Contents

1 Data structures **2**
1.1 Segtree Lazy (Atcoder) . . . 2
1.2 Bitree 2D . . . 3
1.3 Bitree . . . 4
1.4 Convex Hull Trick / Line Container . . . 4
1.5 Disjoint Sparse Table . . . 5
1.6 Dsu . . . 5
1.7 Lichao Tree (dynamic) . . . 6
1.8 Merge Sort Tree . . . 6
1.9 Ordered Set . . . 7
1.10 Prefix Sum 2D . . . 7
1.11 SegTree Range Sum Query Range PA sum/set Update . . . 7
1.12 SegTree Point Update (dynamic function) . . . 9
1.13 Segtree Range Max Query Point Max Assign Update (dynamic) . . . 9
1.14 Segtree Range Max Query Range Max Update . . . 10
1.15 SegTree Range Min Query Point Assign Update . . . 11
1.16 Segtree Range Sum Query Point Sum Update (dynamic) . . . 12
1.17 SegTree Range Xor query Point Assign Update . . . 12
1.18 SegTree Range Min Query Range Sum Update . . . 13
1.19 SegTree Range Sum Query Range Sum Update . . . 14
1.20 Sparse Table . . . 15

2 Dynamic programming **15**
2.1 Binary Knapsack (bottom up) . . . 15
2.2 Binary Knapsack (top down) . . . 16
2.3 Edit Distance . . . 16
2.4 Kadane . . . 16
2.5 Knapsack with quantity (no recover) . . . 17
2.6 Longest Increasing Subsequence (LIS) . . . 17
2.7 Money Sum (Bottom Up) . . . 17
2.8 Travelling Salesman Problem . . . 18

3 Geometry **18**
3.1 Convex Hull . . . 18
3.2 Determinant . . . 18
3.3 Equals . . . 18
3.4 Line . . . 19
3.5 Point Struct And Utils (2d) . . . 19
3.6 Polygon Lattice Points (Pick's Theorem) . . . 19
3.7 Segment . . . 20
3.8 Template Line . . . 20
3.9 Template Point . . . 21
3.10 Template Segment . . . 21

4 Graphs **22**
4.1 2 SAT . . . 22
4.2 Cycle Distances . . . 23
4.3 SCC (struct) . . . 23
4.4 Bellman-Ford (find negative cycle) . . . 23
4.5 Bellman Ford . . . 24
4.6 BFS 01 . . . 24
4.7 Biconnected Components . . . 25
4.8 Binary Lifting/Jumping . . . 25
4.9 Block Cut Tree . . . 25
4.10 Check Bipartitie . . . 26
4.11 Dijkstra (k Shortest Paths) . . . 26
4.12 Dijkstra . . . 27
4.13 Disjoint Edges Path (Maxflow) . . . 27
4.14 Euler Path (directed) . . . 28
4.15 Euler Path (undirected) . . . 29
4.16 Extra Edges to Make Digraph Fully Strongly Connected . . . 30
4.17 Find Articulation/Cut Points . . . 31
4.18 Find Bridge Tree Components . . . 31
4.19 Find Bridges (online) . . . 32
4.20 Find Bridges . . . 33
4.21 Find Centroid . . . 33
4.22 Floyd Warshall . . . 34
4.23 Functional/Successor Graph . . . 34
4.24 Graph Cycle (directed) . . . 34
4.25 Graph Cycle (undirected) . . . 35
4.26 Heavy Light Decomposition . . . 35
4.27 Kruskal . . . 36
4.28 Lowest Common Ancestor (Binary Lifting) . . . 37
4.29 Lowest Common Ancestor . . . 38
4.30 Maximum Flow (Dinic) . . . 38
4.31 Maximum Flow (Edmonds-Karp) . . . 39
4.32 Minimum Cost Flow . . . 40
4.33 Minimum Cut (unweighted) . . . 42
4.34 Prim (MST) . . . 43
4.35 Small to Large . . . 43
4.36 Successor Graph-(struct) . . . 44
4.37 Sum every node distance . . . 45
4.38 Topological Labelling (Kahn) . . . 46
4.39 Topological Sorting (Kahn) . . . 46
4.40 Topological Sorting (Tarjan) . . . 46
4.41 Tree Diameter (DP) . . . 47
4.42 Tree Isomorphism (not rooted) . . . 47
4.43 Tree Isomorphism (rooted) . . . 48
4.44 Tree Maximum Distance . . . 48

4.45 Tree Flatten . . . 49

5 Math **49**
5.1 GCD (with factorization) . . . 49
5.2 GCD . . . 49
5.3 LCM (with factorization) . . . 49
5.4 LCM . . . 50
5.5 Arithmetic Progression Sum . . . 50
5.6 Binomial MOD . . . 50
5.7 Binomial . . . 50
5.8 Chinese Remainder Theorem . . . 50
5.9 Derangement / Matching Problem . . . 51
5.10 Euler phi $\varphi(n)$ (in range) . . . 51
5.11 Euler phi $\varphi(n)$ . . . 51
5.12 Factorial Factorization . . . 52
5.13 Factorial . . . 52
5.14 Factorization (Pollard Rho) . . . 52
5.15 Factorization . . . 53
5.16 Fast pow . . . 53
5.17 FFT Convolution . . . 53
5.18 Find Multiplicative Inverse . . . 54
5.19 Linear Diophantine Equation: Find any solution . . . 54
5.20 Gauss Elimination . . . 55
5.21 Integer Partition . . . 56
5.22 Integer Mod . . . 56
5.23 Matrix Exponentiation . . . 57
5.24 N Choose K (elements) . . . 57
5.25 NTT integer convolution and exponentiation . . . 57
5.26 NTT Integer Convolution (combine 2 modules) . . . 59
5.27 Number Of Divisors (sieve) . . . 61
5.28 Number of Divisors $\tau(n)$ . . . 62
5.29 Power Sum . . . 62
5.30 Sieve list primes . . . 62
5.31 Sum of Divisors $\sigma(n)$ . . . 62
5.32 To Any Base . . . 62

6 Primitives **62**
6.1 Bigint . . . 62
6.2 Integer Mod . . . 67
6.3 Matrix . . . 68

7 Problems **70**
7.1 Hanoi Tower . . . 70

8 Searching **70**
8.1 Meet in the middle . . . 70
8.2 Ternary Search Recursive . . . 71

9 Strings **71**
9.1 Count Distinct Anagrams . . . 71
9.2 Double Hash Range Query . . . 71
9.3 Hash Interal mod $2^{64} - 1$ . . . 72
9.4 Hash Range Query . . . 73
9.5 Hash Ull . . . 73
9.6 K-th digit in digit string . . . 73
9.7 Longest Palindrome Substring (Manacher) . . . 74
9.8 Longest Palindrome . . . 74
9.9 Rabin Karp . . . 75
9.10 String Psum . . . 75
9.11 Suffix Automaton (complete) . . . 75
9.12 Trie . . . 77
9.13 Z-function get occurence positions . . . 78

10 Settings and macros **78**
10.1 gen.cpp . . . 78
10.2 macro.cpp . . . 79
10.3 short-macro.cpp . . . 80
10.4 debug.cpp . . . 80
10.5 .bashrc . . . 81
10.6 .vimrc . . . 82

1 Data structures

1.1 Segtree Lazy (Atcoder)

```
struct Node {
  // need an empty constructor with the neutral node
  Node() : {}
};

struct Lazy {
  // need an empty constructor with the neutral lazy
  Lazy() : {}
};

// how to merge two nodes
Node op(Node a, Node b) {}

// how to apply the lazy into a node
Node mapping(Lazy a, Node b, int, int) {}

// how to merge two lazy
Lazy comp(Lazy a, Lazy b) {}

template <typename T, auto op, typename L, auto mapping,
          auto composition>
struct SegTreeLazy {
  static_assert(
    is_convertible_v<decltype(op), function<T(T, T)>>,
    "op must be a function T(T, T)");
  static_assert(
    is_convertible_v<decltype(mapping),
                     function<T(L, T, int, int)>>,
    "mapping must be a function T(L, T, int, int)");
  static_assert(is_convertible_v<decltype(composition),
                                 function<L(L, L)>>,
                "composition must be a function L(L, L)");

  int N, size, height;
  const T eT;
  const L eL;
  vector<T> d;
  vector<L> lz;

  SegTreeLazy(const T &eT_ = T(), const L &eL_ = L())
    : SegTreeLazy(0, eT_, eL_) {}
  explicit SegTreeLazy(int n, const T &eT_ = T(),
                       const L &eL_ = L())
    : SegTreeLazy(vector<T>(n, eT_), eT_, eL_) {}
  explicit SegTreeLazy(const vector<T> &v,
                       const T &eT_ = T(),
                       const L &eL_ = L())
    : N(int(v.size())), eT(eT_), eL(eL_) {
    size = 1;
    height = 0;
    while (size < N) size <<= 1, height++;
    d = vector<T>(2 * size, eT);
    lz = vector<L>(size, eL);
    for (int i = 0; i < N; i++) d[size + i] = v[i];
    for (int i = size - 1; i >= 1; i--) {
      update(i);
    }
  }

  void set(int p, T x) {
    assert(0 <= p && p < N);
    p += size;
    for (int i = height; i >= 1; i--) push(p >> i);
    d[p] = x;
    for (int i = 1; i <= height; i++) update(p >> i);
  }

  T get(int p) {
    assert(0 <= p && p < N);
    p += size;
    for (int i = height; i >= 1; i--) push(p >> i);
    return d[p];
  }

  T query(int l, int r) {
    assert(0 <= l && l <= r && r < N);

    l += size;
    r += size;

    for (int i = height; i >= 1; i--) {
      if (((l >> i) << i) != l) push(l >> i);
      if ((((r + 1) >> i) << i) != (r + 1)) push(r >> i);
    }
```

```
  T sml = eT, smr = eT;
  while (l <= r) {
    if (l & 1) sml = op(sml, d[l++]);
    if (!(r & 1)) smr = op(d[r--], smr);
    l >>= 1;
    r >>= 1;
  }

  return op(sml, smr);
}

T query_all() { return d[1]; }

void update(int p, L f) {
  assert(0 <= p && p < N);
  p += size;
  for (int i = height; i >= 1; i--) push(p >> i);
  d[p] = mapping(f, d[p]);
  for (int i = 1; i <= height; i++) update(p >> i);
}
void update(int l, int r, L f) {
  assert(0 <= l && l <= r && r < N);

  l += size;
  r += size;

  for (int i = height; i >= 1; i--) {
    if (((l >> i) << i) != l) push(l >> i);
    if ((((r + 1) >> i) << i) != (r + 1)) push(r >> i);
  }

  {
    int l2 = l, r2 = r;
    while (l <= r) {
      if (l & 1) all_apply(l++, f);
      if (!(r & 1)) all_apply(r--, f);
      l >>= 1;
      r >>= 1;
    }
    l = l2;
    r = r2;
  }

  for (int i = 1; i <= height; i++) {
    if (((l >> i) << i) != l) update(l >> i);
      if ((((r + 1) >> i) << i) != (r + 1)) update(r >> i);
    }
  }
  pair<int, int> node_range(int k) const {
    int remain = height;
    for (int kk = k; kk >>= 1; --remain)
      ;
    int fst = k << remain;
    int lst = min(fst + (1 << remain) - 1, size + N - 1);
    return {fst - size, lst - size};
  }

 private:
  void update(int k) { d[k] = op(d[2 * k], d[2 * k + 1]); }
  void all_apply(int k, L f) {
    auto [fst, lst] = node_range(k);
    d[k] = mapping(f, d[k], fst, lst);
    if (k < size) lz[k] = composition(f, lz[k]);
  }
  void push(int k) {
    all_apply(2 * k, lz[k]);
    all_apply(2 * k + 1, lz[k]);
    lz[k] = eL;
  }
};
```

