```cpp
// Never forget to use lambda gcd other than gcd to solve problem.
// error=SegmentTree(a,gcd),
// ok=SegmentTree(a,lambda::gcd);
#include <algorithm>
#include <bit>
#include <cassert>
#include <climits>
#include <cstddef>
#include <cstdint>
#include <deque>
#include <functional>
#include <iostream>
#include <iterator>
#include <limits>
#include <math.h>
#include <numeric>
#include <optional>
#include <ostream>
#include <queue>
#include <set>
#include <sys/types.h>
#include <tuple>
#include <type_traits>
#include <typeindex>
#include <variant>
#include <vector>
#define PI (3.14159265358979323846)
#include <array>
#include <chrono>
#include <tuple>
#include <unordered_map>
#include <unordered_set>
#include <utility>
using i32 = int32_t;
using i64 = int64_t;
using u32 = uint32_t;
using u64 = uint64_t;
using u8 = uint8_t;
const i64 MOD = 1000000007;
template <typename T> using maxheap = std::priority_queue<T>;
template <typename T>
using minheap = std::priority_queue<T, std::vector<T>, std::greater<T>>;
template <typename T> class Vec;
template <typename T> class Set;
template <typename T1, typename T2> class Pair;
```

```cpp
template <typename T> using VVec = Vec<Vec<T>>;
using Vi64 = Vec<i64>;
using Si64 = Set<i64>;
using VVi64 = Vec<Vec<i64>>;
using Pi64 = Pair<i64, i64>;
template <typename T> using limit = std::numeric_limits<T>;
using namespace std;
using namespace std::placeholders;

template <typename T1, typename T2> class Pair : public std::tuple<T1, T2> {
public:
  using std::tuple<T1, T2>::tuple;
  T1 &fir() { return get<0>(this); }
  T1 &fir() const { return get<0>(this); }
  T1 &sec() { return get<1>(this); }
  T1 &sec() const { return get<1>(this); }
};
// Loop macros for brevity
#define RANGE(i, a, b) for (int(i) = (a); (i) < (b); ++(i))
#define RANGEB(i, a, b, s) for (i64 i = (a); (i) < (b) - (s) + 1; (i) += (s))
#define RANGES(i, a, b, s) for (i64 i = (a); (i) < (b); (i) += (s))
#define RANGEREV(i, a, b) for (int(i) = (b) - 1; (i) >= (a); --(i))
#define RANGEREVS(i, a, b, s) for (i64 i = (b) - 1; (i) >= (a); (i) -= (s))

#define LOOP(index, value, vector)                                             \
  for (i64(index) = 0; (index) < (vector).size(); (index)++)                   \
    if (auto && (value) = (vector)[index]; true)
#define LOOP2(index, value1, value2, vector1, vector2)                         \
  for (i64(index) = 0; (index) < min((vector1).size(), (vector2).size());      \
       (index)++)                                                              \
    if (auto && (value1) = (vector1)[index];                                   \
        auto && (value2) = (vector2)[index]; true)
#define LOOPW2(index, value1, value2, vector1)                                 \
  for (i64(index) = 0; (index) < (vector1).size() - 1; (index)++)              \
    if (auto && (value1) = (vector1)[index];                                   \
        auto && (value2) = (vector2)[(index) + 1]; true)

namespace lambda {

const auto add = [](auto x, auto y) { return x + y; };
const auto mul = [](auto x, auto y) { return x * y; };
const auto max_by = [](auto func, auto x, auto y) {
  return func(x) < func(y) ? y : x;
};
const auto min_by = [](auto func, auto x, auto y) {
  return func(x) < func(y) ? x : y;
```

```cpp
};

const auto max = [](auto x, auto y) { return std::max(x, y); };
const auto min = [](auto x, auto y) { return std::min(x, y); };
const auto gcd = [](auto x, auto y) { return std::gcd(x, y); };
const auto lcm = [](auto x, auto y) { return std::lcm(x, y); };
const auto equal = [](auto x, auto y) { return x == y; };
const auto n_eq = [](auto x, auto y) { return x != y; };
} // namespace lambda
// auto add_n(i64 n) {
//   return [n](auto o) { return n + o; };
// }
template <typename Tuple, i64 N>
typename enable_if<N == 1, void>::type __tuple_print(std::ostream &os,
                                                     const Tuple &t) {
  os << std::get<0>(t);
}

template <typename Tuple, std::size_t N>
typename enable_if<N != 1, void>::type __tuple_print(std::ostream &os,
                                                     const Tuple &t) {
