Team Notebook

June 23, 2022

Contents

1 Algebra

1.1 Fundamentals

1.1.1 Extended GCD

```
template <class Z>
constexpr Z extended_gcd(Z a, Z b, Z& x_ref, Z& y_ref) {
    x_ref = 1, y_ref = 0;
    Z x1 = 0, y1 = 1, tmp = 0, q = 0;
    while (b > 0) {
        q = a / b;
        tmp = a, a = b, b = tmp - (q * b);
        tmp = x_ref, x_ref = x1, x1 = tmp - (q * x1);
        tmp = y_ref, y_ref = y1, y1 = tmp - (q * y1);
    }
    return a;
}
```

1.1.2 Modular Binary Exponentiation (Power)

```
template <class B, class E, class M>
constexpr B power(B base, E expo, M mod = 0) {
    assert(expo >= 0);
    if (mod == 1) return 0;
    if (base == 0 || base == 1) return base;
    B res = 1;
    if (!mod) {
        while (expo) {
            if (expo & 1) res *= base;
            base *= base;
            expo >>= 1;
        }
    } else {
        assert(mod > 0);
        base %= mod;
        if (base <= 1) return base;
        while (expo) {
            if (expo & 1) res = (res * base) % mod;
            base = (base * base) % mod;
            expo >>= 1;
        }
    }
    return res;
}
```

1.2 Modular Arithmetic

1.2.1 Modular inverse

```
template <class Z>
constexpr Z inverse(Z num, Z mod) {
    assert(mod > 1);
    if (!(0 <= num && num < mod)) {
        num %= mod;
        if (num < 0) num += mod;
    }
    Z res = 1, tmp = 0;
    assert(extended_gcd(num, mod, res, tmp) == 1);
    if (res < 0) res += mod;
    return res;
}
```

2 Brute-force

2.1 Power Set

```
template <class T>
vector<vector<T>> power_set(const vector<T>& vec) {
    vector<vector<T>> res;
    list<T> buf;
    function<void(int)> recurse = [&](int i) -> void {
        if (i == vec.size()) {
            res.emplace_back(buf.begin(), buf.end());
            return;
        }
        recurse(i + 1);
        buf.push_back(vec[i]), recurse(i + 1), buf.pop_back()
            ;
    };
    recurse(0);
    return res;
}
```

3 Data Structures

3.1 Trees

3.1.1 Disjoint Set Union (DSU)

```
struct DSU {
    int n_nodes = 0;
    int n_components = 0;
    vector<int> component_size;
    vector<int> component_root;

    DSU(int n_nodes, bool make_all_nodes = false)
        : n_nodes(n_nodes),
          component_root(n_nodes, -1),
          component_size(n_nodes, 0) {
        if (make_all_nodes) {
            for (int i = 0; i < n_nodes; ++i) {
                make_node(i);
            }
        }
    }

    void make_node(int v) {
        if (component_root[v] == -1) {
            component_root[v] = v;
            component_size[v] = 1;
            ++n_components;
        }
    }

    int root(int v) {
        auto res = v;
        while (component_root[res] != res) {
            res = component_root[res];
        }
        while (v != res) {
            auto u = component_root[v];
            component_root[v] = res;
            v = u;
        }
        return res;
    }

    int connect(int u, int v) {
        u = root(u), v = root(v);
        if (u == v) return u;
        if (component_size[u] < component_size[v]) {
            swap(u, v);
        }
        component_root[v] = u;
        component_size[u] += component_size[v];
        --n_components;
    }
};
```

4 Starter

4.1 C++ Include GNU PBDS

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

template <class T>
using ordered_set = pbds::tree<T, pbds::null_type, std::less
    <T>,
                               pbds::rb_tree_tag,
                               pbds::
                                   tree_order_statistics_node_update
                                   >;
template <class K, class V>
using hash_map = pbds::gp_hash_table<K, V>;
```

4.2 C++ Starter

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr double eps = 1e-9;
constexpr int inf = 1 << 30;
constexpr int mod = 1e9 + 7;
constexpr int nmax = 1e6;

void runcase(int casen) {

    // cout << "Case " << casen << ": " << '\n';
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int ncases = 1;
    cin >> ncases; // Comment out for single-case tests
    for (int casen = 1; casen <= ncases; ++casen) {
        runcase(casen);
    }

    return 0;
}
```