1.2 Bitree 2D

Given a 2d array allow you to sum val to the position (x, y) and find the sum of the rectangle with left top corner $(x1, y1)$ and right bottom corner $(x2, y2)$

Update and query 1 indexed !

Time: update $O(logn^2)$, query $O(logn^2)$

```
struct Bit2d {
  int n;
  vll2d bit;
  Bit2d(int ni) : n(ni), bit(n + 1, vll(n + 1)) {}
  Bit2d(int ni, vll2d &xs) : n(ni), bit(n + 1, vll(n + 1)) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        update(i, j, xs[i][j]);
      }
    }
  }
  void update(int x, int y, ll val) {
    for (; x <= n; x += (x & (-x))) {
```

```
      for (int i = y; i <= n; i += (i & (-i))) {
        bit[x][i] += val;
      }
    }
  }
  ll sum(int x, int y) {
    ll ans = 0;

    for (int i = x; i; i -= (i & (-i))) {
      for (int j = y; j; j -= (j & (-j))) {
        ans += bit[i][j];
      }
    }
    return ans;
  }
  ll query(int x1, int y1, int x2, int y2) {
    return sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) +
           sum(x1 - 1, y1 - 1);
  }
};
```

1.3 Bitree

```
template <typename T>
struct BITree {
  int N;
  vector<T> v;

  BITree(int n) : N(n), v(n + 1, 0) {}

  void update(int i, const T& x) {
    if (i == 0) return;
    for (; i <= N; i += i & -i) v[i] += x;
  }

  T range_sum(int i, int j) {
    return range_sum(j) - range_sum(i - 1);
  }

  T range_sum(int i) {
    T sum = 0;
    for (; i > 0; i -= i & -i) sum += v[i];
    return sum;
  }
};
```

1.4 Convex Hull Trick / Line Container

Container where you can add lines of the form $mx + b$, and query maximum value at point x.
$insert\_line(m, b)$ inserts the line $m \cdot x + b$ in the container.
$eval(x)$ find the highest value among all lines in the point x.
both in $O(\log N)$

```
const ll LLINF = 1e18;
const ll is_query = -LLINF;
struct Line {
  ll m, b;
  mutable function<const Line*()> succ;
  bool operator<(const Line& rhs) const {
    if (rhs.b != is_query) return m < rhs.m;
    const Line* s = succ();
    if (!s) return 0;
    ll x = rhs.m;
    return b - s->b < (s->m - m) * x;
  }
};
struct Cht : public multiset<Line> {  // maintain max m*x+b
  bool bad(iterator y) {
    auto z = next(y);
    if (y == begin()) {
      if (z == end()) return 0;
      return y->m == z->m && y->b <= z->b;
    }
    auto x = prev(y);
    if (z == end()) return y->m == x->m && y->b <= x->b;
    return (ld)(x->b - y->b) * (z->m - y->m) >=
           (ld)(y->b - z->b) * (y->m - x->m);
  }
  void insert_line(
    ll m, ll b) {  // min -> insert (-m,-b) -> -eval()
    auto y = insert({m, b});
    y->succ = [=] {
      return next(y) == end() ? 0 : &*next(y);
    };
    if (bad(y)) {
      erase(y);
      return;
    }
    while (next(y) != end() && bad(next(y))) erase(next(y));
    while (y != begin() && bad(prev(y))) erase(prev(y));
  }
  ll eval(ll x) {
    auto l = *lower_bound((Line){x, is_query});
```

```
    return l.m * x + l.b;
  }
};
```

1.5 Disjoint Sparse Table

Answers queries of any monoid operation (i.e. has identity element and is associative)
Build: $O(N \log N)$, Query: $O(1)$

```
#define F(expr) [](auto a, auto b) { return expr; }
template <typename T>
struct DisjointSparseTable {
  using Operation = T (*)(T, T);

  vector<vector<T>> st;
  Operation f;
  T identity;

  static constexpr int log2_floor(
    unsigned long long i) noexcept {
    return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
  }

  // Lazy loading constructor. Needs to call build!
  DisjointSparseTable(Operation op, const T neutral = T())
    : st(), f(op), identity(neutral) {}

  DisjointSparseTable(vector<T> v)
    : DisjointSparseTable(v, F(min(a, b))) {}

  DisjointSparseTable(vector<T> v, Operation op,
                      const T neutral = T())
    : st(), f(op), identity(neutral) {
    build(v);
  }

  void build(vector<T> v) {
    st.resize(log2_floor(v.size()) + 1,
              vector<T>(1ll << (log2_floor(v.size()) + 1)));
    v.resize(st[0].size(), identity);
    for (int level = 0; level < (int)st.size(); ++level) {
      for (int block = 0; block < (1 << level); ++block) {
        const auto l = block << (st.size() - level);
        const auto r = (block + 1) << (st.size() - level);
        const auto m = l + (r - l) / 2;
        st[level][m] = v[m];
        for (int i = m + 1; i < r; i++)
          st[level][i] = f(st[level][i - 1], v[i]);
        st[level][m - 1] = v[m - 1];
        for (int i = m - 2; i >= l; i--)
          st[level][i] = f(st[level][i + 1], v[i]);
      }
    }
  }

  T query(int l, int r) const {
    if (l > r) return identity;
    if (l == r) return st.back()[l];

    const auto k = log2_floor(l ^ r);
    const auto level = (int)st.size() - 1 - k;
    return f(st[level][l], st[level][r]);
  }
};
```

1.6 Dsu

```
struct DSU {
  vi ps, sz;

  // vector<unordered_set<int>> sts;

  DSU(int N) : ps(N + 1), sz(N, 1) /*, sts(N) */ {
    iota(all(ps), 0);
    // for (int i = 0; i < N; i++) sts[i].insert(i);
  }
  int find_set(int x) {
    return ps[x] == x ? x : ps[x] = find_set(ps[x]);
  }
  int size(int u) { return sz[find_set(u)]; }
  bool same_set(int x, int y) {
    return find_set(x) == find_set(y);
  }
  void union_set(int x, int y) {
    if (same_set(x, y)) return;

    int px = find_set(x);
    int py = find_set(y);

    if (sz[px] < sz[py]) swap(px, py);
```

```
    ps[py] = px;
    sz[px] += sz[py];
    // sts[px].merge(sts[py]);
  }
};
```

1.7 Lichao Tree (dynamic)

Lichao Tree that creates the nodes dynamically, allowing to query and update from range $[MAXL, MAXR]$
$query(x)$: find the highest point among all lines in the structure
$add(a, b)$: add a line of form $y = ax + b$ in the structure
$addSegment(a, b, l, r)$: add a line segment of form $y = ax + b$ which covers from range $[l, r]$
time: $O(\log N)$

```
template <typename T = ll, T MAXL = 0,
          T MAXR = 1'000'000'001>
struct LiChaoTree {
  static const T inf = -numeric_limits<T>::max() / 2;
  bool first_best(T a, T b) { return a > b; }
  T get_best(T a, T b) { return first_best(a, b) ? a : b; }
  struct line {
    T m, b;
    T operator()(T x) { return m * x + b; }
  };
  struct node {
    line li;
    node *left, *right;
    node(line _li = {0, inf})
      : li(_li), left(nullptr), right(nullptr) {}
    ~node() {
      delete left;
      delete right;
    }
  };
  node *root;
  LiChaoTree(line li = {0, inf}) : root(new node(li)) {}
  ~LiChaoTree() { delete root; }
  T query(T x, node *cur, T l, T r) {
    if (cur == nullptr) return inf;
    if (x < l or x > r) return inf;
    T mid = midpoint(l, r);
    T ans = cur->li(x);
    ans = get_best(ans, query(x, cur->left, l, mid));
    ans = get_best(ans, query(x, cur->right, mid + 1, r));
    return ans;
  }
  T query(T x) { return query(x, root, MAXL, MAXR); }
  void add(line li, node *&cur, T l, T r) {
    if (cur == nullptr) {
      cur = new node(li);
      return;
    }
    T mid = midpoint(l, r);
    if (first_best(li(mid), cur->li(mid)))
      swap(li, cur->li);
    if (first_best(li(l), cur->li(l)))
      add(li, cur->left, l, mid);
    if (first_best(li(r), cur->li(r)))
      add(li, cur->right, mid + 1, r);
  }
  void add(T m, T b) { add({m, b}, root, MAXL, MAXR); }
  void addSegment(line li, node *&cur, T l, T r, T lseg,
                  T rseg) {
    if (r < lseg || l > rseg) return;
    if (cur == nullptr) cur = new node;
    if (lseg <= l && r <= rseg) {
      add(li, cur, l, r);
      return;
    }
    T mid = midpoint(l, r);
    if (l != r) {
      addSegment(li, cur->left, l, mid, lseg, rseg);
      addSegment(li, cur->right, mid + 1, r, lseg, rseg);
    }
  }
  void addSegment(T a, T b, T l, T r) {
    addSegment({a, b}, root, MAXL, MAXR, l, r);
  }
};
```

1.8 Merge Sort Tree

Like a segment tree but each node st_i stores a sorted subarray

- $inrange(l, r, a, b)$: counts the number of elements $x \in [l, r]$ such that $a \leq x \leq b$.