  __tuple_print<Tuple, N - 1>(os, t);
  os << ", " << std::get<N - 1>(t);
}
template <typename T> struct IsTuple : std::false_type {};

template <typename... Args>
struct IsTuple<std::tuple<Args...>> : std::true_type {};
template <typename Tuple, i64 Index>
typename enable_if<Index == tuple_size<Tuple>::value, istream &>::type
__tuple_input(std::istream &is, Tuple &tuple) {
  return is;
}
template <typename Tuple, std::size_t Index = 0>
typename enable_if<Index != tuple_size<Tuple>::value, istream &>::type
__tuple_input(std::istream &is, Tuple &tuple) {
  is >> std::get<Index>(tuple);
  return __tuple_input<Tuple, Index + 1>(is, tuple);
}

template <typename... Args>
std::istream &operator>>(std::istream &is, std::tuple<Args...> &tuple) {
  return __tuple_input(is, tuple);
}
template <typename T> struct Option : public std::optional<T> {
  using optional<T>::optional;
```

```cpp
  T unwarp_or(T data) { return this->value_or(data); }
  template <typename F> T operation_with(F f, T other) {
    if (this->has_value()) {
      return f(data, other);
    }
    return other;
  }
  template <typename F> Option<T> operation_with(F f, Option<T> other) {
    if (this->has_data && other.has_data) {
      return Option(f(this->data, other->data));
      ;
    }
    if (!(this->has_data || other.has_data)) {
      return Option();
    }
    if (this->has_data) {
      return this->data;
    }
    if (other.has_data) {
      return other.data;
    }
  }
  template <typename F> Option<T> map_to(F f) {
    if (this->has_value()) {
      return Option<T>(f(data));
    }
    return Option();
  }
  bool operator==(Option<T> other) {
    if (this->has_value() || other.has_value() == 0)
      return true;
    if (this->has_value() && other.has_value())
      return data == other.data;
    return false;
  }
};
template <typename T> T fromInput() {
  T ans;
  std::cin >> ans;
  return ans;
}
template <typename Iterator> struct Slice {
  // All the static check for the first,subspan etc should be done outside for
  // performance reason.
  using iterator_type = typename std::iterator_traits<Iterator>::value_type;
```

```
protected:
  Iterator _begin;
  Iterator _end;

public:
  Slice(Iterator begin, Iterator end) : _begin(begin), _end(end) {}
  template <typename T>
  Slice(T collection) : Slice(collection.begin(), collection.end()) {}
  void sort() { std::sort(_begin, _end); }
  template <typename F> void sort(F f) { std::sort(_begin, _end, f); }
  void sertRev() {
    std::sort(_begin, _end, [](auto x, auto y) { return x > y; });
  }
  template <typename F> void sort_by(F f) {
    std::sort(_begin, _end, [&f](auto x, auto y) { return f(x) < f(y); });
  }
  template <typename F> void sort_byRev(F f) {
    std::sort(_begin, _end, [&f](auto x, auto y) { return f(x) > f(y); });
  }
  Slice<Iterator> binary_search(iterator_type value) const {
    auto it = std::equal_range(_begin, _end, value);
    return Slice<Iterator>(it.first, it.second);
  }
  bool empty() const { return _begin == _end; }
  template <typename F>
  Slice<Iterator> binary_search(iterator_type value, F f) const {
    auto it = std::equal_range(_begin._end, value, f);
    return Slice<Iterator>(it.first, it.second);
  }
  template <typename F>
  Slice<Iterator> binary_search_by(iterator_type value, F f) const {
    auto it = std::equal_range(_begin, _end, value,
                               [&f](auto x, auto y) { return f(x) < f(y); });
    return Slice<Iterator>(it.first, it.second);
  }
  Iterator linear_search(iterator_type value) const {
    return std::find(_begin, _end, value);
  }
  template <typename Func>
  Iterator linear_searchBy(iterator_type value, Func func) const {
    return std::find_if(_begin, _end, [&](auto x) { return func(x) == value; });
  }
  i64 ls_Index(iterator_type value) const {
    auto ans = std::find(_begin, _end, value);
    if (ans == _end) {
      return -1;
```

```
  }
  return ans - _begin;
}
bool bs_contain(iterator_type value) const { // binary search contain
  return std::binary_search(_begin, _end, value);
}

i64 size() const { return _end - _begin; }
