

Contents

1 Misc	
1.1 Contest	
1.1.1 Makefile	
1.2 How Did We Get Here?	
1.2.1 Macros	
1.2.2 Fast I/O	
1.2.3 constexpr	
1.2.4 Bump Allocator	
1.3 Tools	
1.3.1 Floating Point Binary Search	
1.3.2 SplitMix64	
1.3.3 <random>	
1.4 Algorithms	
1.4.1 Bit Hacks	
1.4.2 Aliens Trick	
1.4.3 Hilbert Curve	
1.4.4 Infinite Grid Knight Distance	
2 Data Structures	
2.1 GNU PBDS	
2.2 Segment Tree (ZKW)	
2.3 Wavelet Matrix	
3 Math	
3.1 Number Theory	
3.1.1 Mod Struct	
3.1.2 Miller-Rabin	
3.1.3 Extended GCD	
3.1.4 Chinese Remainder Theorem	
3.2 Combinatorics	
3.2.1 Matroid Intersection	
4 Geometry	
4.1 Point	
4.1.1 Quarternion	

1. Misc

1.1. Contest

1.1.1. Makefile

```
1 .PRECIOUS: ./p%
3 %: p%
   ulimit -s unlimited && ./<
5 p%: p%.cpp
   g++ -o $@ $< -std=c++17 -Wall -Wextra -Wshadow \
7   -fsanitize=address,undefined
```

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.

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

1.2.2. Fast I/O

```
1 struct scanner {
   static constexpr size_t LEN = 32 << 20;
3   char *buf, *buf_ptr, *buf_end;
   scanner()
5       : buf(new char[LEN]), buf_ptr(buf + LEN),
         buf_end(buf + LEN) {}
7   ~scanner() { delete[] buf; }
   char getc() {
9       if (buf_ptr == buf_end) [[unlikely]]
         buf_end = buf + fread_unlocked(buf, 1, LEN, stdin),
11        buf_ptr = buf;
         return *(buf_ptr++);
}
```

```
13 }
   char seek(char del) {
15     char c;
       while ((c = getc()) < del) {}
17     return c;
   }
19 void read(int &t) {
   bool neg = false;
   char c = seek('-');
   if (c == '-') neg = true, t = 0;
   else t = c ^ '0';
23     while ((c = getc()) >= '0') t = t * 10 + (c ^ '0');
       if (neg) t = -t;
25 }
27 };
   struct printer {
29     static constexpr size_t CPI = 21, LEN = 32 << 20;
   char *buf, *buf_ptr, *buf_end, *tbuf;
   char *int_buf, *int_buf_end;
   printer()
       : buf(new char[LEN]), buf_ptr(buf),
         buf_end(buf + LEN), int_buf(new char[CPI + 1]()),
         int_buf_end(int_buf + CPI - 1) {}
35   ~printer() {
       flush();
37       delete[] buf, delete[] int_buf;
   }
39   void flush() {
41       fwrite_unlocked(buf, 1, buf_ptr - buf, stdout);
       buf_ptr = buf;
43   }
   void write(const char &c) {
45       *buf_ptr = c;
       if (++buf_ptr == buf_end) [[unlikely]]
47           flush();
   }
49   void write(const char *s) {
       for (; *s != '\0'; ++s) write(*s);
51   }
   void write(int x) {
53       if (x < 0) write('-'), x = -x;
       if (x == 0) [[unlikely]]
55           return write('0');
       for (tbuf = int_buf_end; x != 0; --tbuf, x /= 10)
57           *tbuf = '0' + char(x % 10);
       write(++tbuf);
59   }
};
```

1.2.3. constexpr

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

- constexpr recursion depth: 512
- constexpr loop iteration per function: 262144
- constexpr operation count per function: 33554432
- template recursion depth: 900 (gcc *might* segfault first)

```
1 constexpr array<int, 10> fibonacci{[] {
   array<int, 10> a{};
3   a[0] = a[1] = 1;
   for (int i = 2; i < 10; i++) a[i] = a[i - 1] + a[i - 2];
5   return a;
   }()};
7 static_assert(fibonacci[9] == 55, "CE");
9 template <typename F, typename INT, INT... S>
   constexpr void for_constexpr(integer_sequence<INT, S...>,
11                               F &&func) {
       int _[] = {(func(integral_constant<INT, S>{}), 0)...};
13 }
   // example
15 template <typename... T> void print_tuple(tuple<T...> t) {
       for_constexpr(make_index_sequence<sizeof...(T)>{}),
17         [&](auto i) { cout << get<i>(t) << '\n'; }};
}
```