Memory: $O(n \log N)$ Time: build $O(N \log N)$, inrange $O(\log N)$

```
template <class T>
struct MergeSortTree {
  int n;
  vector<vector<T>> st;
  MergeSortTree(vector<T> &xs) : n(len(xs)), st(n << 1) {
```

```
    for (int i = 0; i < n; i++)
      st[i + n] = vector<T>({xs[i]});

    for (int i = n - 1; i > 0; i--) {
      st[i].resize(len(st[i << 1]) + len(st[i << 1 | 1]));
      merge(all(st[i << 1]), all(st[i << 1 | 1]),
            st[i].begin());
    }
  }
  int count(int i, T a, T b) {
    return upper_bound(all(st[i]), b) -
           lower_bound(all(st[i]), a);
  }
  int inrange(int l, int r, T a, T b) {
    int ans = 0;

    for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
      if (l & 1) ans += count(l++, a, b);
      if (r & 1) ans += count(--r, a, b);
    }

    return ans;
  }
};
```

1.9 Ordered Set

If you need an ordered **multi**set you may add an id to each value. Using greater_equal, or less_equal is considered undefined behavior.

- **order_of_key (k)** : Number of items strictly smaller/greater than k .
- **find_by_order(k)** : K-th element in a set (counting from zero).

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

template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
                         tree_order_statistics_node_update>;
```

1.10 Prefix Sum 2D

Given an 2d array with n lines and m columns, find the sum of the subarray that have the left upper corner at $(x1, y1)$ and right bottom corner at $(x2, y2)$.
Time: build $O(n \cdot m)$, query $O(1)$.

```
struct psum2d {
  vll2d s;
  vll2d psum;
  psum2d(vll2d &grid, int n, int m)
    : s(n + 1, vll(m + 1)), psum(n + 1, vll(m + 1)) {
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= m; j++)
        s[i][j] = s[i][j - 1] + grid[i][j];

    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= m; j++)
        psum[i][j] = psum[i - 1][j] + s[i][j];
  }

  ll query(int x1, int y1, int x2, int y2) {
    ll ans = psum[x2][y2] + psum[x1 - 1][y1 - 1];
    ans -= psum[x2][y1 - 1] + psum[x1 - 1][y2];
    return ans;
  }
};
```

1.11 SegTree Range Sum Query Range PA sum/set Update

Makes arithmetic progression updates in range and sum queries.
Considering $PA(A, R) = [A + R, A + 2R, A + 3R, ...]$

- **update_set(l, r, A, R):** sets [l, r] to PA(A, R)
- **update_add(l, r, A, R):** sum PA(A, R) in [l, r]
- **query(l, r):** sum in range [l, r]

0 indexed !
Time: build $O(n)$, updates and queries $O(logn)$

```
const ll oo = 1e18;
struct SegTree {
  struct Data {
    ll sum;
    ll set_a, set_r, add_a, add_r;
    Data()
      : sum(0), set_a(oo), set_r(0), add_a(0), add_r(0) {}
  };
  int n;
  vector<Data> seg;
  SegTree(int n_) : n(n_), seg(vector<Data>(4 * n)) {}

  void prop(int p, int l, int r) {
    int sz = r - l + 1;
```

```
  ll &sum = seg[p].sum, &set_a = seg[p].set_a,
     &set_r = seg[p].set_r, &add_a = seg[p].add_a,
     &add_r = seg[p].add_r;

  if (set_a != oo) {
    set_a += add_a, set_r += add_r;
    sum = set_a * sz + set_r * sz * (sz + 1) / 2;
    if (l != r) {
      int m = (l + r) / 2;

      seg[2 * p].set_a = set_a;
      seg[2 * p].set_r = set_r;
      seg[2 * p].add_a = seg[2 * p].add_r = 0;

      seg[2 * p + 1].set_a = set_a + set_r * (m - l + 1);
      seg[2 * p + 1].set_r = set_r;
      seg[2 * p + 1].add_a = seg[2 * p + 1].add_r = 0;
    }
    set_a = oo, set_r = 0;
    add_a = add_r = 0;
  } else if (add_a or add_r) {
    sum += add_a * sz + add_r * sz * (sz + 1) / 2;
    if (l != r) {
      int m = (l + r) / 2;

      seg[2 * p].add_a += add_a;
      seg[2 * p].add_r += add_r;

      seg[2 * p + 1].add_a += add_a + add_r * (m - l + 1);
      seg[2 * p + 1].add_r += add_r;
    }
    add_a = add_r = 0;
  }
}

int inter(pii a, pii b) {
  if (a.first > b.first) swap(a, b);
  return max(0, min(a.second, b.second) - b.first + 1);
}
ll set(int a, int b, ll aa, ll rr, int p, int l, int r) {
  prop(p, l, r);
  if (b < l or r < a) return seg[p].sum;
  if (a <= l and r <= b) {
    seg[p].set_a = aa;
    seg[p].set_r = rr;
```

```
      prop(p, l, r);
      return seg[p].sum;
    }
    int m = (l + r) / 2;
    int tam_l = inter({l, m}, {a, b});
    return seg[p].sum = set(a, b, aa, rr, 2 * p, l, m) +
                        set(a, b, aa + rr * tam_l, rr,
                            2 * p + 1, m + 1, r);
  }
  void update_set(int l, int r, ll aa, ll rr) {
    set(l, r, aa, rr, 1, 0, n - 1);
  }
  ll add(int a, int b, ll aa, ll rr, int p, int l, int r) {
    prop(p, l, r);
    if (b < l or r < a) return seg[p].sum;
    if (a <= l and r <= b) {
      seg[p].add_a += aa;
      seg[p].add_r += rr;
      prop(p, l, r);
      return seg[p].sum;
    }
    int m = (l + r) / 2;
    int tam_l = inter({l, m}, {a, b});
    return seg[p].sum = add(a, b, aa, rr, 2 * p, l, m) +
                        add(a, b, aa + rr * tam_l, rr,
                            2 * p + 1, m + 1, r);
  }
  void update_add(int l, int r, ll aa, ll rr) {
    add(l, r, aa, rr, 1, 0, n - 1);
  }
  ll query(int a, int b, int p, int l, int r) {
    prop(p, l, r);
    if (b < l or r < a) return 0;
    if (a <= l and r <= b) return seg[p].sum;
    int m = (l + r) / 2;
    return query(a, b, 2 * p, l, m) +
           query(a, b, 2 * p + 1, m + 1, r);
  }
  ll query(int l, int r) {
    return query(l, r, 1, 0, n - 1);
  }
};
```

1.12 SegTree Point Update (dynamic function)

Answers queries of any monoid operation (i.e. has identity element and is associative)
Build: $O(N)$, Query: $O(\log N)$

```
#define F(expr) [](auto a, auto b) { return expr; }
template <typename T>
struct SegTree {
  using Operation = T (*)(T, T);

  int N;
  vector<T> ns;
  Operation operation;
  T identity;

  SegTree(int n, Operation op = F(a + b), T neutral = T())
    : N(n),
      ns(2 * N, neutral),
      operation(op),
      identity(neutral) {}

  SegTree(const vector<T> &v, Operation op = F(a + b),
          T neutral = T())
    : SegTree((int)v.size(), op, neutral) {
    copy(v.begin(), v.end(), ns.begin() + N);

    for (int i = N - 1; i > 0; --i)
      ns[i] = operation(ns[2 * i], ns[2 * i + 1]);
  }

  T query(size_t i) const { return ns[i + N]; }

  T query(size_t l, size_t r) const {
    auto a = l + N, b = r + N;
    auto ans = identity;
    // Non-associative operations needs to be processed
    // backwards
    stack<T> st;
    while (a <= b) {
      if (a & 1) ans = operation(ans, ns[a++]);
      if (not(b & 1)) st.push(ns[b--]);

      a >>= 1;
      b >>= 1;
    }
    for (; !st.empty(); st.pop())
      ans = operation(ans, st.top());

    return ans;
  }

  void update(size_t i, T value) {
    update_set(i, operation(ns[i + N], value));
  }

  void update_set(size_t i, T value) {
    auto a = i + N;

    ns[a] = value;
    while (a >>= 1)
      ns[a] = operation(ns[2 * a], ns[2 * a + 1]);
  }
};
```

1.13 Segtree Range Max Query Point Max Assign Update (dynamic)

Answers range queries in ranges until 10^9 (maybe more)
Time: query and update $O(n \cdot \log n)$

```
struct node;
node *newNode();

struct node {
  node *left, *right;
  int lv, rv;
  ll val;

  node() : left(NULL), right(NULL), val(-oo) {}

  inline void init(int l, int r) {
    lv = l;
    rv = r;
  }

  inline void extend() {
    if (!left) {
      int m = (lv + rv) / 2;
      left = newNode();
      right = newNode();
      left->init(lv, m);
```

```
      right->init(m + 1, rv);
    }
  }

  ll query(int l, int r) {
    if (r < lv || rv < l) {
      return 0;
    }

    if (l <= lv && rv <= r) {
      return val;
    }

    extend();
    return max(left->query(l, r), right->query(l, r));
  }

  void update(int p, ll newVal) {
    if (lv == rv) {
      val = max(val, newVal);
      return;
    }

    extend();
    (p <= left->rv ? left : right)->update(p, newVal);
    val = max(left->val, right->val);
  }
};

const int BUFFSZ(1e7);
node *newNode() {
  static int bufSize = BUFFSZ;
  static node buf[(int)BUFFSZ];
  assert(bufSize);
  return &buf[--bufSize];
}

struct SegTree {
  int n;
  node *root;
  SegTree(int _n) : n(_n) {
    root = newNode();
    root->init(0, n);
  }
  ll query(int l, int r) { return root->query(l, r); }
  void update(int p, ll v) { root->update(p, v); }
};
```

1.14 Segtree Range Max Query Range Max Update

