# Contents



# Header

```
typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;

typedef int8_t i8;
typedef int16_t i16;
typedef int32_t i32;
typedef int64_t i64;

typedef float f32;
typedef double f64;
typedef long double f80;

#define pb push_back
#define pf push_front
#define fst first
#define snd second
```

---

# Mathematics

## Number theory

Given $a, b$, finds $g = \gcd(a, b)$ and $u, v$ such that $ua + vb = g$
*Time*: $\mathcal{O}(\log ab)$

```
#include "../header.h"

array<i64, 3> extended_euclid(i64 a, i64 b) {
  if (b == 0)
    return {a, 1, 0};
  auto [g, x, y] = extended_euclid(b, a % b);
  return {g, y, x - y * (a / b)};
}
```

Finds $x^{-1} \bmod m$ in $\mathcal{O}(\log m)$.

```
optional<i64> inv(i64 x, i64 m) {
  auto [g, y, _] = extended_euclid(x, m);
  if (g != 1)
    return {};
  return (y >= 0 ? y % m : m - (-y) % m);
}
```

## Modular integers

Implements operations over the integers under a modulus.
*Time*: $\mathcal{O}(1)$ for $+$, $-$ and $*$.
*Time*: $\mathcal{O}(\log \text{mod})$ for / (**only if mod is prime**)

```
#include "../header.h"

template<int mod = MOD> struct mint {
  i64 x;

  mint inv() { // only prime mod
    mint i = 1, b = x;
    for (int e = mod - 2; e; e >>= 1) {
      if (e & 1) i *= b;
      b *= b;
    }
    return i;
  }

  mint operator+(mint o) {
    return (x + o.x) % mod;
  }
  mint& operator+=(mint o) {
    (x += o.x) %= mod;
    return *this;
  }
  mint operator-(mint o) {
    return (x - o.x + mod) % mod;
  }
  mint& operator-=(mint o) {
    (x += mod - o.x) %= mod;
    return *this;
  }
  mint operator*(mint o) {
    return x * o.x % mod;
  }
  mint& operator*=(mint o) {
    (x *= o.x) %= mod;
    return *this;
  }
  mint operator/(mint o) { // only prime mod
    return x * o.inv().x % mod;
  }
  mint& operator/=(mint o) { // only prime mod
    (x *= o.inv().x) %= mod;
    return *this;
  }
  mint(i64 x) : x(x % mod) { }
  mint() : x(0) { }
};
```

## Combinatorics

Computes $b^0, \ldots, b^{\text{maxn}}$ modulo mod.
*Time*: $\mathcal{O}(\text{maxn})$.

```
#include "../header.h"
#include "../math/modular_int.cpp"

template<i64 mod = MOD>
vector<mint<mod>> get_pows(i64 b, int maxn) {
  vector<mint<mod>> pows(maxn + 1, 1);
  for (int i = 1; i <= maxn; i++)
    pows[i] = pows[i - 1] * b;
  return pows;
}
```

Computes $0!, \ldots, \text{maxn}!$ and $(0!)^{-1}, \ldots, (\text{maxn}!)$ modulo mod.
*Time*: $\mathcal{O}(\text{maxn})$.

```
array<vector<mint<mod>>, 2> get_fact(int maxn) {
  vector<mint<mod>> f(maxn + 1, 1), i(maxn + 1);

  for (int j = 1; j <= maxn; j++)
    f[j] = f[j - 1] * j;
  i[maxn] = f[maxn].inv();
  for (int j = maxn - 1; j >= 0; j--)
    i[j] = i[j + 1] * (j + 1);
  return {f, i};
}
```

Computes $\binom{n}{\text{ks}_0, \ldots}$. The last element of ks can be omitted and it will be assumed to be $n - \sum \text{ks}_i$.
*Time*: $\mathcal{O}(1)$.

```
template<i64 mod = MOD>
mint<mod> multinom(i64 n, vector<i64> ks) {
  mint ans = fact[n];
```

```
  for (auto k: ks) {
    if (k < 0)
      return 0;
    n -= k;
    ans *= invs[k];
  }
  if (n < 0)
    return 0;
  return ans * invs[n];
}
```

Computes the $n$-th catalan number.
*Time*: $\mathcal{O}(1)$

```
template<i64 mod = MOD>
mint<mod> catalan(i64 n) {
  return multinom(2 * n, {n})
      - multinom(2 * n, {n - 1});
}
```

Counts the possible completions of a bracket sequence where only $n$ '(' and $m$ ')' are left to be placed.
*Time*: $\mathcal{O}(1)$

```
template<i64 mod = MOD>
mint<mod> catalan(i64 m, i64 n) {
  if (m > n || n < 0 || m < 0) return 0;
  return multinom(m + n, {m})
      - multinom(m + n, {m - 1});
}
```