1.2.4. Bump Allocator

```
1 // global bump allocator
   char mem[256 << 20]; // 256 MB
3   size_t rsp = sizeof mem;
   void *operator new(size_t s) {
5       assert(s < rsp); // MLE
       return (void *)&mem[rsp -= s];
7   }
   void operator delete(void *) {}
9
   // bump allocator for STL / pbds containers
11 char mem[256 << 20];
   size_t rsp = sizeof mem;
```

```

13 template <typename T> struct bump {
14     typedef T value_type;
15     bump() {}
16     template <typename U> bump(U, ...) {}
17     T *allocate(size_t n) {
18         rsp -= n * sizeof(T);
19         rsp &= 0 - alignof(T);
20         return (T*)(mem + rsp);
21     }
22     void deallocate(T *, size_t n) {}
23 };

```

1.3. Tools

1.3.1. Floating Point Binary Search

```

1 union di {
2     double d;
3     ull i;
4 };
5 bool check(double);
6 // binary search in [L, R) with relative error 2^-eps
7 double binary_search(double L, double R, int eps) {
8     di l = {L}, r = {R}, m;
9     while (r.i - l.i > 1LL << (52 - eps)) {
10         m.i = (l.i + r.i) >> 1;
11         if (check(m.d)) r = m;
12         else l = m;
13     }
14     return l.d;
15 }

```

1.3.2. SplitMix64

```

1 using ull = unsigned long long;
2 inline ull splitmix64(ull x) {
3     // change to `static ull x = SEED;` for DRBG
4     ull z = (x += 0x9E3779B97F4A7C15);
5     z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9;
6     z = (z ^ (z >> 27)) * 0x94D049BB133111EB;
7     return z ^ (z >> 31);
8 }

```

1.3.3. <random>

```

1 #ifdef __unix__
2     random_device rd;
3     mt19937_64 RNG(rd());
4 #else
5     const auto SEED = chrono::high_resolution_clock::now()
6         .time_since_epoch()
7         .count();
8     mt19937_64 RNG(SEED);
9 #endif
10 // random uint_fast64_t: RNG();
11 // uniform random of type T (int, double, ...) in [l, r]:
12 // uniform_int_distribution<T> dist(l, r); dist(RNG);

```

1.4. Algorithms

1.4.1. Bit Hacks

```

1 // next permutation of x as a bit sequence
2 ull next_bits_permutation(ull x) {
3     ull c = __builtin_ctzll(x), r = x + (1 << c);
4     return (r ^ x) >> (c + 2) | r;
5 }
6 // iterate over all (proper) subsets of bitset s
7 void subsets(ull s) {
8     for (ull x = s; x;) { --x &= s; /* do stuff */ }
9 }

```

1.4.2. Aliens Trick

```

1 // min dp[i] value and its i (smallest one)
2 pll get_dp(int cost);
3 ll aliens(int k, int l, int r) {
4     while (l != r) {
5         int m = (l + r) / 2;
6         auto [f, s] = get_dp(m);
7         if (s == k) return f - m * k;
8         if (s < k) r = m;
9         else l = m + 1;
10    }
11    return get_dp(l).first - l * k;
12 }

```

1.4.3. Hilbert Curve

```

1 ll hilbert(ll n, int x, int y) {
2     ll res = 0;
3     for (ll s = n / 2; s; s >= 1) {
4         int rx = !(x & s), ry = !(y & s);
5         res += s * s * ((3 * rx) ^ ry);
6         if (ry == 0) {
7             if (rx == 1) x = s - 1 - x, y = s - 1 - y;
8             swap(x, y);
9         }
10    }
11    return res;
12 }

```

1.4.4. Infinite Grid Knight Distance

```

1 ll get_dist(ll dx, ll dy) {
2     if (++(dx = abs(dx)) > ++(dy = abs(dy))) swap(dx, dy);
3     if (dx == 1 && dy == 2) return 3;
4     if (dx == 3 && dy == 3) return 4;
5     ll lb = max(dy / 2, (dx + dy) / 3);
6     return ((dx ^ dy ^ lb) & 1) ? ++lb : lb;
7 }

