Contents

1 Misc 1
1.1 Contest 1
1.1.1 Makefile 1
1.1.2 Default Code 1
1.2 How Did We Get Here? 1
1.2.1 Macros 1
1.2.2 constexpr 1
1.2.3 Bump Allocator 1
1.3 Tools 1
1.3.1 Floating Point Binary Search 1
1.3.2 SplitMix64 1
1.3.3 <random> 2
1.4 Algorithms 2
1.4.1 Bit Hacks 2
1.4.2 Aliens Trick 2
1.4.3 Hilbert Curve 2

2 Data Structures 2
2.1 GNU PBDS 2
2.2 Segment Tree (ZKW) 2
2.3 Wavelet Matrix 2

3 Math 3
3.1 Number Theory 3
3.1.1 Mod Struct 3
3.1.2 Miller-Rabin 3
3.1.3 Extended GCD 3
3.1.4 Chinese Remainder Theorem 3
3.2 Combinatorics 3
3.2.1 Matroid Intersection 3

4 Geometry 3
4.1 Point 3
4.1.1 Quarternion 4

1. Misc

1.1. Contest

1.1.1. Makefile

```
.PRECIOUS: ./p%

%: p%
	ulimit -s unlimited && ./$<
p%: p%.cpp
	g++ -o $@ $< -std=c++17 -Wall -Wextra -Wshadow \
		-fsanitize=address,undefined
```

1.1.2. Default Code

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

#define pb        push_back
#define eb        emplace_back
#define F         first
#define S         second
#define SZ(v)     ((int)(v).size())
#define ALL(v)    (v).begin(), (v).end()
#define MEM(a, b) memset(a, b, sizeof a)
#define unpair(p) (p).F][(p).S

using namespace std;
using ll = long long;
using ld = long double;
using LL = __int128;
using pii = pair<int, int>;
using pll = pair<ll, ll>;

int main() { ios::sync_with_stdio(0), cin.tie(0); }
```

1.2. How Did We Get Here?

1.2.1. Macros

Use vectorizations and math optimizations at your own peril. For gcc≥9, there are `[[likely]]` and `[[unlikely]]` attributes. Call gcc with `-fopt-info-optimized-missed-optall` for optimization info.

```
#define _GLIBCXX_DEBUG            1 // for debug mode
#define _GLIBCXX_SANITIZE_VECTOR 1 // for asan on vectors
#pragma GCC optimize("O3", "unroll-loops")
#pragma GCC optimize("fast-math")
#pragma GCC target("avx,avx2,abm,bmi,bmi2") // tip: `lscpu`
// before a loop
#pragma GCC unroll 16 // 0 or 1 -> no unrolling
#pragma GCC ivdep
```

1.2.2. constexpr

Some default limits in gcc (7.x - trunk):

- constexpr recursion depth: 512
- constexpr loop iteration per function: 262 144
- constexpr operation count per function: 33 554 432
- template recursion depth: 900 (gcc *might* segfault first)

```
constexpr array<int, 10> fibonacci{[] {
  array<int, 10> a{};
  a[0] = a[1] = 1;
  for (int i = 2; i < 10; i++) a[i] = a[i - 1] + a[i - 2];
  return a;
}()};
static_assert(fibonacci[9] == 55, "CE");

template <typename F, typename INT, INT... S>
constexpr void for_constexpr(integer_sequence<INT, S...>,
                             F &&func) {
  int _[] = {(func(integral_constant<INT, S>{}), 0)...};
}
// example
template <typename... T> void print_tuple(tuple<T...> t) {
  for_constexpr(make_index_sequence<sizeof...(T)>{},
                [&](auto i) { cout << get<i>(t) << '\n'; });
}
```

1.2.3. Bump Allocator

```
// global bump allocator
char mem[256 << 20]; // 256 MB
size_t rsp = sizeof mem;
void *operator new(size_t s) {
  assert(s < rsp); // MLE
  return (void *)&mem[rsp -= s];
}
void operator delete(void *) {}