## Permutations

Implements swaps in a permutation mantaining the inverse.
*Time*: $\mathcal{O}(N)$ construction and $\mathcal{O}(1)$ query.

```
struct Perm {
  int n;
  vector<int> perm, pos_of; // perm and inverse

  void swap(int i, int j) { // perm_i <-> perm_j
    ::swap(perm[i], perm[j]);
    ::swap(pos_of[perm[i]], pos_of[perm[j]]);
  }

  void invert() { ::swap(perm, pos_of); }

  Perm(int n) : n(n) {
    iota(perm.begin(), perm.end(), 0);
    iota(pos_of.begin(), pos_of.end(), 0);
  }
  Perm(const vector<int> &p) :
    n(p.size()), perm(p), pos_of(n)
  {
    for (int i = 0; i < n; i++)
      pos_of[perm[i]] = i;
  }
};
```

## Dihedral group

Implements operations over $D_n$ in $\mathcal{O}(1)$.

```
#include "../header.h"

struct Dihedral {
  i64 n, rot;
  bool flip;

  Dihedral inv() const {
    if (flip) return *this;
    return {n, (n - rot) % n, false};
  }

  Dihedral operator*(Dihedral o) const {
    if (flip) {
      o.flip ^= true;
      o.rot = (n - o.rot) % n;
    }
    o.rot = (o.rot + rot) % n;
    return o;
  }

  Dihedral(i64 n, i64 rot, bool flip) :
    n(n), rot(rot), flip(flip) { }
  Dihedral(i64 n) : Dihedral(n, 0, false) { }
};
```

---

# String algorithms

## Z-function

Builds the Z function of a string.
*Time*: $\mathcal{O}(N)$ where $N$ is the length of the string.

```
template<typename T>
vector<int> z_function(T s) {
  int n = s.size();
  vector<int> z(n);

  int l = 0, r = 0;
  for(int i = 1; i < n; i++) {
    if (i < r) z[i] = min(r - i, z[i - l]);
    while (
      i + z[i] < n && s[z[i]] == s[i + z[i]]
    ) z[i]++;
    if(i + z[i] > r) l = i, r = i + z[i];
  }
  return z;
}
```

## Aho-Corasick

Builds the Aho-Corasick automaton.
*Time*: $\mathcal{O}(N)$ where $N$ is the total length of the strings.
*Memory*: $\mathcal{O}(\Sigma N)$ where $\Sigma$ is the size of the alphabet.

```
template<int K = 26> class AhoCorasick {
  struct Node {
    Node* tr[K];        // transitions
    Node* suff;         // dictionary suffix
    vector<Node*> adj; // incoming dict suffixes

    Node() : suff(nullptr) {
      fill(tr, tr + K, nullptr);
    }
  };

  Node* root;
  vector<Node*> dict;

  Node* insert(const string &s) {
    Node* curr = root;
    for (auto c: s) {
      if (!curr->tr[c - 'a'])
        curr->tr[c - 'a'] = new Node;
      curr = curr->tr[c - 'a'];
    }

    return curr;
  }

  void get_suffixes() {
    queue<Node*> q;

    for (int i = 0; i < K; i++) {
      if (root->tr[i]) {
        root->tr[i]->suff = root;
        root->adj.push_back(root->tr[i]);
        q.push(root->tr[i]);
```

```
      } else {
        root->tr[i] = root;
      }
    }

    while (!q.empty()) {
      Node* curr = q.front(); q.pop();

      for (int i = 0; i < K; i++) {
        if (curr->tr[i]) {
          curr->tr[i]->suff = curr->suff->tr[i];
          curr->tr[i]->suff->adj
            .push_back(curr->tr[i]);
          q.push(curr->tr[i]);
        } else {
          curr->tr[i] = curr->suff->tr[i];
        }
      }
    }
  }

public:

  AhoCorasick(const vector<string> &words) {
    root = new Node;
    for (auto &word: words) {
      dict.push_back(insert(word));
    }
    get_suffixes();
  }
};
```

# Graph algorithms

## Triangulation Tree

Builds the tree of a triangulation. *Time*: $\mathcal{O}(N \log N)$, $\mathcal{O}(N)$ is counting sort is used.

```
vector<vector<int>> triangulation_tree(
  int n, vector<array<int, 2>> edges
) {
  for (auto &[u, v]: edges)
    if (u > v) swap(u, v);

  sort(
    edges.begin(), edges.end(),
    [](const auto &a, const auto &b) {
      return a[1] - a[0] < b[1] - b[0];
    }
  );

  vector<vector<int>> children;
  vector<pair<int, int>> nxt(n);

  for (int i = 0; i < n; i++)
    nxt[i] = {(i + 1) % n, -1};

  for (auto [l, r]: edges) {
    int curr = children.size();
    children.emplace_back();

    int i = l;
    while (i != r) {
      if (nxt[i].second != -1)
        children[curr].push_back(nxt[i].second);

      int p = i;
      i = nxt[i].first;
      nxt[p] = {-1, -1};
    }
    nxt[l] = {r, curr};
  }

  children.emplace_back();
  for (int i = 0; i < n; i++)
    if (nxt[i].second != -1)
      children.back().push_back(nxt[i].second);

  return children;
}
```