template <typename F> void transform(F f) { std::transform(_begin, _end, f); }
iterator_type reduce(iterator_type defalut_value) const {
  return std::accumulate(_begin, _end, defalut_value);
}
void accumulate() { std::partial_sum(_begin, _end, _begin); }
iterator_type max() const {
  return reduce(lambda::max, numeric_limits<iterator_type>::min());
}
template <typename Func> iterator_type max_by(Func f) const {
  return reduce([&](auto a, auto b) { return lambda::max_by(f, a, b); },
                numeric_limits<iterator_type>::max());
}
template <typename Func> iterator_type min_by(Func f) const {
  return reduce([&](auto a, auto b) { return lambda::min_by(f, a, b); },
                numeric_limits<iterator_type>::max());
}
iterator_type min() const {
  return reduce(lambda::min, numeric_limits<iterator_type>::max());
}
iterator_type sum() const { return reduce(0); }
template <typename F>
iterator_type reduce(F f, iterator_type defalut_value) const {
  return std::accumulate(_begin, _end, defalut_value, f);
}
template <typename F> iterator_type accumulate(F f) const {
  return std::partial_sum(_begin, _end, _begin, f);
}
Iterator begin() const { return _begin; }
Iterator end() const { return _end; }
Vec<iterator_type> collect(i64 shift = 0) const {
  Vec<iterator_type> ans;
  auto begin = this->_begin;
  ans.reserve(size() + shift);
  if (0 < shift) {
    RANGE(i, 0, shift) ans.push_back(iterator_type());
  } else {
    begin += (-shift);
    begin = begin < _end ? begin : _end;
```

```cpp
  }
  std::copy(begin, this->_end, std::back_inserter(ans));
  return ans;
}
Vec<iterator_type> sumArray() {
  auto ans = this->collect(1);
  ans.accumulate();
  return ans;
}
Option<iterator_type> get(i64 index) const {
  if (index < 0) {
    return Option<iterator_type>();
  }
  auto it = std::next(_begin, index);
  if (it >= _end) {
    return Option<iterator_type>();
  }
  return Option(*it);
}
iterator_type &at(i64 index) const { return *std::next(_begin, index); }
void reverse() { std::reverse(_begin, _end); }

Slice<Iterator> subspan(i64 index) const {
  return Slice<Iterator>(_begin + index, _end);
} // TODO:
Slice<Iterator> first(i64 index) const {
  return Slice<Iterator>(_begin, _begin + index);
}
Slice<Iterator> subspan(i64 index, i64 size) const {
  auto last = std::min(_begin + index + size, _end);
  return Slice<Iterator>(_begin + index, _begin + index + size);
}
void shift(i64 shiftSize) {
  _begin += shiftSize;
  _end += shiftSize;
}
void oneShift() {
  _begin++;
  _end++;
}
Slice<Iterator> shiftRet(i64 shiftSize) const {
  return Slice<Iterator>(_begin + shiftSize, _end + shiftSize);
}
void getInput() {
  for (auto &x : this) {
    cin >> x;
```

```
    }
  }
  Slice<Iterator> operator++() {
    auto ans = *this;
    this->shift(1);
    return ans;
  }
  Slice<Iterator> operator+=(i64 shiftSize) { this->shift(shiftSize); }
  bool operator==(Slice<Iterator> other) {
    assert(this->size() == other.size());
    return this->_begin == other._begin;
  }
  bool operator!=(Slice<Iterator> other) {
    assert(this->size() == other.size());
    return this->_begin != other._begin;
  }
};

template <typename T> Slice<typename T::reverse_iterator> rSlice(T collection) {
  return Slice(collection.rbegin(), collection.rend());
}
template <typename Collection, typename... Collections, typename F>
bool all_op(F f, Collection &&main, Collections &&...sub) {
  auto min_size_value = min(main.size(), sub.size()...);
  RANGE(i, 0, min_size_value) {
    if (!f(main[i], sub[i]...)) {
      return false;
    }
  }

  return true;
}

template <typename Collection, typename... Collections, typename F>
bool any_op(F f, Collection &&main, Collections &&...sub) {
  RANGE(i, 0, min(main.size(), sub.size()...)) {
    if (f(main[i], sub[i]...)) {
      return true;
    }
  }
  return false;
}
template <typename T> class Vec : public std::vector<T> {
public:
  using std::vector<T>::vector; // Inherit constructors
  using It = typename Vec<T>::iterator;
```

```cpp
using rIt = typename Vec<T>::reverse_iterator;
// Add any additional functions or modifications as needed