5 String Processing

5.1 Fundamentals

5.1.1 Polynomial Rolling Hash (String Hashing)

```cpp
#include <bits/stdc++.h>
using namespace std;

namespace hash_utils {
    constexpr std::array<int, 10U>
        bases = {257, 263, 269, 271, 277, 281, 283, 293, 307,
             311};
    constexpr std::array<int, 10U>
        moduli = {1000000007, 1000000009, 1000000021,
             1000000033, 1000000087,
                 1000000093, 1000000097, 1000000103,
                      1000000123, 1000000181};
} // namespace hash_utils



template <std::size_t Dim, class Tp>
using require_valid_hash_params_t =
    std::enable_if_t<(1U <= Dim && Dim <= 3U) &&
                     (std::is_same<Tp, int>::value ||
                      std::is_same<Tp, std::int64_t>::value)>;

template <std::size_t Dim, class Tp = int,
          class = require_valid_hash_params_t<Dim, Tp>>
class Rolling_hash {
private:
    using Vec = std::vector<Tp>;

    static std::conditional_t<Dim == 1U, Tp, std::array<Tp,
        Dim>> base_;
    static std::conditional_t<Dim == 1U, Tp, std::array<Tp,
        Dim>> mod_;
    static std::conditional_t<Dim == 1U, Vec, std::array<Vec,
         Dim>> pow_;
    static std::conditional_t<Dim == 1U, Vec, std::array<Vec,
         Dim>> inv_;
    static int nchanges_;
    static bool ischanged_;

    std::conditional_t<Dim == 1U, Vec, std::array<Vec, Dim>>
        pref_;
    std::conditional_t<Dim == 1U, Vec, std::array<Vec, Dim>>
        suff_;
    int changeid_;
    bool ishashed_;
    bool isbidirect_;

    template <class T = Tp,
              std::enable_if_t<std::is_same<T, int>::value>* =
                   nullptr>
    static constexpr Tp mul(const Tp& a, const Tp& b, const
        Tp& mod) {
        return ((static_cast<std::int64_t>(a) * b) % mod);
    }

    template <class T = Tp,
              std::enable_if_t<std::is_same<T, std::int64_t>::
                   value>* = nullptr>
    static constexpr Tp mul(const Tp& a, const Tp& b, const
        Tp& mod) {
        long double prod = static_cast<long double>(a) * b;
        std::int64_t quot = prod / mod;
        return (prod - (quot * mod) + 1e-6);
    }

    template <class T = Tp,
              std::enable_if_t<std::is_same<T, int>::value>* =
                   nullptr>
    static constexpr Tp add(const Tp& a, const Tp& b, const
        Tp& mod) {
        return ((static_cast<std::int64_t>(a) + b) % mod);
    }

    template <class T = Tp,
              std::enable_if_t<std::is_same<T, std::int64_t>::
                   value>* = nullptr>
    static constexpr Tp add(const Tp& a, const Tp& b, const
        Tp& mod) {
        long double sum = static_cast<long double>(a) + b;
        std::int64_t quot = sum / mod;
        return (sum - (quot * mod) + 1e-6);
    }

    static constexpr Tp inverse(Tp a, const Tp& mod) {
        Tp b = mod, x = 1, y = 0;
        Tp x1 = 0, y1 = 1, tmp = 0, q = 0;
        while (b > 0) {
            q = a / b;
            tmp = a, a = b, b = tmp - (q * b);
            tmp = x, x = x1, x1 = tmp - (q * x1);
            tmp = y, y = y1, y1 = tmp - (q * y1);
        }
        assert(a == 1);
        if (x < 0) x += mod;
```

```
    return x;
  }

  template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
      >* = nullptr>
  static constexpr void normalize_kernel(std::size_t len) {
    if (!(ischanged_ || pow_.size() < len)) {
      return;
    }
    auto cur_len = ischanged_ ? 0U : pow_.size();
    pow_.resize(len);
    pow_[0U] = 1;
    for (auto i = ((cur_len == 0U) ? 1U : cur_len); i <
        len; ++i) {
      pow_[i] = mul(pow_[i - 1U], base_, mod_);
    }
    inv_.resize(len);
    for (auto i = cur_len; i < len; ++i) {
      inv_[i] = inverse(pow_[i], mod_);
    }
    ischanged_ = false;
  }

  template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
      >* = nullptr>