```
template <typename T = ll>
struct SegTree {
  int N;
  T nu, nq;
  vector<T> st, lazy;
  SegTree(const vector<T> &xs)
    : N(len(xs)),
      nu(numeric_limits<T>::min()),
      nq(numeric_limits<T>::min()),
      st(4 * N + 1, nu),
      lazy(4 * N + 1, nu) {
    for (int i = 0; i < len(xs); ++i) update(i, i, xs[i]);
  }

  void update(int l, int r, T value) {
    update(1, 0, N - 1, l, r, value);
  }

  T query(int l, int r) { return query(1, 0, N - 1, l, r); }

  void update(int node, int nl, int nr, int ql, int qr,
              T v) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return;

    st[node] = max(st[node], v);
    if (ql <= nl and nr <= qr) {
      if (nl < nr) {
        lazy[left(node)] = max(lazy[left(node)], v);
        lazy[right(node)] = max(lazy[right(node)], v);
      }
      return;
    }
    update(left(node), nl, mid(nl, nr), ql, qr, v);
    update(right(node), mid(nl, nr) + 1, nr, ql, qr, v);

    st[node] = max(st[left(node)], st[right(node)]);
  }
```

```
  T query(int node, int nl, int nr, int ql, int qr) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return nq;

    if (ql <= nl and nr <= qr) return st[node];

    T x = query(left(node), nl, mid(nl, nr), ql, qr);
    T y = query(right(node), mid(nl, nr) + 1, nr, ql, qr);

    return max(x, y);
  }

  void propagation(int node, int nl, int nr) {
    if (lazy[node] != nu) {
      st[node] = max(st[node], lazy[node]);

      if (nl < nr) {
        lazy[left(node)] =
          max(lazy[left(node)], lazy[node]);
        lazy[right(node)] =
          max(lazy[right(node)], lazy[node]);
      }

      lazy[node] = nu;
    }
  }

  int left(int p) { return p << 1; }
  int right(int p) { return (p << 1) + 1; }
  int mid(int l, int r) { return (r - l) / 2 + l; }
};
int main() {
  int n;
  cin >> n;
  vector<array<int, 3>> xs(n);
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < 3; ++j) {
      cin >> xs[i][j];
    }
  }
  vi aux(n, 0);
  SegTree<int> st(aux);
  for (int i = 0; i < n; ++i) {
    int a = min(i + xs[i][1], n);
    int b = min(i + xs[i][2], n);
    st.update(i, i, st.query(i, i) + xs[i][0]);
    int cur = st.query(i, i);
    st.update(a, b, cur);
  }

  cout << st.query(0, n) << '\n';
}
```

1.15 SegTree Range Min Query Point Assign Update

```
template <typename T = ll>
struct SegTree {
  int n;
  T nu, nq;
  vector<T> st;
  SegTree(const vector<T> &v)
    : n(len(v)),
      nu(0),
      nq(numeric_limits<T>::max()),
      st(n * 4 + 1, nu) {
    for (int i = 0; i < n; ++i) update(i, v[i]);
  }
  void update(int p, T v) { update(1, 0, n - 1, p, v); }
  T query(int l, int r) { return query(1, 0, n - 1, l, r); }

  void update(int node, int nl, int nr, int p, T v) {
    if (p < nl or p > nr) return;

    if (nl == nr) {
      st[node] = v;
      return;
    }

    update(left(node), nl, mid(nl, nr), p, v);
    update(right(node), mid(nl, nr) + 1, nr, p, v);

    st[node] = min(st[left(node)], st[right(node)]);
  }

  T query(int node, int nl, int nr, int ql, int qr) {
    if (ql <= nl and qr >= nr) return st[node];
    if (nl > qr or nr < ql) return nq;
    if (nl == nr) return st[node];
```

```
    return min(
      query(left(node), nl, mid(nl, nr), ql, qr),
      query(right(node), mid(nl, nr) + 1, nr, ql, qr));
  }

  int left(int p) { return p << 1; }
  int right(int p) { return (p << 1) + 1; }
  int mid(int l, int r) { return (r - l) / 2 + l; }
};
```

1.16 Segtree Range Sum Query Point Sum Update (dynamic)

Answers range queries in ranges until 10^9 (maybe more)
Time: query and update $O(n \cdot \log n)$

```
struct node;
node *newNode();

struct node {
  node *left, *right;
  int lv, rv;
  ll val;

  node() : left(NULL), right(NULL), val(0) {}

  inline void init(int l, int r) {
    lv = l;
    rv = r;
  }

  inline void extend() {
    if (!left) {
      int m = (rv - lv) / 2 + lv;
      left = newNode();
      right = newNode();
      left->init(lv, m);
      right->init(m + 1, rv);
    }
  }

  ll query(int l, int r) {
    if (r < lv || rv < l) {
      return 0;
    }

    if (l <= lv && rv <= r) {
      return val;
    }

    extend();
    return left->query(l, r) + right->query(l, r);
  }

  void update(int p, ll newVal) {
    if (lv == rv) {
      val += newVal;
      return;
    }

    extend();
    (p <= left->rv ? left : right)->update(p, newVal);
    val = left->val + right->val;
  }
};

const int BUFFSZ(1.3e7);
node *newNode() {
  static int bufSize = BUFFSZ;
  static node buf[(int)BUFFSZ];
  // assert(bufSize);
  return &buf[--bufSize];
}

struct SegTree {
  int n;
  node *root;
  SegTree(int _n) : n(_n) {
    root = newNode();
    root->init(0, n);
  }
  ll query(int l, int r) { return root->query(l, r); }
  void update(int p, ll v) { root->update(p, v); }
};
```

1.17 SegTree Range Xor query Point Assign Update

```
template <typename T = ll>
struct SegTree {
  int n;
  T nu, nq;
  vector<T> st;
  SegTree(const vector<T> &v)
```

```
    : n(len(v)), nu(0), nq(0), st(n * 4 + 1, nu) {
    for (int i = 0; i < n; ++i) update(i, v[i]);
  }
  void update(int p, T v) { update(1, 0, n - 1, p, v); }
  T query(int l, int r) { return query(1, 0, n - 1, l, r); }

  void update(int node, int nl, int nr, int p, T v) {
    if (p < nl or p > nr) return;

    if (nl == nr) {
      st[node] = v;
      return;
    }

    update(left(node), nl, mid(nl, nr), p, v);
    update(right(node), mid(nl, nr) + 1, nr, p, v);

    st[node] = st[left(node)] ^ st[right(node)];
  }

  T query(int node, int nl, int nr, int ql, int qr) {
    if (ql <= nl and qr >= nr) return st[node];
    if (nl > qr or nr < ql) return nq;
    if (nl == nr) return st[node];

    return query(left(node), nl, mid(nl, nr), ql, qr) ^
           query(right(node), mid(nl, nr) + 1, nr, ql, qr);
  }

  int left(int p) { return p << 1; }
  int right(int p) { return (p << 1) + 1; }
  int mid(int l, int r) { return (r - l) / 2 + l; }
};
```

1.18 SegTree Range Min Query Range Sum Update

```
template <typename t = ll>
struct SegTree {
  int n;
  t nu;
  t nq;
  vector<t> st, lazy;
  SegTree(const vector<t> &xs)
    : n(len(xs)),
      nu(0),
      nq(numeric_limits<t>::max()),
      st(4 * n, nu),
      lazy(4 * n, nu) {
    for (int i = 0; i < len(xs); ++i) update(i, i, xs[i]);
  }

  SegTree(int n) : n(n), st(4 * n, nu), lazy(4 * n, nu) {}

  void update(int l, int r, ll value) {
    update(1, 0, n - 1, l, r, value);
  }

  t query(int l, int r) { return query(1, 0, n - 1, l, r); }

  void update(int node, int nl, int nr, int ql, int qr,
              ll v) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return;

    if (ql <= nl and nr <= qr) {
      st[node] += (nr - nl + 1) * v;

      if (nl < nr) {
        lazy[left(node)] += v;
        lazy[right(node)] += v;
      }

      return;
    }

    update(left(node), nl, mid(nl, nr), ql, qr, v);
    update(right(node), mid(nl, nr) + 1, nr, ql, qr, v);

    st[node] = min(st[left(node)], st[right(node)]);
  }

  t query(int node, int nl, int nr, int ql, int qr) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return nq;

    if (ql <= nl and nr <= qr) return st[node];

    t x = query(left(node), nl, mid(nl, nr), ql, qr);
```

```
    t y = query(right(node), mid(nl, nr) + 1, nr, ql, qr);

    return min(x, y);
  }

  void propagation(int node, int nl, int nr) {
    if (lazy[node]) {
      st[node] += lazy[node];

      if (nl < nr) {
        lazy[left(node)] += lazy[node];
        lazy[right(node)] += lazy[node];
      }

      lazy[node] = nu;
    }
  }

  int left(int p) { return p << 1; }
  int right(int p) { return (p << 1) + 1; }
  int mid(int l, int r) { return (r - l) / 2 + l; }
};
```

1.19 SegTree Range Sum Query Range Sum Update

```
template <typename T = ll>
struct SegTree {
  int N;
  T nu;
  T nq;
  vector<T> st, lazy;
  SegTree(const vector<T> &xs)
    : N(len(xs)),
      nu(0),
      nq(0),
      st(4 * N, nu),
      lazy(4 * N, nu) {
    for (int i = 0; i < len(xs); ++i) update(i, i, xs[i]);
  }

  SegTree(int n)
    : N(n), nu(0), nq(0), st(4 * N, nu), lazy(4 * N, nu) {}

  void update(int l, int r, ll value) {
    update(1, 0, N - 1, l, r, value);
  }

  T query(int l, int r) { return query(1, 0, N - 1, l, r); }

  void update(int node, int nl, int nr, int ql, int qr,
              ll v) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return;

    if (ql <= nl and nr <= qr) {
      st[node] += (nr - nl + 1) * v;

      if (nl < nr) {
        lazy[left(node)] += v;
        lazy[right(node)] += v;
      }

      return;
    }

    update(left(node), nl, mid(nl, nr), ql, qr, v);
    update(right(node), mid(nl, nr) + 1, nr, ql, qr, v);

    st[node] = st[left(node)] + st[right(node)];
  }

  T query(int node, int nl, int nr, int ql, int qr) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return nq;

    if (ql <= nl and nr <= qr) return st[node];