static Vec<T> fromInput(i64 n, i64 shift = 0) {
  auto v = Vec<T>(n + shift);
  RANGE(i, shift, n + shift) { cin >> v[i]; }
  return v;
}

Option<T> get(i64 index) {
  if (index < 0 || index >= this->size())
    return Option<T>();
  return Option<T>(this->at(index));
}
Option<T *> get_ref(i64 index) {
  if (index < 0 || index >= this->size())
    return Option<T *>(nullptr);
  return Option<T *>(&this->at(index));
}
static Vec<T> with_capacity(i64 size) {
  auto vec = Vec<T>();
  if (size <= 0) {
    return vec;
  }
  vec.reserve(size);
  return vec;
}
static Vec<T> range(i64 last) {
  auto vec = Vec<T>();
  vec.reserve(last);
  RANGE(i, 0, last) { vec.push_back(i); }
  return vec;
}
static Vec<T> range(i64 first, i64 last) {
  auto vec = Vec<T>();
  vec.reserve(last - first);
  RANGE(i, first, last) { vec.push_back(i); }
  return vec;
}
static Vec<T> range(i64 first, i64 last, i64 step) {
  auto vec = Vec<T>();
  vec.reserve(((last - first) + (step - 1)) / step);
  RANGEB(i, first, last, step) { vec.push_back(i); }
  return vec;
}
Slice<It> slice() { return Slice<It>(this->begin(), this->end()); }
Slice<It> firstSlice(i64 n) {
```

```
    return Slice(std::next(this->begin, n), this->end);
  }
  Slice<It> subSlice(i64 n) {
    return Slice(this->begin(), std::next(this->begin, n));
  }
  Slice<It> slice(i64 f, i64 l) {
    return Slice(std::next(this->begin(), f), std::next(this->begin(), l));
  }

  Slice<rIt> rslice() { return Slice<rIt>(this->rbegin(), this->rend()); }

  Slice<rIt> firstRSlice(i64 n) {
    return Slice(this->rbegin(), this->rbegin() + n);
  }

  Slice<rIt> subRslice(i64 n) {
    return Slice(this->rbegin() + n, this->rend());
  }

  Slice<rIt> rslice(i64 f, i64 l) {
    return Slice(this->rbegin() + f, this->rbegin() + l);
  }
  template <typename... Collections, typename F>
  bool all(F f, Collections... coll) {
    return all_op(f, *this, coll...);
  }

  template <typename... Collections, typename F>
  bool any(F f, Collections... coll) {
    return any_op(f, *this, coll...);
  }
};
// Recursive template to input elements of a tuple
// Base case to stop recursion
// Helper function to call TupleInput

template <typename... T> tuple<T...> tuple_input() {
  return fromInput<tuple<T...>>();
}
template <typename T>auto n2Input(){
    return tuple_input<T,T>();
}
template <typename T>auto n3Input(){
    return tuple_input<T,T,T>();
}
template <typename T>auto n4Input(){
```

```
    return tuple_input<T,T,T,T>();
}
int sign(const i64 value) { return value == 0 ? 0 : value >= 0 ? 1 : -1; }

template <typename... Args>
std::ostream &operator<<(std::ostream &os, const std::tuple<Args...> &t) {
  os << "(";
  __tuple_print<decltype(t), sizeof...(Args)>(os, t);
  os << ")";
  return os;
}

template <typename T, size_t I>
std::ostream &operator<<(std::ostream &os, const array<T, I> &t) {
  os << "[";
  for (auto x : t) {
    os << x << ",";
  }
  os << "]";
  return os;
}
template <typename T>
std::ostream &operator<<(std::ostream &os, const Vec<T> &t) {
  os << "[";
  for (auto x : t) {
    os << x << ",";
  }
  os << "]";
  return os;
}

// here are some initial coding for operation overloading
template <size_t I = 0, typename... T>
inline typename enable_if<(I == sizeof...(T)), tuple<T...>>::type
add_(const tuple<T...> &a, const tuple<T...> &b,
     tuple<T...> dummy = tuple<T...>()) {
  return dummy;
}

template <size_t I = 0, typename... T>
inline typename enable_if<(I < sizeof...(T)), tuple<T...>>::type
add_(const tuple<T...> &a, const tuple<T...> &b,
     tuple<T...> dummy = tuple<T...>()) {
  get<I>(dummy) = get<I>(a) + get<I>(b);
  return _add_<I + 1, T...>(a, b, dummy);
}
```

```
template <size_t I = 0, typename... T>
typename enable_if<(I == sizeof...(T)), tuple<T...>>::type
sub_(const tuple<T...> &a, const tuple<T...> &b,
     tuple<T...> dummy = tuple<T...>()) {
  return dummy;
}

template <size_t I = 0, typename... T>
typename enable_if<(I < sizeof...(T)), tuple<T...>>::type
sub_(const tuple<T...> &a, const tuple<T...> &b,
     tuple<T...> dummy = tuple<T...>()) {
  get<I>(dummy) = get<I>(a) - get<I>(b);
