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> 1
- 1.4 Algorithms 2
  - 1.4.1 Bit Hacks 2
  - 1.4.2 Aliens Trick 2
  - 1.4.3 Hilbert Curve 2

2 Math 2
- 2.1 Number Theory 2
  - 2.1.1 Mod Struct 2
  - 2.1.2 Miller-Rabin 2
- 2.2 Combinatorics 2
  - 2.2.1 Matroid Intersection 2

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

```cpp
#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$\geq$9, there are `[[likely]]` and `[[unlikely]]` attributes.
Call gcc with `-fopt-info-optimized-missed-optall` for optimization info.

```cpp
#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)

```cpp
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

```cpp
// 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

```cpp
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

```cpp
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

```
// 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

```
// 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

```
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. Math

2.1. Number Theory

2.1.1. 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;
```

2.1.2. 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;
}
```

2.2. Combinatorics

2.2.1. 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;

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);
    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; 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;
}
```