// bump allocator for STL / pbds containers
char mem[256 << 20];
size_t rsp = sizeof mem;
template <typename T> struct bump {
  typedef T value_type;
  bump() {}
  template <typename U> bump(U, ...) {}
  T *allocate(size_t n) {
    rsp -= n * sizeof(T);
    rsp &= 0 - alignof(T);
    return (T *)(mem + rsp);
  }
  void deallocate(T *, size_t n) {}
};
```

1.3. Tools

1.3.1. Floating Point Binary Search

```
union di {
  double d;
  ull i;
};
bool check(double);
// binary search in [L, R) with relative error 2^-eps
double binary_search(double L, double R, int eps) {
  di l = {L}, r = {R}, m;
  while (r.i - l.i > 1LL << (52 - eps)) {
    m.i = (l.i + r.i) >> 1;
    if (check(m.d)) r = m;
    else l = m;
  }
  return l.d;
}
```

1.3.2. SplitMix64

```
using ull = unsigned long long;
inline ull splitmix64(ull x) {
  // change to `static ull x = SEED;` for DRBG
  ull z = (x += 0x9E3779B97F4A7C15);
  z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9;
  z = (z ^ (z >> 27)) * 0x94D049BB133111EB;
  return z ^ (z >> 31);
}
```

1.3.3. <random>

```cpp
#ifdef __unix__
random_device rd;
mt19937_64 RNG(rd());
#else
const auto SEED = chrono::high_resolution_clock::now()
                      .time_since_epoch()
                      .count();
mt19937_64 RNG(SEED);
#endif
// random uint_fast64_t: RNG();
// uniform random of type T (int, double, ...) in [l, r]:
// uniform_int_distribution<T> dist(l, r); dist(RNG);
```

1.4. Algorithms

1.4.1. Bit Hacks

```cpp
// next permutation of x as a bit sequence
ull next_bits_permutation(ull x) {
  ull c = __builtin_ctzll(x), r = x + (1 << c);
  return (r ^ x) >> (c + 2) | r;
}
// iterate over all (proper) subsets of bitset s
void subsets(ull s) {
  for (ull x = s; x;) { --x &= s; /* do stuff */ }
}
```

1.4.2. Aliens Trick

```cpp
// min dp[i] value and its i (smallest one)
pll get_dp(int cost);
ll aliens(int k, int l, int r) {
  while (l != r) {
    int m = (l + r) / 2;
    auto [f, s] = get_dp(m);
    if (s == k) return f - m * k;
    if (s < k) r = m;
    else l = m + 1;
  }
  return get_dp(l).first - l * k;
}
```

1.4.3. Hilbert Curve

```cpp
ll hilbert(ll n, int x, int y) {
  ll res = 0;
  for (ll s = n / 2; s; s >>= 1) {
    int rx = !!(x & s), ry = !!(y & s);
    res += s * s * ((3 * rx) ^ ry);
    if (ry == 0) {
      if (rx == 1) x = s - 1 - x, y = s - 1 - y;
      swap(x, y);
    }
  }
  return res;
}
```

2. Data Structures

2.1. GNU PBDS

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/priority_queue.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

// most of std::map + order_of_key, find_by_order
template <typename T, typename U = null_type>
using ordered_map = tree<T, U, std::less<>, rb_tree_tag,
                         tree_order_statistics_node_update>;
// useful tags: rb_tree_tag, splay_tree_tag

template <typename T> struct myhash {
  size_t operator()(T x) const; // splitmix, bswap(x*R), ...
};
// most of std::unordered_map, but faster (needs good hash)
template <typename T, typename U = null_type>
using hash_table = gp_hash_table<T, U, myhash<T>>;

// most of std::priority_queue + merge
using heap = priority_queue<int, std::less<>>;
// useful tags: pairing_heap_tag, binary_heap_tag,
// binomial_heap_tag
```