  return _add_<I + 1>(a, b, dummy);
}

template <size_t I = 0, typename... T>
typename enable_if<(I == sizeof...(T)), tuple<T...>>::type
mul_(const tuple<T...> &a, const tuple<T...> &b,
     tuple<T...> dummy = tuple<T...>()) {
  return dummy;
}

template <size_t I = 0, typename... T>
typename enable_if<(I < sizeof...(T)), tuple<T...>>::type
mul_(const tuple<T...> &a, const tuple<T...> &b,
     tuple<T...> dummy = tuple<T...>()) {
  get<I>(dummy) = get<I>(a) * get<I>(b);
  return _add_<I + 1>(a, b, dummy);
}

template <size_t I = 0, typename... T>
typename enable_if<(I == sizeof...(T)), tuple<T...>>::type
negative_(const tuple<T...> &a, const tuple<T...> &b,
          tuple<T...> dummy = tuple<T...>()) {
  return dummy;
}

template <size_t I = 0, typename... T>
typename enable_if<(I < sizeof...(T)), tuple<T...>>::type
negative_(const tuple<T...> &a, tuple<T...> dummy = tuple<T...>()) {
  get<I>(dummy) = -get<I>(a);
  return _add_<I + 1>(a, dummy);
}

// the over layer code
```

```cpp
template <typename... T>
tuple<T...> operator+(const tuple<T...> &a, const tuple<T...> &b) {
  return add_(a, b);
}

template <typename... T>
tuple<T...> operator-(const tuple<T...> &a, const tuple<T...> &b) {
  return sub_(a, b);
}

template <typename... T>
tuple<T...> operator*(const tuple<T...> &a, const tuple<T...> &b) {
  return mul_(a, b);
}

template <typename... T> tuple<T...> operator-(const tuple<T...> &a) {
  return negative_(a);
}

template <typename T, size_t N>
std::array<T, N> operator+(const std::array<T, N> &arr1,
                           const std::array<T, N> &arr2) {
  std::array<int, N> result;
  for (size_t i = 0; i < N; ++i) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}

template <typename T, size_t N>
std::array<T, N> operator-(const std::array<T, N> &arr1,
                           const std::array<T, N> &arr2) {
  std::array<int, N> result;
  for (size_t i = 0; i < N; ++i) {
    result[i] = arr1[i] - arr2[i];
  }
  return result;
}

template <typename T, size_t N>
std::array<int, N> operator*(const std::array<T, N> &arr1,
                             const std::array<T, N> &arr2) {
  std::array<int, N> result;
  for (size_t i = 0; i < N; ++i) {
    result[i] = arr1[i] * arr2[i];
  }
```

```
  return result;
}

template <typename T, size_t N>
std::array<int, N> operator/(const std::array<T, N> &arr1,
                             const std::array<T, N> &arr2) {
  std::array<int, N> result;
  for (size_t i = 0; i < N; ++i) {
    if (arr2[i] == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
    }
    result[i] = arr1[i] / arr2[i];
  }
  return result;
}
template <typename T, size_t N>
std::array<T, N> operator-(const std::array<T, N> &arr) {
  std::array<T, N> result;
  for (size_t i = 0; i < N; ++i) {
    result[i] = -arr[i];
  }
  return result;
}
template <typename T, size_t N>
std::array<T, N> &operator+=(std::array<T, N> &arr1,
                             const std::array<T, N> &arr2) {
  for (size_t i = 0; i < N; ++i) {
    arr1[i] += arr2[i];
  }
  return arr1;
}

template <typename T, size_t N>
std::array<T, N> &operator-=(std::array<T, N> &arr1,
                             const std::array<T, N> &arr2) {
  for (size_t i = 0; i < N; ++i) {
    arr1[i] -= arr2[i];
  }
  return arr1;
}

template <typename T, size_t N>
std::array<T, N> &operator*=(std::array<T, N> &arr1,
                             const std::array<T, N> &arr2) {
  for (size_t i = 0; i < N; ++i) {
    arr1[i] *= arr2[i];
```

```
  }
  return arr1;
}

template <typename T, size_t N>
std::array<T, N> &operator/=(std::array<T, N> &arr1,
                             const std::array<T, N> &arr2) {
  for (size_t i = 0; i < N; ++i) {
    if (arr2[i] == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
    }
    arr1[i] /= arr2[i];
  }
  return arr1;
}
template <typename T> Vec<T> operator+(const Vec<T> &vec1, const Vec<T> &vec2) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    result[i] = vec1[i] + vec2[i];
  }
  return result;
}

template <typename T> Vec<T> operator-(const Vec<T> &vec1, const Vec<T> &vec2) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    result[i] = vec1[i] - vec2[i];
