1 Integration via Simpson's Method

```
// O(Evaluate f)=g(f)
// Numerical Integration of f in interval [a,b]
double simpsons_rule(function<double(double)> f, double a, double b) {
  double c = (a + b) / 2;
  double h3 = abs(b - a) / 6;
  return h3 * (f(a) + 4 * f(c) + f(b));
}

// O(n g(f))
// Integrate f between a and b, using intervals of length (b-a)/n
double simpsons_rule(function<double(double)> f, double a, double b, int
    n) {
  // n sets the precision for the result
  double ans = 0;
  double step = 0, h = (b - a) / n;
  rep(i, n) {
    ans += simpsons_rule(f, step, step + h);
    step += h;
  }
  return ans;
}
```

8.2.2 Green's Theorem

```
// Line integrals for calculating areas with green's theorem

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);
}

double segment_integral(Point &a, Point &b) {
  return 0.5 * (a.x + b.x) * (b.y - a.y);
}
```

8.3 Convex Hull

```
// ------------------------------------------------
// 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, 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, n) {
    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;
}
```

8.4 Pick's Theorem

Given a simple polygon (no self intersections) in a lattice such that all vertices are grid points. Pick's theorem relates the Area A, points inside of the polygon i and the points of the border of the polygon b, in the following way:

$$A = i + \frac{b}{2} - 1$$

9 Strings

9.1 KMP

```

vi prefix(string &S)
{
    vector<int> p(S.size());
    p[0] = 0;
    for (int i = 1; i < S.size(); ++i)
    {
        p[i] = p[i - 1];
        while (p[i] > 0 && S[p[i]] != S[i])
            p[i] = p[p[i] - 1];
        if (S[p[i]] == S[i])
            p[i]++;
    }
    return p;
}

vi KMP(string &P, string &S)
{
    vector<int> pi = prefix(P);
    vi matches;
    int n = S.length(), m = P.length();
    int j = 0, ans = 0;
    for (int i = 0; i < n; ++i)
    {
        while (j > 0 && S[i] != P[j])
            j = pi[j - 1];
        if (S[i] == P[j])
            ++j;

        if (j == P.length())
        {
            /* This is where KMP found a match
             * we can calculate its position on S by using i - m + 1
             * or we can simply count it
             */
```

```
            ans += 1; // count the number of matches
            matches.eb(i - m + 1); // store the position of those matches
            // return; we can return on the first match if needed
            // this must stay the same
            j = pi[j - 1];
        }
    }
    return matches; // can be modified to return number of matches or location
}
```

9.2 Rolling Hashing

```

const int MAXLEN = 1e6;

class rollingHashing {
  static const ull base = 127;
  static const vector<ull> primes;
  static vector<vector<ull>> POW;

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

  static void init(int a) {
    if (POW.size() <= a + 1) {
      POW.eb(MAXLEN, 1);
    }
    repx(i, 1, MAXLEN) POW[a][i] = mul(POW[a][i], base, a);
  }

  static void init() { rep(i, primes.size()) init(i); }

  vector<vector<ull>> h;
  int len;
  rollingHashing(string &s) {
    len = s.size();
    h.assign(primes.size(), vector<ull>(len, 0));
    rep(a, primes.size()) {
      h[a][0] = s[0] - 'a'; // Assuming alphabetic alphabet
      repx(i, 1, len) h[a][i] = add(s[i] - 'a', mul(h[a][i - 1], base, a
          ), a);
    }
  }

  ull hash(int i, int j, int a) // Inclusive-Exclusive [i,i)?
  {
    if (i == 0)
      return h[a][j - 1];
    return add(h[a][j - 1], primes[a] - mul(h[a][i - 1], POW[a][j - i],
        a), a);
  }

  ull hash(int i, int j) // Supports at most two primes
  {
    return hash(i, j, 1) << 32 |
           hash(i, j, 0); // Using that 1e18<__LONG_LONG_MAX__
  }

  ull hash() { return hash(0, len); } // Also supports at most two primes
};

const vector<ull> rollingHashing ::primes({(ull)1e9 + 7,
                                           (ull)1e9 + 9}); // Add more if needed
```