    T x = query(left(node), nl, mid(nl, nr), ql, qr);
    T y = query(right(node), mid(nl, nr) + 1, nr, ql, qr);

    return x + y;
  }

  void propagation(int node, int nl, int nr) {
    if (lazy[node]) {
      st[node] += (nr - nl + 1) * lazy[node];

      if (nl < nr) {
```

```
        lazy[left(node)] += lazy[node];
        lazy[right(node)] += lazy[node];
      }

      lazy[node] = nu;
    }
  }

  int left(int p) { return p << 1; }
  int right(int p) { return (p << 1) + 1; }
  int mid(int l, int r) { return (r - l) / 2 + l; }
};
```

1.20 Sparse Table

Answer the range query defined at the function **op**.
Build: O(NlogN), Query: O(1)

```
template <typename T>
struct SparseTable {
  vector<T> v;
  int n;
  static const int b = 30;
  vi mask, t;

  int op(int x, int y) { return v[x] < v[y] ? x : y; }
  int msb(int x) {
    return __builtin_clz(1) - __builtin_clz(x);
  }
  SparseTable() {}
  SparseTable(const vector<T>& v_)
    : v(v_), n(v.size()), mask(n), t(n) {
    for (int i = 0, at = 0; i < n; mask[i++] = at |= 1) {
      at = (at << 1) & ((1 << b) - 1);
      while (at and op(i, i - msb(at & -at)) == i)
        at ^= at & -at;
    }
    for (int i = 0; i < n / b; i++)
      t[i] = b * i + b - 1 - msb(mask[b * i + b - 1]);
    for (int j = 1; (1 << j) <= n / b; j++)
      for (int i = 0; i + (1 << j) <= n / b; i++)
        t[n / b * j + i] =
          op(t[n / b * (j - 1) + i],
             t[n / b * (j - 1) + i + (1 << (j - 1))]);
  }
  int small(int r, int sz = b) {
    return r - msb(mask[r] & ((1 << sz) - 1));
  }
  T query(int l, int r) {
    if (r - l + 1 <= b) return small(r, r - l + 1);
    int ans = op(small(l + b - 1), small(r));
    int x = l / b + 1, y = r / b - 1;
    if (x <= y) {
      int j = msb(y - x + 1);
      ans = op(ans, op(t[n / b * j + x],
                       t[n / b * j + y - (1 << j) + 1]));
    }
    return ans;
  }
};
```

2 Dynamic programming

2.1 Binary Knapsack (bottom up)

Given the points each element have, and it repespective cost, computes the maximum points we can get if we can ignore/choose an element, in such way that the sum of costs don't exceed the maximum cost allowed
Time and space: $O(N * W)$
the vectors VS and WS starts at one, so it need an empty value at index 0.

```
const int MAXN(1'000), MAXCOST(1'000 * 20);
ll dp[MAXN + 1][MAXCOST + 1];
bool ps[MAXN + 1][MAXCOST + 1];
pair<ll, vi> knapsack(const vll &points, const vi &costs,
                      int maxCost) {
  int n = len(points) - 1;  // ELEMENTS START AT INDEX 1 !

  for (int m = 0; m <= maxCost; m++) {
    dp[0][m] = 0;
  }

  for (int i = 1; i <= n; i++) {
    dp[i][0] = dp[i - 1][0] + (costs[i] == 0) * points[i];
    ps[i][0] = costs[i] == 0;
  }

  for (int i = 1; i <= n; i++) {
    for (int m = 1; m <= maxCost; m++) {
      dp[i][m] = dp[i - 1][m], ps[i][m] = 0;
      int w = costs[i];
      ll v = points[i];

      if (w <= m and dp[i - 1][m - w] + v > dp[i][m]) {
```

```
        dp[i][m] = dp[i - 1][m - w] + v, ps[i][m] = 1;
      }
    }
  }

  vi is;
  for (int i = n, m = maxCost; i >= 1; --i) {
    if (ps[i][m]) {
      is.emplace_back(i);
      m -= costs[i];
    }
  }

  return {dp[n][maxCost], is};
}
```

2.2 Binary Knapsack (top down)

Given N items, each with its own value V_i and weight W_i and a maximum knapsack weight W, compute the maximum value of the items that we can carry, if we can either ignore or take a particular item.
Assume that $1 \leq n \leq 1000$, $1 \leq S \leq 10000$.
Time and space: $O(N * W)$
the bottom up version is 5 times faster !

```
const int MAXN(2000), MAXM(2000);
ll memo[MAXN][MAXM + 1];
char choosen[MAXN][MAXM + 1];
ll knapSack(int u, int w, vll &VS, vi &WS) {
  if (u < 0) return 0;
  if (memo[u][w] != -1) return memo[u][w];

  ll a = 0, b = 0;
  a = knapSack(u - 1, w, VS, WS);
  if (WS[u] <= w)
    b = knapSack(u - 1, w - WS[u], VS, WS) + VS[u];
  if (b > a) {
    choosen[u][w] = true;
  }
  return memo[u][w] = max(a, b);
}
pair<ll, vi> knapSack(int W, vll &VS, vi &WS) {
  memset(memo, -1, sizeof(memo));
  memset(choosen, 0, sizeof(choosen));
  int n = len(VS);
  ll v = knapSack(n - 1, W, VS, WS);
  ll cw = W;
  vi choosed;
  for (int i = n - 1; i >= 0; i--) {
    if (choosen[i][cw]) {
      cw -= WS[i];
      choosed.emplace_back(i);
    }
  }
  return {v, choosed};
}
```

2.3 Edit Distance

$O(N * M)$

```
int edit_distance(const string &a, const string &b) {
  int n = a.size();
  int m = b.size();
  vector<vi> dp(n + 1, vi(m + 1, 0));

  int ADD = 1, DEL = 1, CHG = 1;
  for (int i = 0; i <= n; ++i) {
    dp[i][0] = i * DEL;
  }
  for (int i = 1; i <= m; ++i) {
    dp[0][i] = ADD * i;
  }

  for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
      int add = dp[i][j - 1] + ADD;
      int del = dp[i - 1][j] + DEL;
      int chg = dp[i - 1][j - 1] +
                (a[i - 1] == b[j - 1] ? 0 : 1) * CHG;
      dp[i][j] = min({add, del, chg});
    }
  }

  return dp[n][m];
}
```

2.4 Kadane

Find the maximum subarray sum in a given a rray.

```
int kadane(const vi &as) {
  vi s(len(as));
  s[0] = as[0];
```

```
  for (int i = 1; i < len(as); ++i)
    s[i] = max(as[i], s[i - 1] + as[i]);

  return *max_element(all(s));
}
```

2.5 Knapsack with quantity (no recover)

finds the maximum score you can achieve, given that you have n items, each item has a *cost*, a *point* and a *quantity*, you can spent at most *maxcost* and buy each item the maximum quantity it has.
time: $O(n \cdot maxcost \cdot \log maxqtd)$ memory: $O(maxcost)$.

```
ll knapsack(const vi &weight, const vll &value,
            const vi &qtd, int maxCost) {
  vi costs;
  vll values;
  for (int i = 0; i < len(weight); i++) {
    ll q = qtd[i];
    for (ll x = 1; x <= q; q -= x, x <<= 1) {
      costs.eb(x * weight[i]);
      values.eb(x * value[i]);
    }
    if (q) {
      costs.eb(q * weight[i]);
      values.eb(q * value[i]);
    }
  }

  vll dp(maxCost + 1);
  for (int i = 0; i < len(values); i++) {
    for (int j = maxCost; j > 0; j--) {
      if (j >= costs[i])
        dp[j] = max(dp[j], values[i] + dp[j - costs[i]]);
    }
  }
  return dp[maxCost];
}
```

2.6 Longest Increasing Subsequence (LIS)

Finds the lenght of the longest subsequence in

$$O(n \log n)$$

.

```
int LIS(const vi& as) {
  const ll oo = 1e18;
  int n = len(as);
  vll lis(n + 1, oo);
  lis[0] = -oo;

  auto ans = 0;

  for (int i = 0; i < n; ++i) {
    auto it = lower_bound(all(lis), as[i]);
    auto pos = (int)(it - lis.begin());

    ans = max(ans, pos);
    lis[pos] = as[i];
  }

  return ans;
}
```

2.7 Money Sum (Bottom Up)

Find every possible sum using the given values only once.

```
set<int> money_sum(const vi &xs) {
  using vc = vector<char>;
  using vvc = vector<vc>;
  int _m = accumulate(all(xs), 0);
  int _n = xs.size();
  vvc _dp(_n + 1, vc(_m + 1, 0));
  set<int> _ans;
  _dp[0][xs[0]] = 1;
  for (int i = 1; i < _n; ++i) {
    for (int j = 0; j <= _m; ++j) {
      if (j == 0 or _dp[i - 1][j]) {
        _dp[i][j + xs[i]] = 1;
        _dp[i][j] = 1;
      }
    }
  }

  for (int i = 0; i < _n; ++i)
    for (int j = 0; j <= _m; ++j)
      if (_dp[i][j]) _ans.insert(j);
  return _ans;
}
```

2.8 Travelling Salesman Problem

```
using vi = vector<int>;
vector<vi> dist;
vector<vi> memo;
/* O ( N^2 * 2^N )*/
int tsp(int i, int mask, int N) {
  if (mask == (1 << N) - 1) return dist[i][0];
  if (memo[i][mask] != -1) return memo[i][mask];
  int ans = INT_MAX << 1;
  for (int j = 0; j < N; ++j) {
    if (mask & (1 << j)) continue;
    auto t = tsp(j, mask | (1 << j), N) + dist[i][j];
    ans = min(ans, t);
  }
  return memo[i][mask] = ans;
}
```

3 Geometry

3.1 Convex Hull

Given a set of points find the smallest convex polygon that contains all the given points.
Time: $O(N \log N)$
By default it removes the collinear points, set the boolean to true if you don't want that