  }
  return result;
}

template <typename T> Vec<T> operator*(const Vec<T> &vec1, const Vec<T> &vec2) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    result[i] = vec1[i] * vec2[i];
  }
  return result;
}

template <typename T> Vec<T> operator/(const Vec<T> &vec1, const Vec<T> &vec2) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    if (vec2[i] == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
    }
    result[i] = vec1[i] / vec2[i];
```

```
  }
  return result;
}

template <typename T> Vec<T> operator-(const Vec<T> &vec) {
  Vec<T> result(vec.size());
  for (size_t i = 0; i < vec.size(); ++i) {
    result[i] = -vec[i];
  }
  return result;
}

template <typename T> Vec<T> &operator+=(Vec<T> &vec1, const Vec<T> &vec2) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    vec1[i] += vec2[i];
  }
  return vec1;
}

template <typename T> Vec<T> &operator-=(Vec<T> &vec1, const Vec<T> &vec2) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    vec1[i] -= vec2[i];
  }
  return vec1;
}

template <typename T> Vec<T> &operator*=(Vec<T> &vec1, const Vec<T> &vec2) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    vec1[i] *= vec2[i];
  }
  return vec1;
}

template <typename T> Vec<T> &operator/=(Vec<T> &vec1, const Vec<T> &vec2) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    if (vec2[i] == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
    }
    vec1[i] /= vec2[i];
  }
  return vec1;
}

template <typename T, size_t N>
std::array<T, N> operator+(const std::array<T, N> &arr, const T &scalar) {
  std::array<T, N> result;
```

```cpp
  for (size_t i = 0; i < N; ++i) {
    result[i] = arr[i] + scalar;
  }
  return result;
}

template <typename T, size_t N>
std::array<T, N> operator-(const std::array<T, N> &arr, const T &scalar) {
  std::array<T, N> result;
  for (size_t i = 0; i < N; ++i) {
    result[i] = arr[i] - scalar;
  }
  return result;
}

template <typename T, size_t N>
std::array<T, N> operator*(const std::array<T, N> &arr, const T &scalar) {
  std::array<T, N> result;
  for (size_t i = 0; i < N; ++i) {
    result[i] = arr[i] * scalar;
  }
  return result;
}

template <typename T, size_t N>
std::array<T, N> operator/(const std::array<T, N> &arr, const T &scalar) {
  std::array<T, N> result;
  if (scalar == 0) {
    throw std::runtime_error("Division by zero is not allowed.");
  }
  for (size_t i = 0; i < N; ++i) {
    result[i] = arr[i] / scalar;
  }
  return result;
}

template <typename T, size_t N>
std::array<T, N> operator+(const T &scalar, const std::array<T, N> &arr) {
  return arr + scalar; // Commutative property
}

template <typename T, size_t N>
std::array<T, N> operator-(const T &scalar, const std::array<T, N> &arr) {
  std::array<T, N> result;
  for (size_t i = 0; i < N; ++i) {
    result[i] = scalar - arr[i];
```

```
  }
  return result;
}

template <typename T, size_t N>
std::array<T, N> operator*(const T &scalar, const std::array<T, N> &arr) {
  return arr * scalar; // Commutative property
}

template <typename T, size_t N>
std::array<T, N> operator/(const T &scalar, const std::array<T, N> &arr) {
  std::array<T, N> result;
  for (size_t i = 0; i < N; ++i) {
    if (arr[i] == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
    }
    result[i] = scalar / arr[i];
  }
  return result;
}
template <typename T> Vec<T> operator+(const Vec<T> &vec1, const T &scalar) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    result[i] = vec1[i] + scalar;
  }
  return result;
}
template <typename T> Vec<T> operator+(const T &scalar, const Vec<T> &vec1) {
  return vec1 + scalar;
}

template <typename T> Vec<T> operator-(const Vec<T> &vec1, const T &scalar) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    result[i] = vec1[i] - scalar;
  }
  return result;
}

template <typename T> Vec<T> operator*(const Vec<T> &vec1, const T &scalar) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    result[i] = vec1[i] * scalar;
  }
  return result;
}
```

```cpp
template <typename T> Vec<T> operator/(const Vec<T> &vec1, const T &scalar) {