  static constexpr void normalize_kernel(std::size_t len) {
    if (!(ischanged_ || pow_[0U].size() < len)) {
      return;
    }
    auto cur_len = ischanged_ ? 0U : pow_[0U].size();
    auto start_idx = (cur_len == 0U) ? 1U : cur_len;
    for (auto k = 0U; k < Dim; ++k) {
      const auto& m = mod_[k];
      pow_[k].resize(len);
      pow_[k][0U] = 1;
      for (auto i = start_idx; i < len; ++i) {
        pow_[k][i] = mul(pow_[k][i - 1U], base_[k], m)
            ;
      }
      inv_[k].resize(len);
      for (auto i = cur_len; i < len; ++i) {
        inv_[k][i] = inverse(pow_[k][i], m);
      }
    }
    ischanged_ = false;
  }

public:
  Rolling_hash() : ishashed_(false), isbidirect_(false),
      changeid_(-1) {}

  template <class InputIter,
          std::_RequireInputIter<InputIter>* = nullptr>
  Rolling_hash(InputIter first, InputIter last, bool
      bidirectional = false)
      : Rolling_hash() { hash(first, last, bidirectional);
          }

  template <class InputIter,
          std::_RequireInputIter<InputIter>* = nullptr,
          std::size_t KK = Dim, std::enable_if_t<KK == 1U
              >* = nullptr>
  void hash(InputIter first, InputIter last,
          bool bidirectional = false) {
    const std::size_t len = std::distance(first, last);
    assert(len > 0U);
    normalize_kernel(len);

    isbidirect_ = bidirectional;
    changeid_ = nchanges_;

    auto i = 0U, j = 0U;

    pref_.resize(len);
    pref_[0U] = static_cast<Tp>(*first) % mod_;
    i = 1U;
    for (auto it = next(first); it != last; ++it) {
      pref_[i] = add(pref_[i - 1U], mul(static_cast<Tp
          >(*it), pow_[i], mod_), mod_);
      ++i;
    }

    if (!bidirectional) {
      ishashed_ = true;
      return;
    }

    suff_.resize(len);
    const auto &prev_first = prev(first), prev_last =
        prev(last);
    suff_[len - 1U] = static_cast<Tp>(*prev_last) % mod_;
    i = len - 2U, j = 1U;
    for (auto it = prev(prev_last); it != prev_first; --
        it) {
      suff_[i] = add(suff_[i + 1U], mul(static_cast<Tp
          >(*it), pow_[j], mod_), mod_);
      --i, ++j;
    }

    ishashed_ = true;
  }

  template <class InputIter,
          std::_RequireInputIter<InputIter>* = nullptr,
          std::size_t KK = Dim, std::enable_if_t<KK != 1U
              >* = nullptr>
  void hash(InputIter first, InputIter last,
          bool bidirectional = false) {
    const std::size_t len = std::distance(first, last);
    assert(len > 0U);
    normalize_kernel(len);

    isbidirect_ = bidirectional;
    changeid_ = nchanges_;

    auto i = 0U, j = 0U;
    const auto &prev_first = prev(first), prev_last =
        prev(last);

    for (auto k = 0U; k < Dim; ++k) {
      const auto& m = mod_[k];
      pref_[k].resize(len);
      pref_[k][0U] = static_cast<Tp>(*first) % m;
      i = 1U;
      for (auto it = next(first); it != last; ++it) {
        pref_[k][i] = add(pref_[k][i - 1U],
                          mul(static_cast<Tp>(*it), pow_
                              [k][i], m), m);
        ++i;
      }
      if (!bidirectional) {
        continue;
      }
      suff_[k].resize(len);
      suff_[k][len - 1U] = static_cast<Tp>(*prev_last)
          % m;
      i = len - 2U, j = 1U;
      for (auto it = prev(prev_last); it != prev_first;
          --it) {
        suff_[k][i] = add(suff_[k][i + 1U],
                          mul(static_cast<Tp>(*it), pow_
                              [k][j], m), m);
        --i, ++j;
      }
    }

    ishashed_ = true;
  }
```

```
template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
    >* = nullptr>
Tp get(std::size_t pos = 0U, std::size_t len = SIZE_MAX)
    const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (changeid_ != nchanges_) {
        throw std::runtime_error("At least one of `base`
            and `modulus` has \
        changed and hence this instance can no longer be