```
struct pt {
  double x, y;
  int id;
};

int orientation(pt a, pt b, pt c) {
  double v = a.x * (b.y - c.y) + b.x * (c.y - a.y) +
             c.x * (a.y - b.y);
  if (v < 0) return -1;  // clockwise
  if (v > 0) return +1;  // counter-clockwise
  return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
  int o = orientation(a, b, c);
  return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) {
  return orientation(a, b, c) == 0;
}
void convex_hull(vector<pt>& pts,
                 bool include_collinear = false) {
  pt p0 = *min_element(all(pts), [](pt a, pt b) {
    return make_pair(a.y, a.x) < make_pair(b.y, b.x);
  });
  sort(all(pts), [&p0](const pt& a, const pt& b) {
    int o = orientation(p0, a, b);
    if (o == 0)
      return (p0.x - a.x) * (p0.x - a.x) +
               (p0.y - a.y) * (p0.y - a.y) <
             (p0.x - b.x) * (p0.x - b.x) +
               (p0.y - b.y) * (p0.y - b.y);
    return o < 0;
  });
  if (include_collinear) {
    int i = len(pts) - 1;
    while (i >= 0 && collinear(p0, pts[i], pts.back())) i--;
    reverse(pts.begin() + i + 1, pts.end());
  }

  vector<pt> st;
  for (int i = 0; i < len(pts); i++) {
    while (st.size() > 1 && !cw(st[len(st) - 2], st.back(),
                                pts[i], include_collinear))
      st.pop_back();
    st.push_back(pts[i]);
  }

  pts = st;
}
```

3.2 Determinant

```
#include "Point.cpp"

template <typename T>
T D(const Point<T> &P, const Point<T> &Q,
    const Point<T> &R) {
  return (P.x * Q.y + P.y * R.x + Q.x * R.y) -
         (R.x * Q.y + R.y * P.x + Q.x * P.y);
}
```

3.3 Equals

```
template <typename T>
```

```
bool equals(T a, T b) {
  const double EPS{1e-9};
  if (is_floating_point<T>::value)
    return fabs(a - b) < EPS;
  else
    return a == b;
}
```

3.4 Line

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

#include "point-struct-and-utils.cpp"
using namespace std;

struct line {
  ld a, b, c;
};

// the answer is stored in the third parameter (pass by
// reference)
void pointsToLine(const point &p1, const point &p2,
                  line &l) {
  if (fabs(p1.x - p2.x) < EPS)
    // vertical line
    l = {1.0, 0.0, -p1.x};
  // default values
  else
    l = {-(ld)(p1.y - p2.y) / (p1.x - p2.x), 1.0,
         -(ld)(l.a * p1.x) - p1.y};
}
```

3.5 Point Struct And Utils (2d)

```
#include <bits/stdc++.h>
using namespace std;
using ld = long double;

struct point {
  ld x, y;
  int id;
  point(ld x = 0.0, ld y = 0.0, int id = -1)
    : x(x), y(y), id(id) {}

  point& operator+=(const point& t) {
    x += t.x;
    y += t.y;
    return *this;
  }
  point& operator-=(const point& t) {
    x -= t.x;
    y -= t.y;
    return *this;
  }
  point& operator*=(ld t) {
    x *= t;
    y *= t;
    return *this;
  }
  point& operator/=(ld t) {
    x /= t;
    y /= t;
    return *this;
  }
  point operator+(const point& t) const {
    return point(*this) += t;
  }
  point operator-(const point& t) const {
    return point(*this) -= t;
  }
  point operator*(ld t) const { return point(*this) *= t; }
  point operator/(ld t) const { return point(*this) /= t; }
};

ld dot(point& a, point& b) { return a.x * b.x + a.y * b.y; }

ld norm(point& a) { return dot(a, a); }

ld abs(point a) { return sqrt(norm(a)); }

ld proj(point a, point b) { return dot(a, b) / abs(b); }

ld angle(point a, point b) {
  return acos(dot(a, b) / abs(a) / abs(b));
}

ld cross(point a, point b) { return a.x * b.y - a.y * b.x; }
```

3.6 Polygon Lattice Points (Pick's Theorem)

Given a polygon with N points finds the number of lattice points inside and on boundaries. Time : $O(N)$

```
ll cross(ll x1, ll y1, ll x2, ll y2) {
```

```cpp
  return x1 * y2 - x2 * y1;
}

ll polygonArea(vector<pll>& pts) {
  ll ats = 0;
  for (int i = 2; i < len(pts); i++)
    ats += cross(pts[i].first - pts[0].first,
                 pts[i].second - pts[0].second,
                 pts[i - 1].first - pts[0].first,
                 pts[i - 1].second - pts[0].second);
  return abs(ats / 2ll);
}

ll boundary(vector<pll>& pts) {
  ll ats = pts.size();
  for (int i = 0; i < len(pts); i++) {
    ll deltax =
      (pts[i].first - pts[(i + 1) % pts.size()].first);
    ll deltay =
      (pts[i].second - pts[(i + 1) % pts.size()].second);
    ats += abs(__gcd(deltax, deltay)) - 1;
  }
  return ats;
}

pll latticePoints(vector<pll>& pts) {
  ll bounds = boundary(pts);
  ll area = polygonArea(pts);
  ll inside = area + 1ll - bounds / 2ll;

  return {inside, bounds};
}
```

3.7 Segment

```cpp
#include "Line.cpp"
#include "Point.cpp"
#include "equals.cpp"

template <typename T>
struct segment {
  Point<T> A, B;

  bool contains(const Point<T> &P) const;
  Point<T> closest(const Point<T> &p) const;
};

template <typename T>
bool segment<T>::contains(const Point<T> &P) const {
  // verifica se P áest contido na reta
  double dAB = Point<T>::dist(A, B),
         dAP = Point<T>::dist(A, P),
         dPB = Point<T>::dist(P, B);

  return equals(dAP + dPB, dAB);
}

template <typename T>
Point<T> segment<T>::closest(const Point<T> &P) const {
  Line<T> R(A, B);
  auto Q = R.closest(P);

  if (this->contains(Q)) return Q;

  auto distA = Point<T>::dist(P, A);
  auto distB = Point<T>::dist(P, B);

  if (distA <= distB)
    return A;
  else
    return B;
}
```

3.8 Template Line

```cpp
#include "template-point.cpp"

template <typename T>
struct Line {
  T a, b, c;

  Line(T av, T bv, T cv) : a(av), b(bv), c(cv) {}

  Line(const Point<T> &P, const Point<T> &Q)
    : a(P.y - Q.y),
      b(Q.x - P.x),
      c(P.x * Q.y - Q.x * P.y) {}

  // verify if a point belongs to the line
```

```cpp
  bool contains(const Point<T> &P) {
    return equals(a * P.x + b * P.y + c, 0);
  }

  // shortest distance between P and a point Q that belongs
  // to this line
  double distance(const Point<T> &P) const {
    return fabs(a * P.x + b * P.y + c) / hypot(a, b);
  }

  // the closest point in this line to the given point
  Point<T> closest(const Point<T> &P) const {
    auto den = (a * a) + (b * b);

    auto x = (b * (b * P.x - a * P.y) - a * c) / den;
    auto y = (a * (-b * P.x + a * P.y) - b * c) / den;

    return Point<T>{x, y};
  }
};
```

3.9 Template Point

```cpp
template <typename T>
struct Point {
  T x, y;

  Point(T xv = 0, T yv = 0) : x(xv), y(yv) {}

  double distance(const Point<T> &P) const {
    return hypot(static_cast<double>(P.x - this->x),
                 static_cast<double>(P.y - this->y));
  }
};
```

3.10 Template Segment

```cpp
#include "equals.cpp"
#include "template-line.cpp"
#include "template-point.cpp"

template <typename T>
struct Segment {
  Point<T> A, B;

  Segment(const Point<T> &a, const Point<T> &b)
    : A(a), B(b) {}

  /*
   * Verify if a given point P belongs to the segment,
   * considering that P belongs to the line defined with A
   * and B
   */
  bool contains(const Point<T> &P) const {
    return equals(A.x, B.x)
             ? min(A.y, B.y) <= P.y and P.y <= max(A.y, B.y)
             : min(A.x, B.x) <= P.x and
                 P.x <= max(A.x, B.x);
  }

  /*
   * Verify if P belongs to the segment AB,
   * even if P don't belong to the line defined with A and B
   * */
  bool contains2(const Point<T> &P) const {
    double dAB = dist(A, B), dAP = dist(A, P),
           dPB = dist(P, B);
    return equals(dAP + dPB, dAB);
  }

  /*
   * Find the closest point in P that belongs to the segment
   * */
  Point<T> closest(const Point<T> &P) {
    Line<T> r(A, B);
    auto Q = r.closest(P);

    if (this->contains(Q)) return Q;

    auto distA = P.distance(A);
    auto distB = P.distance(B);

    return distA <= distB ? A : B;
  }

  double distToClosest(const Point<T> &P) {
    return closest(P).distance(P);
  }
};
```

4 Graphs

4.1 2 SAT

```cpp
struct SAT {
  int n;
  vi2d g, tg;
  vi vis;
  vi order, comp;
  vc assignment;
  bool solvable;
  int qtdcomp;

  SAT(int _n)
    : n(2 * _n),
      g(n),
      tg(n),
      vis(n),
      comp(n, -1),
      assignment(n / 2) {}

  void dfs1(int u) {
    vis[u] = 1;
    for (auto v : g[u]) {
      if (!vis[v]) {
        dfs1(v);
      }
    }
    order.emplace_back(u);
  }

  void dfs2(int u) {
    comp[u] = qtdcomp;
    for (auto v : tg[u]) {
      if (comp[v] == -1) {
        dfs2(v);
      }
    }
  }

  bool solve2sat() {
    for (int i = 0; i < n; i++) {
      if (!vis[i]) dfs1(i);
    }
    reverse(all(order));
    qtdcomp = 0;
    for (auto u : order) {
      if (comp[u] == -1) {
        dfs2(u);
        qtdcomp++;
      }
    }

    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
      if (comp[i] == comp[i + 1]) {
        solvable = false;
        return false;
      }

      assignment[i / 2] = comp[i] < comp[i + 1];
    }

    solvable = 1;
    return solvable;
  }

  void add_dis(int a, bool va, int b, bool vb) {  // a V b
    va = !va, vb = !vb;
    a = (2 * a) ^ va, b = (2 * b) ^ vb;
    int nota = a ^ 1, notb = b ^ 1;
    g[nota].emplace_back(b), g[notb].emplace_back(a),
      tg[b].emplace_back(nota), tg[a].emplace_back(notb);
  }

  void add_impl(int a, bool va, int b, int vb) {  // a -> b
    add_dis(a, !va, b, vb);
  }

  void add_equiv(int a, bool va, int b,
                 bool vb) {  // a <-> b
    add_impl(a, 1, b, 1);
    add_impl(b, 1, a, 1);
    add_impl(a, 0, b, 0);
    add_impl(b, 0, a, 0);
  }

  void add_xor(int a, bool va, int b, bool vb) {  // a xor b
    add_impl(a, 1, b, 0);
```

```
    add_impl(a, 0, b, 1);
    add_impl(b, 1, a, 0);
    add_impl(b, 0, a, 1);
  }
};
```