  Vec<T> result(vec1.size());
  for (size_t i = 0; i < vec1.size(); ++i) {
    if (scalar == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
    }
    result[i] = vec1[i] / scalar;
  }
  return result;
}

template <typename T> Vec<T> &operator+=(Vec<T> &vec1, const T &scalar) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    vec1[i] += scalar;
  }
  return vec1;
}

template <typename T> Vec<T> &operator-=(Vec<T> &vec1, const T &scalar) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    vec1[i] -= scalar;
  }
  return vec1;
}

template <typename T> Vec<T> &operator*=(Vec<T> &vec1, const T &scalar) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    vec1[i] *= scalar;
  }
  return vec1;
}

template <typename T> Vec<T> &operator/=(Vec<T> &vec1, const T &scalar) {
  for (size_t i = 0; i < vec1.size(); ++i) {
    if (scalar == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
    }
    vec1[i] /= scalar;
  }
  return vec1;
}

template <typename T> struct Matrix {
  Vec<Vec<T>> data;
```

```
Matrix(Vec<Vec<T>> elements) : data(elements) {}
Matrix(i64 rows, i64 coloum, T default_value = 0)
    : data(Vec<Vec<T>>(rows, Vec<T>(coloum, default_value))) {}
// Addition of two matrices
Matrix<T> operator+(const Matrix<T> &other) const {
  Vec<Vec<T>> result(data.size(), Vec<T>(data[0].size()));
  for (size_t i = 0; i < data.size(); ++i) {
    for (size_t j = 0; j < data[0].size(); ++j) {
      result[i][j] = data[i][j] + other.data[i][j];
    }
  }
  return Matrix<T>(result);
}

// Subtraction of two matrices
Matrix<T> operator-(const Matrix<T> &other) const {
  Vec<Vec<T>> result(data.size(), Vec<T>(data[0].size()));
  for (size_t i = 0; i < data.size(); ++i) {
    for (size_t j = 0; j < data[0].size(); ++j) {
      result[i][j] = data[i][j] - other.data[i][j];
    }
  }
  return Matrix<T>(result);
}

// Scalar multiplication of a matrix
Matrix<T> operator*(const T &scalar) const {
  Vec<Vec<T>> result(data.size(), Vec<T>(data[0].size()));
  for (size_t i = 0; i < data.size(); ++i) {
    for (size_t j = 0; j < data[0].size(); ++j) {
      result[i][j] = data[i][j] * scalar;
    }
  }
  return Matrix<T>(result);
}

// Matrix multiplication
Matrix<T> operator*(const Matrix<T> &other) const {
  if (data[0].size() != other.data.size()) {
    throw std::invalid_argument("Incompatible matrix dimensions");
  }
  Vec<Vec<T>> result(data.size(), Vec<T>(other.data[0].size()));
  for (size_t i = 0; i < data.size(); ++i) {
    for (size_t j = 0; j < other.data[0].size(); ++j) {
      T sum = 0;
      for (size_t k = 0; k < data[0].size(); ++k) {
```

```cpp
          sum += data[i][k] * other.data[k][j];
        }
        result[i][j] = sum;
      }
    }
    return Matrix<T>(result);
  }

  // Display the matrix
  void display() const {
    for (const auto &row : data) {
      for (const auto &elem : row) {
        std::cout << elem << " ";
      }
      std::cout << '\n';
    }
  }
  Vec<T> operator[](i64 i) { return data[i]; }
};
void fast_io() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);
}
template <typename T> using vec2d = array<T, 2>;
template <typename T> using vec3d = array<T, 3>;

template <typename T, typename F> struct SegmentTree {
  SegmentTree() {}
  SegmentTree(Vec<Vec<T>> core, F f) : core(std::move(core)), f(f) {}
  SegmentTree(Vec<T> input, F f) : f(f) {
    while (input.size() > 1) {
      Vec<T> temp;
      RANGEB(i, 0, input.size(), 2) {
        temp.push_back(f(input[i], input[i + 1]));
      }
      this->core.push_back(std::move(input));
      input = std::move(temp);
    }
    this->core.push_back(std::move(input));
  }

  T query(i64 L, i64 R, i64 level = 0) {
    tuple<bool, T> def = tuple(false, T());
    auto defset = [&def, this](auto other) {
      if (get<0>(def) == false) {
```

```
        def = tuple(true, other);
      } else {
        get<1>(def) = f(get<1>(def), other);
      }
    };
    if (L == R) {
      return core[level][R];
    }
    if (L > R) {
      return T();
    }
    if (R % 2 == 0) {
      defset(core[level][R]);
      R--;
    }
    if (L % 2 == 1) {
      defset(core[level][L]);
      L++;