2.2. Segment Tree (ZKW)

```cpp
struct segtree {
  using T = int;
  T f(T a, T b) { return a + b; } // any monoid operation
  static constexpr T ID = 0;      // identity element
  int n;
  vector<T> v;
  segtree(int n_) : n(n_), v(2 * n, ID) {}
  segtree(vector<T> &a) : n(a.size()), v(2 * n, ID) {
    copy_n(a.begin(), n, v.begin() + n);
    for (int i = n - 1; i > 0; i--)
      v[i] = f(v[i << 1], v[i << 1 | 1]);
  }
  void update(int i, T x) {
    for (v[i += n] = x; i /= 2;)
      v[i] = f(v[i << 1], v[i << 1 | 1]);
  }
  T query(int l, int r) {
    T tl = ID, tr = ID;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) tl = f(tl, v[l++]);
      if (r & 1) tr = f(v[--r], tr);
    }
    return f(tl, tr);
  }
};
```

2.3. Wavelet Matrix

```cpp
#pragma GCC target("popcnt,abm,bmi,bmi2")
#include <immintrin.h>

// T is unsigned. You might want to compress values first
template <typename T> struct wavelet_matrix {
  static_assert(is_unsigned_v<T>, "only unsigned T");
  struct bit_vector {
    static constexpr uint W = 64;
    uint n, cnt0;
    vector<ull> bits;
    vector<uint> sum;
    bit_vector(uint n_)
        : n(n_), bits(n / W + 1), sum(n / W + 1) {}
    void build() {
      for (uint j = 0; j != n / W; ++j)
        sum[j + 1] = sum[j] + _mm_popcnt_u64(bits[j]);
      cnt0 = rank0(n);
    }
    void set_bit(uint i) { bits[i / W] |= 1ULL << i % W; }
    bool get_bit(uint i) const {
      return !!(bits[i / W] & 1ULL << (i % W));
    }
    uint rank1(uint i) const {
      return sum[i / W] +
             _mm_popcnt_u64(_bzhi_u64(bits[i / W], i % W));
    }
    uint rank0(uint i) const { return i - rank1(i); }
  };
  uint n, lg;
  vector<bit_vector> b;
  wavelet_matrix(uint _n = 0) : n(_n) {}
  wavelet_matrix(const vector<T> &a) : n(a.size()) {
    lg =
    __lg(max(*max_element(a.begin(), a.end()), T(1))) + 1;
    b.assign(lg, n);
    vector<T> cur = a, nxt(n);
    for (int h = lg; h--;) {
      for (uint i = 0; i < n; ++i)
        if (cur[i] & (T(1) << h)) b[h].set_bit(i);
      b[h].build();
      int il = 0, ir = b[h].cnt0;
      for (uint i = 0; i < n; ++i)
        nxt[(b[h].get_bit(i) ? ir : il)++] = cur[i];
      swap(cur, nxt);
    }
  }
  // query k-th smallest (0-based) in a[l, r)
  T kth(uint l, uint r, uint k) const {
    T res = 0;
    for (int h = lg; h--;) {
      uint tl = b[h].rank0(l), tr = b[h].rank0(r);
      if (k >= tr - tl) {
        k -= tr - tl;
        l += b[h].cnt0 - tl;
        r += b[h].cnt0 - tr;
        res |= T(1) << h;
      } else l = tl, r = tr;
    }
    return res;
  }
  // count of i in [l, r) with a[i] < u
```

```
  uint count(uint l, uint r, T u) const {
    if (u >= T(1) << lg) return r - l;
    uint res = 0;
    for (int h = lg; h--;) {
      uint tl = b[h].rank0(l), tr = b[h].rank0(r);
      if (u & (T(1) << h)) {
        l += b[h].cnt0 - tl;
        r += b[h].cnt0 - tr;
        res += tr - tl;
      } else l = tl, r = tr;
    }
    return res;
  }
};
```

3. Math

3.1. Number Theory

3.1.1. Mod Struct

A list of safe primes: 26003, 27767, 28319, 28979, 29243, 29759, 30467 910927547, 919012223, 947326223, 990669467, 1007939579, 1019126699 929760389146037459, 975500632317046523, 989312547895528379