4.2 Cycle Distances

Given a vertex s finds the longest cycle that end's in s, note that the vector **dist** will contain the distance that each vertex u needs to reach s.
Time: $O(N)$

```
using adj = vector<vector<pair<int, ll>>>;
ll cycleDistances(int u, int n, int s, vc &vis, adj &g,
                  vll &dist) {
  vis[u] = 1;

  for (auto [v, d] : g[u]) {
    if (v == s) {
      dist[u] = max(dist[u], d);
      continue;
    }

    if (vis[v] == 1) {
      continue;
    }

    if (vis[v] == 2) {
      dist[u] = max(dist[u], dist[v] + d);
    } else {
      ll d2 = cycleDistances(v, n, s, vis, g, dist);
      if (d2 != -oo) {
        dist[u] = max(dist[u], d2 + d);
      }
    }
  }
  vis[u] = 2;
  return dist[u];
}
```

4.3 SCC (struct)

Build the condensation graph based in the strongly connected components.
tiem: $O(V + E)$

```
struct SCC {
  int num_sccs = 0;
  vi scc_id;
  vector<set<ll>> gscc;

  SCC(const vi2d& adj)
    : scc_id(len(adj), -1), gscc(len(adj)) {
    int n = len(adj), timer = 1;
    vi tin(n), st;
    st.reserve(n);
    auto dfs = [&](auto&& self, int u) -> int {
      int low = tin[u] = timer++, siz = len(st);
      st.push_back(u);
      for (int v : adj[u])
        if (scc_id[v] < 0)
          low = min(low, tin[v] ? tin[v] : self(self, v));
      if (tin[u] == low) {
        for (int i = siz; i < len(st); i++)
          scc_id[st[i]] = num_sccs;
        st.resize(siz);
        num_sccs++;
      }
      return low;
    };
    for (int i = 0; i < n; i++)
      if (!tin[i]) dfs(dfs, i);

    for (int i = 0; i < len(adj); ++i)
      for (auto j : adj[i])
        if (scc_id[i] != scc_id[j])
          gscc[scc_id[i]].emplace(scc_id[j]);
  }
};
```

4.4 Bellman-Ford (find negative cycle)

Given a directed graph find a negative cycle by running n iterations, and if the last one produces a relaxation than there is a cycle.
Time: $O(V \cdot E)$

```
const ll oo = 2500 * 1e9;

using graph = vector<vector<pair<int, ll>>>;
vi negative_cycle(graph &g, int n) {
  vll d(n, oo);
  vi p(n, -1);
  int x = -1;
  d[0] = 0;
  for (int i = 0; i < n; i++) {
```

```
    x = -1;
    for (int u = 0; u < n; u++) {
      for (auto &[v, l] : g[u]) {
        if (d[u] + l < d[v]) {
          d[v] = d[u] + l;
          p[v] = u;
          x = v;
        }
      }
    }
  }

  if (x == -1)
    return {};
  else {
    for (int i = 0; i < n; i++) x = p[x];
    vi cycle;
    for (int v = x;; v = p[v]) {
      cycle.eb(v);
      if (v == x and len(cycle) > 1) break;
    }
    reverse(all(cycle));
    return cycle;
  }
}
```

4.5 Bellman Ford

Find shortest path from a single source to all other nodes. Can detect negative cycles.
Time: $O(V * E)$

```
bool bellman_ford(const vector<vector<pair<int, ll>>> &g,
                  int s, vector<ll> &dist) {
  int n = (int)g.size();
  dist.assign(n, LLONG_MAX);

  vector<int> count(n);
  vector<char> in_queue(n);
  queue<int> q;

  dist[s] = 0;
  q.push(s);
  in_queue[s] = true;

  while (not q.empty()) {
    int cur = q.front();
    q.pop();
    in_queue[cur] = false;

    for (auto [to, w] : g[cur]) {
      if (dist[cur] + w < dist[to]) {
        dist[to] = dist[cur] + w;
        if (not in_queue[to]) {
          q.push(to);
          in_queue[to] = true;
          count[to]++;
          if (count[to] > n) return false;
        }
      }
    }
  }

  return true;
}
```

4.6 BFS 01

Similar to a Dijkstra given a weighted graph finds the distance from source s to every other node (SSSP).
Applicable only when the weight of the edges $\in \{0, x\}$
Time: $O(V + E)$

```
vector<pair<ll, int>> adj[maxn];
ll dists[maxn];
int s, n;
void bfs_01() {
  fill(dists, dists + n, oo);
  dist[s] = 0;

  deque<int> q;
  q.emplace_back(s);

  while (not q.empty()) {
    auto u = q.front();
    q.pop_front();

    for (auto [v, w] : adj[u]) {
      if (dist[v] <= dist[u] + w) continue;
      dist[v] = dist[u] + w;
      w ? q.emplace_back(v) : q.emplace_front(v);
    }
  }
}
```

4.7 Biconnected Components

Build a vector of vectors, where the i-th vector correspond to the nodes of the i-th biconnected component, a biconnected component is a subset of nodes and edges in which there is no cut point, also exist at least two distinct routes in vertex between any two vertex in the same biconnected component.
time: $O(N + M)$

```
const int maxn(5'00'000);
int tin[maxn], stck[maxn], bcc_cnt, n, top = 0, timer = 1;
vector<int> g[maxn], nodes[maxn];

int tarjan(int u, int p = -1) {
  int lowu = tin[u] = timer++;
  int son_cnt = 0;
  stck[++top] = u;
  for (auto v : g[u]) {
    if (!tin[v]) {
      son_cnt++;
      int lowx = tarjan(v, u);
      lowu = min(lowu, lowx);
      if (lowx >= tin[u]) {
        while (top != -1 && stck[top + 1] != v)
          nodes[bcc_cnt].emplace_back(stck[top--]);
        nodes[bcc_cnt++].emplace_back(u);
      }
    } else {
      lowu = min(lowu, tin[v]);
    }
  }

  if (p == -1 && son_cnt == 0) {
    nodes[bcc_cnt++].emplace_back(u);
  }

  return lowu;
}

void build_bccs() {
  timer = 1;
  top = -1;
  memset(tin, 0, sizeof(int) * n);
  for (int i = 0; i < n; i++) nodes[i] = {};
  bcc_cnt = 0;

  for (int u = 0; u < n; u++)
    if (!tin[u]) tarjan(u);
}
```

4.8 Binary Lifting/Jumping

Given a function/successor grpah answers queries of the form which is the node after k moves starting from u.
Time: build $O(N \cdot MAXLOG2)$, query $O(MAXLOG2)$.

```
const int MAXN(2e5), MAXLOG2(30);
int bl[MAXN][MAXLOG2 + 1];
int N;

int jump(int u, ll k) {
  for (int i = 0; i <= MAXLOG2; i++) {
    if (k & (1ll << i)) u = bl[u][i];
  }
  return u;
}
void build() {
  for (int i = 1; i <= MAXLOG2; i++) {
    for (int j = 0; j < N; j++) {
      bl[j][i] = bl[bl[j][i - 1]][i - 1];
    }
  }
}
```

4.9 Block Cut Tree

```
struct block_cut_tree {
  int n;
  vector<int> id, is_cutpoint, tin, low, stk;
  vector<vector<int>> comps, tree;
  block_cut_tree(vector<vector<int>> &g)
    : n(g.size()), id(n), is_cutpoint(n), tin(n), low(n) {
    // build comps
    for (int i = 0; i < n; i++) {
      if (!tin[i]) {
        int timer = 0;
        dfs(i, -1, timer, g);
      }
    }

    int node_id = 0;
    for (int u = 0; u < n; u++) {
      if (is_cutpoint[u]) {
        id[u] = node_id++;
        tree.push_back({});
      }
```

```
    }

    for (auto &comp : comps) {
      int node = node_id++;
      tree.push_back({});
      for (int u : comp) {
        if (!is_cutpoint[u]) {
          id[u] = node;
        } else {
          tree[node].emplace_back(id[u]);
          tree[id[u]].emplace_back(node);
        }
      }
    }
  }

  void dfs(int u, int p, int &timer,
           vector<vector<int>> &g) {
    tin[u] = low[u] = ++timer;
    stk.emplace_back(u);

    for (auto v : g[u]) {
      if (v == p) continue;
      if (!tin[v]) {
        dfs(v, u, timer, g);
        low[u] = min(low[u], low[v]);
        if (low[v] >= tin[u]) {
          is_cutpoint[u] = (tin[u] > 1 or tin[v] > 2);
          comps.push_back({u});
          while (comps.back().back() != v) {
            comps.back().emplace_back(stk.back());
            stk.pop_back();
          }
        }
      } else
        low[u] = min(low[u], tin[v]);
    }
  }
};
```

4.10 Check Bipartitie

$O(V)$

```
vi2d G;
int N, M;
bool check() {
  vi side(N, -1);
  queue<int> q;
  for (int st = 0; st < N; st++) {
    if (side[st] == -1) {
      q.emplace(st);
      side[st] = 0;
      while (not q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v : G[u]) {
          if (side[v] == -1) {
            side[v] = side[u] ^ 1;
            q.push(v);
          } else if (side[u] == side[v])
            return false;
        }
      }
    }
  }
  return true;
}
```

4.11 Dijkstra (k Shortest Paths)

```
const ll oo = 1e9 * 1e5 + 1;
using adj = vector<vector<pll>>;
vector<priority_queue<ll>> dijkstra(
  const vector<vector<pll>> &g, int n, int s, int k) {
  priority_queue<pll, vector<pll>, greater<pll>> pq;

  vector<priority_queue<ll>> dist(n);
  dist[0].emplace(0);
  pq.emplace(0, s);
  while (!pq.empty()) {
    auto [d1, v] = pq.top();
    pq.pop();

    if (not dist[v].empty() and dist[v].top() < d1)
      continue;

    for (auto [d2, u] : g[v]) {
      if (len(dist[u]) < k) {
        pq.emplace(d2 + d1, u);
        dist[u].emplace(d2 + d1);
```

```
      } else {
        if (dist[u].top() > d1 + d2) {
          dist[u].pop();
          dist[u].emplace(d1 + d2);
          pq.emplace(d2 + d1, u);
        }
      }
    }
  }
  return dist;
}
```