    }
    defset(query(L / 2, R / 2, level + 1));
    return get<1>(def);
  }
  void change(i64 pointer, T value) {
    for (auto &x : this->core) {
      x[pointer] = value;
      auto other_pointer = pointer % 2 == 0 ? pointer + 1 : pointer - 1;
      if (other_pointer >= x.size())
        break;
      value = f(x[pointer], x[other_pointer]);
      pointer /= 2;
    }
  }

private:
  Vec<Vec<T>> core;
  const F f;
};
template <typename Key, class Val>
class HashMap : public std::unordered_map<Key, Val> {
public:
  using std::unordered_map<Key, Val>::unordered_map; // Inherit constructors
  // Add any additional functions or modifications as needed
  bool contains(Key key) { return this->find(key) != this->end(); }
};

const tuple<Vec<i32>, Vec<i32>> prime_factorization(const i32 range) {
```

```
  Vec<i32> prime_factors;
  if (range <= 0) {
    return tuple(Vec<i32>(), Vec<i32>());
  }
  auto factors = Vec<i32>::with_capacity(range + 5);
  factors.push_back(0);
  factors.push_back(1);
  RANGE(i, 2, range + 1) {
    bool prime = true;
    for (auto x : prime_factors) {
      if (i % x == 0) {
        factors.push_back(x);
        prime = false;
      }
    }
    if (prime) {
      factors.push_back(i);
      prime_factors.push_back(i);
    }
  }
  return tuple(prime_factors, factors);
}
const auto [primes, factors] = prime_factorization(1);

tuple<Vec<i64>, Vec<i64>> number_factors(i64 num) {
  Vec<i64> ans1;
  Vec<i64> ans2;
  while (num != 1) {
    i64 p = factors[num];
    i64 n = 0;
    while (num % p == 0) {
      n++;
      num /= p;
    }
    ans1.push_back(p);
    ans2.push_back(n);
  }
  return tuple(ans1, ans2);
}
template <typename T> class Set : public std::set<T> {
  using set<T>::set;
  using Iterator = set<T>::iterator;

public:
  bool contains(T element) { return this->find(element) != this->end(); }
  void mutate(Iterator element, T object) {
```

```
    assert(element != this->end());
    this->erase(element);
    this->insert(object);
  }
  void remove(T element) {
    auto index = this->find(element);
    if (index != this->end())
      std::set<T>::erase(index);
  }
  void mutate(T element, T object) {
    auto ele = this->find(element);
    mutate(ele, object);
  }
  Option<T> findClosestSmallorEqual(T value) {
    auto it = this->lower_bound(value);
    if (it == this->begin() && value < *it)
      return std::nullopt;
    if (it == this->end() || *it < value)
      --it;
    return *it;
  }
  Option<T> findClosestSmall(T value) {
    auto it = this->lower_bound(value);
    if (it == this->begin())
      return std::nullopt;
    --it;
    *it;
  }
  Option<T> findClosestLargerEqual(T value) {
    auto it = this->lower_bound(value);
    if (it == this->end())
      return std::nullopt;
    return *it;
  }
  Option<T> findClosestLarger(T value) {
    auto it = this->upper_bound(value);
    if (it == this->end())
      return std::nullopt;
    return *it;
  }
};
int ilog2(unsigned int value) {
  int result = 0;
  while (value >>= 1) { // Right shift until value is 0
    ++result;
  }
```

```cpp
  return result;
}
int maxFactor(i64 value,i64 divisor) {
  int result = 0;
  while (value % divisor == 0) {
    result++;
    value /= divisor;
  }
  return result;
}
i64 modExp(i64 base, i64 exp, i64 mod = MOD) {
  long long result = 1;
  base = base % mod;

  if (exp < 0) {
    // Calculate modular inverse of base using Fermat's Little Theorem
    base = modExp(base, mod - 2, mod);
    exp = -exp;
  }

  while (exp > 0) {
    if (exp % 2 == 1) {
      result = (result * base) % mod;
    }
    base = (base * base) % mod;
    exp /= 2;
  }

  return result;
}
void sol() {
  i64 n = fromInput<i64>();
  auto slice = Vi64::fromInput(n);
  i64 Csum = 0;
  i64 Ct2sum = 0;
  i64 Cmax = -1;
  RANGE(i, 0, n) {
    int temp = maxFactor(slice[i],2);
    slice[i] >>= temp;
    Ct2sum += temp % MOD;
    Csum += slice[i] % MOD;
    Cmax = max(Cmax, slice[i]);
    cout << (Csum + (Cmax * ((1 << Ct2sum) - 1))) % MOD << ' ';
  }
  cout << '\n';
}
```

```
int main() {
  fast_io();
  i32 N = fromInput<i32>();
  RANGE(_, 0, N) { sol(); }
}
```