NTT prime p	$p-1$	primitive root
65537	$1 \ll 16$	3
998244353	$119 \ll 23$	3
2748779069441	$5 \ll 39$	3
1945555039024054273	$27 \ll 56$	5

```
template <typename T> struct M {
  static T MOD; // change to constexpr if already known
  T v;
  M() : v(0) {}
  M(T x) {
    v = (-MOD <= x && x < MOD) ? x : x % MOD;
    if (v < 0) v += MOD;
  }
  explicit operator T() const { return v; }
  bool operator==(const M &b) const { return v == b.v; }
  bool operator!=(const M &b) const { return v != b.v; }
  M operator-() { return M(-v); }
  M operator+(M b) { return M(v + b.v); }
  M operator-(M b) { return M(v - b.v); }
  M operator*(M b) { return M((__int128)v * b.v % MOD); }
  M operator/(M b) { return *this * (b ^ (MOD - 2)); }
  friend M operator^(M a, ll b) {
    M ans(1);
    for (; b; b >>= 1, a *= a)
      if (b & 1) ans *= a;
    return ans;
  }
  friend M &operator+=(M &a, M b) { return a = a + b; }
  friend M &operator-=(M &a, M b) { return a = a - b; }
  friend M &operator*=(M &a, M b) { return a = a * b; }
  friend M &operator/=(M &a, M b) { return a = a / b; }
};
using Mod = M<int>;
template <> int Mod::MOD = 1'000'000'007;
int &MOD = Mod::MOD;
```

3.1.2. Miller-Rabin

Requires: Mod Struct

```
// checks if Mod::MOD is prime
bool is_prime() {
  if (MOD < 2 || MOD % 2 == 0) return MOD == 2;
  Mod A[] = {2, 7, 61}; // for int values (< 2^31)
  // ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
  int s = __builtin_ctzll(MOD - 1), i;
  for (Mod a : A) {
    Mod x = a ^ (MOD >> s);
    for (i = 0; i < s && (x + 1).v > 2; i++) x *= x;
    if (i && x != -1) return 0;
  }
  return 1;
}
```

3.1.3. Extended GCD

```
// [p, q, g]: p * a + q * b == g == gcd(a, b)
// g is not guaranteed to be positive when a < 0 or b < 0!
tuple<ll, ll, ll> extgcd(ll a, ll b) {
  ll s = 1, t = 0, u = 0, v = 1;
  while (b) {
    ll q = a / b;
    swap(a -= q * b, b);
    swap(s -= q * t, t);
    swap(u -= q * v, v);
  }
  return {s, u, a};
}
```

3.1.4. Chinese Remainder Theorem

Requires: Extended GCD

```
ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  auto [x, y, g] = extgcd(m, n);
  assert((a - b) % g == 0); // no solution
  x = ((b - a) / g * x) % (n / g) * m + a;
  return x < 0 ? x + m / g * n : x;
}
```

3.2. Combinatorics

3.2.1. Matroid Intersection

This template assumes 2 weighted matroids of the same type, and that removing an element is much more expensive than checking if one can be added. **Remember to change the implementation details.**

The ground set is $0, 1, \ldots, n-1$, where element i has weight $w[i]$. For the unweighted version, remove weights and change BF/SPFA to BFS.

```
constexpr int N = 100;
constexpr int INF = 1e9;

struct Matroid {       // represents an independent set
  Matroid(bitset<N>);  // initialize from an independent set
  bool can_add(int);   // if adding will break independence
  Matroid remove(int); // removing from the set
};

auto matroid_intersection(int n, const vector<int> &w) {
  bitset<N> S;
  for (int sz = 1; sz <= n; sz++) {
    Matroid M1(S), M2(S);

    vector<vector<pii>> e(n + 2);
    for (int j = 0; j < n; j++)
      if (!S[j]) {
        if (M1.can_add(j)) e[n].emplace_back(j, -w[j]);
        if (M2.can_add(j)) e[j].emplace_back(n + 1, 0);
      }
    for (int i = 0; i < n; i++)
      if (S[i]) {
        Matroid T1 = M1.remove(i), T2 = M2.remove(i);
        for (int j = 0; j < n; j++)
          if (!S[j]) {
            if (T1.can_add(j)) e[i].emplace_back(j, -w[j]);
            if (T2.can_add(j)) e[j].emplace_back(i, w[i]);
          }
      }

    vector<pii> dis(n + 2, {INF, 0});
    vector<int> prev(n + 2, -1);
    dis[n] = {0, 0};
    // change to SPFA for more speed, if necessary
    bool upd = 1;
    while (upd) {
      upd = 0;
      for (int u = 0; u < n + 2; u++)
        for (auto [v, c] : e[u]) {
          pii x(dis[u].first + c, dis[u].second + 1);
          if (x < dis[v]) dis[v] = x, prev[v] = u, upd = 1;
        }
    }

    if (dis[n + 1].first < INF)
      for (int x = prev[n + 1]; x != n; x = prev[x])
        S.flip(x);
    else break;

    // S is the max-weighted independent set with size sz
  }
  return S;
}
```