            sliced");
    }
    if (pos >= pref_.size()) {
        throw std::out_of_range("Starting index is out of
            range");
    }
    if (len == 0U) {
        return 0;
    }

    std::size_t r = std::min(pos + len, pref_.size()) - 1
        U;
    if (pos == 0U) {
        return pref_[r];
    }
    return mul((pref_[r] - pref_[pos - 1U] + mod_) % mod_
        , inv_[pos], mod_);
}

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
    >* = nullptr>
std::array<Tp, Dim> get(std::size_t pos = 0U, std::size_t
    len = SIZE_MAX) const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (changeid_ != nchanges_) {
        throw std::runtime_error("At least one of `base`
            and `modulus` has \
        changed and hence this instance can no longer be
            sliced");
    }
    if (pos >= pref_[0U].size()) {
        throw std::out_of_range("Starting index is out of
            range");
    }
    if (len == 0U) {
        return std::array<Tp, Dim>({});
    }

    std::size_t r = std::min(pos + len, pref_[0U].size())
        - 1U;
    std::array<Tp, Dim> res;
    if (pos == 0U) {
        for (auto k = 0U; k < Dim; ++k) {
            res[k] = pref_[k][r];
        }
        return res;
    }
    for (auto k = 0U; k < Dim; ++k) {
        const auto& m = mod_[k];
        res[k] = mul((pref_[k][r] - pref_[k][pos - 1U] +
            m) % m,
                     inv_[k][pos], m);
    }
    return res;
}

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
    >* = nullptr>
Tp getrev(std::size_t pos = 0U, std::size_t len =
    SIZE_MAX) const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (!isbidirect_) {
        throw std::runtime_error("Not hashed
            bidirectionally");
    }
    if (changeid_ != nchanges_) {
        throw std::runtime_error("At least one of `base`
            and `modulus` has \
        changed and hence this instance can no longer be
            sliced");
    }
    if (pos >= suff_.size()) {
        throw std::out_of_range("Starting index is out of
            range");
    }
    if (len == 0U) {
        return 0;
    }

    std::size_t r = std::min(pos + len, suff_.size()) - 1
        U;
    auto rem = suff_.size() - 1U - r;
    if (rem == 0U) {
        return suff_[pos];
    }
    return mul((suff_[pos] - suff_[r + 1U] + mod_) % mod_
        , inv_[rem], mod_);
}

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
    >* = nullptr>
std::array<Tp, Dim> getrev(std::size_t pos = 0U, std::
    size_t len = SIZE_MAX) const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (!isbidirect_) {
        throw std::runtime_error("Not hashed
            bidirectionally");
    }
    if (changeid_ != nchanges_) {
        throw std::runtime_error("At least one of `base`
            and `modulus` has \
        changed and hence this instance can no longer be
            sliced");
    }
    if (pos >= suff_[0U].size()) {
        throw std::out_of_range("Starting index is out of
            range");
    }
    if (len == 0U) {
        return std::array<Tp, Dim>({});
    }

    std::size_t r = std::min(pos + len, suff_[0U].size())
        - 1U;
    auto rem = suff_[0U].size() - 1U - r;
    std::array<Tp, Dim> res;
    if (rem == 0U) {
        for (auto k = 0U; k < Dim; ++k) {
            res[k] = suff_[k][pos];
        }
        return res;
    }
    for (auto k = 0U; k < Dim; ++k) {
        const auto& m = mod_[k];
        res[k] = mul((suff_[k][pos] - suff_[k][r + 1U] +
            m) % m,
                     inv_[k][rem], m);
    }
    return res;
}

bool is_hashed() const { return ishashed_; }
bool is_bidirectional() const { return isbidirect_; }
```

```
template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
    >* = nullptr>
std::size_t size() const { return pref_.size(); }

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
    >* = nullptr>