```

2. Data Structures

2.1. GNU PBDS

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/priority_queue.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 using namespace __gnu_pbds;
5
6 // most std::map + order_of_key, find_by_order, split, join
7 template <typename T, typename U = null_type>
8 using ordered_map = tree<T, U, std::less<>, rb_tree_tag,
9     tree_order_statistics_node_update>;
10 // useful tags: rb_tree_tag, splay_tree_tag
11
12 template <typename T> struct myhash {
13     size_t operator()(T x) const; // splitmix, bswap(x*R), ...
14 };
15 // most of std::unordered_map, but faster (needs good hash)
16 template <typename T, typename U = null_type>
17 using hash_table = gp_hash_table<T, U, myhash<T>>;
18
19 // most std::priority_queue + modify, erase, split, join
20 using heap = priority_queue<int, std::less<>>;
21 // useful tags: pairing_heap_tag, binary_heap_tag,
22 // (rc)?binomial_heap_tag, thin_heap_tag

```

2.2. Segment Tree (ZKW)

```

1 struct segtree {
2     using T = int;
3     T f(T a, T b) { return a + b; } // any monoid operation
4     static constexpr T ID = 0; // identity element
5     int n;
6     vector<T> v;
7     segtree(int n_) : n(n_), v(2 * n, ID) {}
8     segtree(vector<T> &a) : n(a.size()), v(2 * n, ID) {
9         copy_n(a.begin(), n, v.begin() + n);
10        for (int i = n - 1; i > 0; i--)
11            v[i] = f(v[i * 2], v[i * 2 + 1]);
12    }
13    void update(int i, T x) {
14        for (v[i += n] = x; i /= 2;)
15            v[i] = f(v[i * 2], v[i * 2 + 1]);
16    }
17    T query(int l, int r) {
18        T tl = ID, tr = ID;
19        for (l += n, r += n; l < r; l /= 2, r /= 2) {
20            if (l & 1) tl = f(tl, v[l++]);
21            if (r & 1) tr = f(v[--r], tr);
22        }
23        return f(tl, tr);
24    }
25 };

```

2.3. Wavelet Matrix

```

1 #pragma GCC target("popcnt,bmi2")
2 #include <immintrin.h>
3
4 // T is unsigned. You might want to compress values first
5 template <typename T> struct wavelet_matrix {
6     static_assert(is_unsigned_v<T>, "only unsigned T");
7     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 operator[](uint i) const {
    return !(bits[i / W] & 1ULL << i % W);
}
uint rank1(uint i) const {
    return sum[i / W] +
        _mm_popcnt_u64(_bzhil_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][i] ? ir : il)++] = cur[i];
        swap(cur, nxt);
    }
}
T operator[](uint i) const {
    T res = 0;
    for (int h = lg; h--;)
        if (b[h][i])
            i += b[h].cnt0 - b[h].rank0(i), res |= T(1) << h;
        else i = b[h].rank0(i);
    return res;
}
// 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 >= 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

```

// returns (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

```

// for 0 <= a < m, 0 <= b < n, returns the smallest x >= 0
// such that x % m == a and x % n == b
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 $0, 1, \dots, n - 1$, where element i has weight $w[i]$. For the unweighted version, remove weights and change BF/SPFA to BFS.

```

1 constexpr int N = 100;
2 constexpr int INF = 1e9;
3
4 struct Matroid { // represents an independent set
5     Matroid(bitset<N>); // initialize from an independent set
6     bool can_add(int); // if adding will break independence
7     Matroid remove(int); // removing from the set
8 };
9
10 auto matroid_intersection(int n, const vector<int> &w) {
11     bitset<N> S;
12     for (int sz = 1; sz <= n; sz++) {
13         Matroid M1(S), M2(S);
14
15         vector<vector<pii>> e(n + 2);
16         for (int j = 0; j < n; j++)
17             if (!S[j]) {
18                 if (M1.can_add(j)) e[n].emplace_back(j, -w[j]);
19                 if (M2.can_add(j)) e[j].emplace_back(n + 1, 0);
20             }
21         for (int i = 0; i < n; i++)
22             if (S[i]) {
23                 Matroid T1 = M1.remove(i), T2 = M2.remove(i);
24                 for (int j = 0; j < n; j++)
25                     if (!S[j]) {
26                         if (T1.can_add(j)) e[i].emplace_back(j, -w[j]);
27                         if (T2.can_add(j)) e[j].emplace_back(i, w[i]);
28                     }
29             }
30
31         vector<pii> dis(n + 2, {INF, 0});
32         vector<int> prev(n + 2, -1);
33         dis[n] = {0, 0};
34         // change to SPFA for more speed, if necessary
35         bool upd = 1;
36         while (upd) {
37             upd = 0;
38             for (int u = 0; u < n + 2; u++)
39                 for (auto [v, c] : e[u]) {
40                     pii x(dis[u].first + c, dis[u].second + 1);
41                     if (x < dis[v]) dis[v] = x, prev[v] = u, upd = 1;
42                 }
43         }
44
45         if (dis[n + 1].first < INF)
46             for (int x = prev[n + 1]; x != n; x = prev[x])
47                 S.flip(x);
48         else break;
49
50         // S is the max-weighted independent set with size sz
51     }
52     return S;
53 }