4. Geometry

4.1. Point

```
template <typename T> struct P {
  T x, y;
  P(T x = 0, T y = 0) : x(x), y(y) {}
  bool operator<(const P &p) const {
    return tie(x, y) < tie(p.x, p.y);
  }
  bool operator==(const P &p) const {
    return tie(x, y) == tie(p.x, p.y);
  }
```

```
  P operator-() const { return {-x, -y}; }
  P operator+(P p) const { return {x + p.x, y + p.y}; }
  P operator-(P p) const { return {x - p.x, y - p.y}; }
  P operator*(T d) const { return {x * d, y * d}; }
  P operator/(T d) const { return {x / d, y / d}; }
  T dist2() const { return x * x + y * y; }
  double len() const { return sqrt(dist2()); }
  P unit() const { return *this / len(); }
  friend T dot(P a, P b) { return a.x * b.x + a.y * b.y; }
  friend T cross(P a, P b) { return a.x * b.y - a.y * b.x; }
  friend T cross(P a, P b, P o) {
    return cross(a - o, b - o);
  }
};
using pt = P<ll>;
```

4.1.1. Quarternion

```
constexpr double PI = 3.141592653589793;
constexpr double EPS = 1e-7;
struct Q {
  using T = double;
  T x, y, z, r;
  Q(T r = 0) : x(0), y(0), z(0), r(r) {}
  Q(T x, T y, T z, T r = 0) : x(x), y(y), z(z), r(r) {}
  friend bool operator==(const Q &a, const Q &b) {
    return (a - b).abs2() <= EPS;
  }
  friend bool operator!=(const Q &a, const Q &b) {
    return !(a == b);
  }
  Q operator-() { return Q(-x, -y, -z, -r); }
  Q operator+(const Q &b) const {
    return Q(x + b.x, y + b.y, z + b.z, r + b.r);
  }
  Q operator-(const Q &b) const {
    return Q(x - b.x, y - b.y, z - b.z, r - b.r);
  }
  Q operator*(const T &t) const {
    return Q(x * t, y * t, z * t, r * t);
  }
  Q operator*(const Q &b) const {
    return Q(r * b.x + x * b.r + y * b.z - z * b.y,
             r * b.y - x * b.z + y * b.r + z * b.x,
             r * b.z + x * b.y - y * b.x + z * b.r,
             r * b.r - x * b.x - y * b.y - z * b.z);
  }
  Q operator/(const Q &b) const { return *this * b.inv(); }
  T abs2() const { return r * r + x * x + y * y + z * z; }
  T len() const { return sqrt(abs2()); }
  Q conj() const { return Q(-x, -y, -z, r); }
  Q unit() const { return *this * (1.0 / len()); }
  Q inv() const { return conj() * (1.0 / abs2()); }
  friend T dot(Q a, Q b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
  }
  friend Q cross(Q a, Q b) {
    return Q(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z,
             a.x * b.y - a.y * b.x);
  }
  friend Q rotation_around(Q axis, T angle) {
    return axis.unit() * sin(angle / 2) + cos(angle / 2);
  }
  Q rotated_around(Q axis, T angle) {
    Q u = rotation_around(axis, angle);
    return u * *this / u;
  }
  friend Q rotation_between(Q a, Q b) {
    a = a.unit(), b = b.unit();
    if (a == -b) {
      // degenerate case
      Q ortho = abs(a.y) > EPS ? cross(a, Q(1, 0, 0))
                               : cross(a, Q(0, 1, 0));
      return rotation_around(ortho, PI);
    }
    return (a * (a + b)).conj();
  }
};
```