std::size_t size() const { return pref_[0U].size(); }

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
    >* = nullptr>
static constexpr Tp base() { return base_; }

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
    >* = nullptr>
static constexpr Tp base(std::size_t i) { return base_[i
    ]; }

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
    >* = nullptr>
static constexpr void base(Tp new_base) {
    if (new_base <= 1) {
        throw std::invalid_argument("`new_base` must be
            greater than 1");
    }
    base_ = new_base;
    ischanged_ = true;
    ++nchanges_;
}

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
    >* = nullptr>
static constexpr void base(std::size_t i, Tp new_base) {
    if (new_base <= 1) {
        throw std::invalid_argument("`new_base` must be
            greater than 1");
    }
    base_[i] = new_base;
    ischanged_ = true;
    ++nchanges_;
}

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
    >* = nullptr>
static constexpr Tp modulus() { return mod_; }

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
    >* = nullptr>
static constexpr Tp modulus(std::size_t i) {
    return mod_[i];
}

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
    >* = nullptr>
static constexpr void modulus(Tp new_modulus) {
    if (new_modulus <= 1) {
        throw std::invalid_argument("`new_modulus` must
            be greater than 1");
    }
    mod_ = new_modulus;
    ischanged_ = true;
    ++nchanges_;
}

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
    >* = nullptr>
static constexpr void modulus(std::size_t i, Tp
    new_modulus) {
    if (new_modulus <= 1) {
        throw std::invalid_argument("`new_modulus` must
            be greater than 1");
    }
    mod_[i] = new_modulus;
    ischanged_ = true;
    ++nchanges_;
}
};

template <std::size_t Dim, class Tp, class Enabler>
int Rolling_hash<Dim, Tp, Enabler>::nchanges_ = 0;
template <std::size_t Dim, class Tp, class Enabler>
bool Rolling_hash<Dim, Tp, Enabler>::ischanged_ = false;

#ifndef ROLLING_HASH_PARTIAL_SPEC_INIT
#define ROLLING_HASH_PARTIAL_SPEC_INIT(Tp)
                                    \
    template <>

        \
    std::array<Tp, 3U> Rolling_hash<3U, Tp>::base_ = {257,
        263, 269};  \
    template <>

        \
    std::array<Tp, 3U> Rolling_hash<3U, Tp>::mod_ =
        {1000000007, 1000000009, \
                                                1000000021};

                                                    \
    template <>

        \
    std::array<std::vector<Tp>, 3U> Rolling_hash<3U, Tp>::
        pow_ = {};   \
    template <>

        \
    std::array<std::vector<Tp>, 3U> Rolling_hash<3U, Tp>::
        inv_ = {};   \


    template <>

        \
    std::array<Tp, 2U> Rolling_hash<2U, Tp>::base_ = {257,
        263};       \
    template <>

        \
    std::array<Tp, 2U> Rolling_hash<2U, Tp>::mod_ =
        {1000000007, 1000000009}; \
    template <>

        \
    std::array<std::vector<Tp>, 2U> Rolling_hash<2U, Tp>::
        pow_ = {};   \
    template <>

        \
    std::array<std::vector<Tp>, 2U> Rolling_hash<2U, Tp>::
        inv_ = {};   \


    template <>

        \
    Tp Rolling_hash<1U, Tp>::base_ = 257;
                                        \
    template <>

        \
    Tp Rolling_hash<1U, Tp>::mod_ = 1000000007;
                                    \
    template <>

        \
    std::vector<Tp> Rolling_hash<1U, Tp>::pow_ = {};
                            \
```

```
    template <>

        \
    std::vector<Tp> Rolling_hash<1U, Tp>::inv_ = {};

ROLLING_HASH_PARTIAL_SPEC_INIT(int)
ROLLING_HASH_PARTIAL_SPEC_INIT(std::int64_t)
#endif

template <class Tp>
using single_hash = Rolling_hash<1U, Tp>;
template <class Tp>
using double_hash = Rolling_hash<2U, Tp>;
template <class Tp>
using triple_hash = Rolling_hash<3U, Tp>;
```