```

4. Geometry

4.1. Point

```

1 template <typename T> struct P {
2     T x, y;
3     P(T x = 0, T y = 0) : x(x), y(y) {}
4     bool operator<(const P &p) const {
5         return tie(x, y) < tie(p.x, p.y);
6     }
7     bool operator==(const P &p) const {
8         return tie(x, y) == tie(p.x, p.y);
9     }
10    P operator-() const { return {-x, -y}; }
11    P operator+(P p) const { return {x + p.x, y + p.y}; }
12    P operator-(P p) const { return {x - p.x, y - p.y}; }
13    P operator*(T d) const { return {x * d, y * d}; }
14    P operator/(T d) const { return {x / d, y / d}; }
15    T dist2() const { return x * x + y * y; }
16    double len() const { return sqrt(dist2()); }
17    P unit() const { return *this / len(); }
18    friend T dot(P a, P b) { return a.x * b.x + a.y * b.y; }
19    friend T cross(P a, P b) { return a.x * b.y - a.y * b.x; }
20    friend T cross(P a, P b, P o) {
21        return cross(a - o, b - o);
22    }
23 };
24 using pt = P<ll>;

```

4.1.1. Quaternion

```

1 constexpr double PI = 3.141592653589793;
2 constexpr double EPS = 1e-7;
3 struct Q {

```

```

4     using T = double;
5     T x, y, z, r;
6     Q(T r = 0) : x(0), y(0), z(0), r(r) {}
7     Q(T x, T y, T z, T r = 0) : x(x), y(y), z(z), r(r) {}
8     friend bool operator==(const Q &a, const Q &b) {
9         return (a - b).abs2() <= EPS;
10    }
11    friend bool operator!=(const Q &a, const Q &b) {
12        return !(a == b);
13    }
14    Q operator-() const { return {-x, -y, -z, -r}; }
15    Q operator+(const Q &b) const {
16        return Q(x + b.x, y + b.y, z + b.z, r + b.r);
17    }
18    Q operator-(const Q &b) const {
19        return Q(x - b.x, y - b.y, z - b.z, r - b.r);
20    }
21    Q operator*(const T &t) const {
22        return Q(x * t, y * t, z * t, r * t);
23    }
24    Q operator*(const Q &b) const {
25        return Q(r * b.x + x * b.r + y * b.z - z * b.y,
26                r * b.y - x * b.z + y * b.r + z * b.x,
27                r * b.z + x * b.y - y * b.x + z * b.r,
28                r * b.r - x * b.x - y * b.y - z * b.z);
29    }
30    Q operator/(const Q &b) const { return *this * b.inv(); }
31    T abs2() const { return r * r + x * x + y * y + z * z; }
32    T len() const { return sqrt(abs2()); }
33    Q conj() const { return {-x, -y, -z, r}; }
34    Q unit() const { return *this * (1.0 / len()); }
35    Q inv() const { return conj() * (1.0 / abs2()); }
36    friend T dot(Q a, Q b) {
37        return a.x * b.x + a.y * b.y + a.z * b.z;
38    }
39    friend Q cross(Q a, Q b) {
40        return Q(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z,
41                a.x * b.y - a.y * b.x);
42    }
43    friend Q rotation_around(Q axis, T angle) {
44        return axis.unit() * sin(angle / 2) + cos(angle / 2);
45    }
46    Q rotated_around(Q axis, T angle) {
47        Q u = rotation_around(axis, angle);
48        return u * *this / u;
49    }
50    friend Q rotation_between(Q a, Q b) {
51        a = a.unit(), b = b.unit();
52        if (a == -b) {
53            // degenerate case
54            Q ortho = abs(a.y) > EPS ? cross(a, Q(1, 0, 0))
55                : cross(a, Q(0, 1, 0));
56            return rotation_around(ortho, PI);
57        }
58        return (a * (a + b)).conj();
59    }
60 };

```