9.3 Trie

```

/* Implementation from: https://pastebin.com/fyqsH65k */
struct TrieNode {
  int leaf;   // number of words that end on a TrieNode (allows for duplicate
              // words)
  int height; // height of a TrieNode, root starts at height = 1, can be changed
              // with the default value of constructor
  // number of words that pass through this node,
  // ask root node for this count to find the number of entries on the whole
  // Trie all nodes have 1 as they count the words than end on themselves (ie
  // leaf nodes count themselves)
  int count;
  TrieNode *parent; // pointer to parent TrieNode, used on erasing entries
  map<char, TrieNode *> child;
  TrieNode(TrieNode *parent = NULL, int height = 1)
      : parent(parent), leaf(0), height(height),
        count(0), // change to -1 if leaf nodes are to have count 0 insead of 1
        child() {}
};

/**
 * Complexity: O(|key| * log(k))
 */
TrieNode *trie_find(TrieNode *root, const string &str) {
  TrieNode *pNode = root;
  for (string::const_iterator key = str.begin(); key != str.end(); key
      ++) {
    if (pNode->child.find(*key) == pNode->child.end())
      return NULL;
    pNode = pNode->child[*key];
  }
  return (pNode->leaf) ? pNode : NULL; // returns only whole word
  // return pNode; // allows to search for a suffix
}
```

```cpp

/**
 * Complexity: O(|key| * log(k))
 */
void trie_insert(TrieNode *root, const string &str) {
  TrieNode *pNode = root;
  root->count += 1;
  for (string::const_iterator key = str.begin(); key != str.end(); key
       ++) {
    if (pNode->child.find(*key) == pNode->child.end())
      pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
    pNode = pNode->child[*key];
    pNode->count += 1;
  }
  pNode->leaf += 1;
}

/**
 * Complexity: O(|key| * log(k))
 */
void trie_erase(TrieNode *root, const string &str) {
  TrieNode *pNode = root;
  string::const_iterator key = str.begin();
  for (; key != str.end(); key++) {
    if (pNode->child.find(*key) == pNode->child.end())
      return;
    pNode = pNode->child[*key];
  }
  pNode->leaf -= 1;
  pNode->count -= 1;
  while (pNode->parent != NULL) {
    if (pNode->child.size() > 0 || pNode->leaf)
      break;
    pNode = pNode->parent, key--;
    pNode->child.erase(*key);
    pNode->count -= 1;
  }
}
```

9.4 Suffix Tree

```cpp

struct Node {
  // map<int,int> children;
  vector<int> children;
  int suffix_link;
  int start;
  int end;

  Node(int start, int end) : start(start), end(end) {
    children.resize(27, -1);
    suffix_link = 0;
  }
  inline bool has_child(int i) {
    // return children.find(i) != children.end();
    return children[i] != -1;
  }
};

struct SuffixTree {
  int size;
  int i;
  vector<int> suffix_array;
  vector<Node> tree;
  inline int length(int index) {
    if (tree[index].end == -1)
      return i - tree[index].start + 1;
    return tree[index].end - tree[index].start + 1;
  }
  // se puede usar string& s
  SuffixTree(vector<int> &s) {
    size = s.size();
    tree.emplace_back(-1, -1);
    int remaining_suffix = 0;
    int active_node = 0;
    int active_edge = -1;
    int active_length = 0;
    for (i = 0; i < size; ++i) {
      int last_new = -1;
      remaining_suffix++;
      while (remaining_suffix > 0) {
        if (active_length == 0)
          active_edge = i;
        if (!tree[active_node].has_child(s[active_edge])) {
          tree[active_node].children[s[active_edge]] = tree.size();
          tree.emplace_back(i, -1);
          if (last_new != -1) {
            tree[last_new].suffix_link = active_node;
            last_new = -1;
          }
        } else {
          int next = tree[active_node].children[s[active_edge]];
          if (active_length >= length(next)) {
            active_edge += length(next);
            active_length -= length(next);
            active_node = next;
            continue;
          }
          if (s[tree[next].start + active_length] == s[i]) {
            if (last_new != -1 and active_node != 0) {
              tree[last_new].suffix_link = active_node;
            }
            active_length++;
            break;
          }
          int split_end = tree[next].start + active_length - 1;
          int split = tree.size();
          tree.emplace_back(tree[next].start, split_end);
          tree[active_node].children[s[active_edge]] = split;
          int new_leaf = tree.size();
```

```cpp
          tree.emplace_back(i, -1);
          tree[split].children[s[i]] = new_leaf;
          tree[next].start += active_length;
          tree[split].children[s[tree[next].start]] = next;
          if (last_new != -1) {
            tree[last_new].suffix_link = split;
          }
          last_new = split;
        }
        remaining_suffix--;
        if (active_node == 0 and active_length > 0) {
          active_length--;
          active_edge = i - remaining_suffix + 1;
        } else if (active_node != 0) {
          active_node = tree[active_node].suffix_link;
        }
      }
    }
    i = size - 1;
  }
  vector<int> lcp;
  // last for lcp
  void dfs(int node, int &index, int depth, int min_depth) {
    if (tree[node].end == -1 and node != 0) {
      suffix_array[index] = size - depth;
      if (index != 0) {
        lcp[index - 1] = min_depth;
      }
      index++;
    }
    for (auto it : tree[node].children) {
      // if(i.second != -1){
      //    dfs(i.second,index,depth + length(i.second));
      //    min_depth = depth;
      //}
      if (it != -1) {
        dfs(it, index, depth + length(it), min_depth);
        min_depth = depth;
      }
    }
  }
  void build_suffix_array() {
    suffix_array.resize(size, 0);
    lcp.resize(size, 0);
    int index = 0;
    int depth = 0;
    dfs(0, index, 0, 0);
  }

  // pensado para map<int,int>, pero puede modificarse para vector<int>
  bool match(string &a, string &base) {
    int active_node = 0;
    int active_length = 0;
    int active_char = -1;
    for (int i = 0; i < a.size();) {
      if (active_length == 0) {
        if (!tree[active_node].has_child(a[i]))
          return false;
        active_char = a[i];
        active_length++;
        i++;
        continue;
      }
      int next = tree[active_node].children[active_char];
      if (active_length == length(next)) {
        active_node = next;
        active_length = 0;
        active_char = -1;
        continue;
      }
      if ((base)[tree[next].start + active_length] != a[i])
        return false;
      active_length++;
      i++;
    }
    return true;
  }
};
```