4.12 Dijkstra

Finds the shortest path from s to every other node, and keep the 'parent' tracking.
Time: $O(E \cdot \log V)$

```
pair<vll, vi> dijkstra(const vector<vector<pll>> &g, int n,
                       int s) {
  priority_queue<pll, vector<pll>, greater<pll>> pq;
  vll dist(n, oo);
  vi p(n, -1);
  pq.emplace(0, s);
  dist[s] = 0;
  while (!pq.empty()) {
    auto [d1, v] = pq.top();
    pq.pop();
    if (dist[v] < d1) continue;

    for (auto [d2, u] : g[v]) {
      if (dist[u] > d1 + d2) {
        dist[u] = d1 + d2;
        p[u] = v;
        pq.emplace(dist[u], u);
      }
    }
  }
  return {dist, p};
}
```

4.13 Disjoint Edges Path (Maxflow)

Given a directed graph find's every path with disjoint edges that starts at s and ends at t
Time : $O(E \cdot V^2)$

```
struct DisjointPaths {
  int n;
  vi2d g, capacity;
  vector<vc> isedge;

  DisjointPaths(int _n)
    : n(_n), g(n), capacity(n, vi(n)), isedge(n, vc(n)) {}

  void add(int u, int v, int w = 1) {
    g[u].emplace_back(v);
    g[v].emplace_back(u);
    capacity[u][v] += w;
    isedge[u][v] = true;
  }

  // finds the new flow to insert
  int bfs(int s, int t, vi &parent) {
    fill(all(parent), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({oo, s});

    while (!q.empty()) {
      auto [flow, cur] = q.front();
      q.pop();

      for (auto next : g[cur]) {
        if (parent[next] == -1 and capacity[cur][next]) {
          parent[next] = cur;
          ll new_flow = min(flow, capacity[cur][next]);
          if (next == t) return new_flow;
          q.push({new_flow, next});
        }
      }
    }

    return 0;
  }

  int maxflow(int s, int t) {
    int flow = 0;
    vi parent(n);
    int new_flow;

    while ((new_flow = bfs(s, t, parent))) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
```

```
      int prev = parent[cur];
      capacity[prev][cur] -= new_flow;
      capacity[cur][prev] += new_flow;
      cur = prev;
    }
  }

  return flow;
}

// build the distinct routes based in the capacity set by
// maxflow
void dfs(int u, int t, vc2d &vis, vi &route,
         vi2d &routes) {
  route.eb(u);
  if (u == t) {
    routes.emplace_back(route);
    route.pop_back();
    return;
  }

  for (auto &v : g[u]) {
    if (capacity[u][v] == 0 and isedge[u][v] and
        not vis[u][v]) {
      vis[u][v] = true;
      dfs(v, t, vis, route, routes);
      route.pop_back();
      return;
    }
  }
}

vi2d disjoint_paths(int s, int t) {
  int mf = maxflow(s, t);
  vi2d routes;
  vi route;
  vc2d vis(n, vc(n));
  for (int i = 0; i < mf; i++)
    dfs(s, t, vis, route, routes);
  return routes;
}
};
```

4.14 Euler Path (directed)

Given a **directed** graph finds a path that visits every edge exactly once.
Time: $O(E)$

```
vector<int> euler_cycle(vector<vector<int>> &g, int u) {
  vector<int> res;

  stack<int> st;
  st.push(u);
  while (!st.empty()) {
    auto cur = st.top();
    if (g[cur].empty()) {
      res.push_back(cur);
      st.pop();
    } else {
      auto next = g[cur].back();
      st.push(next);

      g[cur].pop_back();
    }
  }

  for (auto &x : g)
    if (!x.empty()) return {};

  return res;
}

vector<int> euler_path(vector<vector<int>> &g, int first) {
  {
    int n = (int)g.size();
    vector<int> in(n), out(n);
    for (int i = 0; i < n; i++)
      for (auto x : g[i]) in[x]++, out[i]++;

    int a = 0, b = 0, c = 0;
    for (int i = 0; i < n; i++)
      if (in[i] == out[i])
        c++;
      else if (in[i] - out[i] == 1)
        b++;
      else if (in[i] - out[i] == -1)
        a++;
```

```
    if (c != n - 2 or a != 1 or b != 1) return {};
  }

  auto res = euler_cycle(g, first);
  if (res.empty()) return res;

  reverse(all(res));
  return res;
}
```

4.15 Euler Path (undirected)

Given a **undirected** graph finds a path that visits every edge exactly once.
Time: $O(E)$

```
vector<int> euler_cycle(vector<vector<int>> &g, int u) {
  vector<int> res;
  multiset<pair<int, int>> vis;

  stack<int> st;
  st.push(u);
  while (!st.empty()) {
    auto cur = st.top();

    while (!g[cur].empty()) {
      auto it = vis.find(make_pair(cur, g[cur].back()));
      if (it == vis.end()) break;
      g[cur].pop_back();
      vis.erase(it);
    }

    if (g[cur].empty()) {
      res.push_back(cur);
      st.pop();
    } else {
      auto next = g[cur].back();
      st.push(next);

      vis.emplace(next, cur);
      g[cur].pop_back();
    }
  }

  for (auto &x : g)
    if (!x.empty()) return {};

  return res;
}

vector<int> euler_path(vector<vector<int>> &g, int first) {
  int n = (int)g.size();
  int v1 = -1, v2 = -1;
  {
    bool bad = false;
    for (int i = 0; i < n; i++)
      if (g[i].size() & 1) {
        if (v1 == -1)
          v1 = i;
        else if (v2 == -1)
          v2 = i;
        else
          bad = true;
      }

    if (bad or (v1 != -1 and v2 == -1)) return {};
  }

  if (v2 != -1) {
    // insert cycle
    g[v1].push_back(v2);
    g[v2].push_back(v1);
  }

  auto res = euler_cycle(g, first);
  if (res.empty()) return res;

  if (v1 != -1) {
    for (int i = 0; i + 1 < (int)res.size(); i++) {
      if ((res[i] == v1 and res[i + 1] == v2) ||
          (res[i] == v2 and res[i + 1] == v1)) {
        vector<int> res2;
        for (int j = i + 1; j < (int)res.size(); j++)
          res2.push_back(res[j]);
        for (int j = 1; j <= i; j++) res2.push_back(res[j]);
        res = res2;
        break;
      }
    }
  }
```

```
  reverse(all(res));
  return res;
}
```

4.16 Extra Edges to Make Digraph Fully Strongly Connected

Given a directed graph G find the necessary edges to add to make the graph a single strongly connected component.
time : $O(N+M)$, memory : $O(N)$

```
struct SCC {
  int num_sccs = 0;
  vi scc_id;
  SCC(const vi2d& adj) : scc_id(len(adj), -1) {
    int n = len(adj), timer = 1;
    vi tin(n), st;
    st.reserve(n);
    auto dfs = [&](auto&& self, int u) -> int {
      int low = tin[u] = timer++, siz = len(st);
      st.push_back(u);
      for (int v : adj[u])
        if (scc_id[v] < 0)
          low = min(low, tin[v] ? tin[v] : self(self, v));
      if (tin[u] == low) {
        for (int i = siz; i < len(st); i++)
          scc_id[st[i]] = num_sccs;
        st.resize(siz);
        num_sccs++;
      }
      return low;
    };
    for (int i = 0; i < n; i++)
      if (!tin[i]) dfs(dfs, i);
  }
};

vector<array<int, 2>> extra_edges(const vi2d& adj) {
  SCC scc(adj);
  auto scc_id = scc.scc_id;
  auto num_sccs = scc.num_sccs;

  if (num_sccs == 1) return {};

  int n = len(adj);
  vi2d scc_adj(num_sccs);
  vi zero_in(num_sccs, 1);
  for (int u = 0; u < n; u++)
    for (int v : adj[u]) {
      if (scc_id[u] == scc_id[v]) continue;
      scc_adj[scc_id[u]].eb(scc_id[v]);
      zero_in[scc_id[v]] = 0;
    }

  int random_source =
    max_element(all(zero_in)) - zero_in.begin();

  vi vis(num_sccs);
  auto dfs = [&](auto&& self, int u) {
    if (empty(scc_adj[u])) return u;
    for (int v : scc_adj[u])
      if (!vis[v]) {
        vis[v] = 1;
        int zero_out = self(self, v);
        if (zero_out != -1) return zero_out;
      }
    return -1;
  };

  vector<array<int, 2>> edges;
  vi in_unused;
  for (int i = 0; i < num_sccs; i++)
    if (zero_in[i]) {
      vis[i] = 1;
      int zero_out = dfs(dfs, i);
      if (zero_out != -1)
        edges.push_back({zero_out, i});
      else
        in_unused.push_back(i);
    }

  for (int i = 1; i < len(edges); i++)
    swap(edges[i][0], edges[i - 1][0]);

  for (int i = 0; i < num_sccs; i++) {
    if (scc_adj[i].empty() && !vis[i]) {
      if (!in_unused.empty()) {
        edges.push_back({i, in_unused.back()});
        in_unused.pop_back();
      } else {
        edges.push_back({i, random_source});
      }
```

```
    }
  }

  for (int u : in_unused) edges.push_back({0, u});

  vi to_node(num_sccs);
  for (int i = 0; i < n; i++) to_node[scc_id[i]] = i;
  for (auto& [u, v] : edges) u = to_node[u], v = to_node[v];

  return edges;
}
```

4.17 Find Articulation/Cut Points

Given an **undirected** graph find it's articulation points.
articulation point (or cut vertex): is defined as a **vertex** which, when removed along with associated edges, increases thee number of connected components in the graph.
A vertex u can be an articulation point if and only if has at least 2 adjascent vertex
Time: $O(N + M)$

```
const int MAXN(100);
int N;
vi2d G;
int timer;
int tin[MAXN], low[MAXN];
set<int> cpoints;

int dfs(int u, int p = -1) {
  int cnt = 0;
  low[u] = tin[u] = timer++;
  for (auto v : G[u]) {
    if (not tin[v]) {
      cnt++;
      dfs(v, u);

      if (low[v] >= tin[u]) cpoints.insert(u);
      low[u] = min(low[u], low[v]);
    } else if (v != p)
      low[u] = min(low[u], tin[v]);
  }

  return cnt;
}
void getCutPoints() {
  memset(low, 0, sizeof(low));
  memset(tin, 0, sizeof(tin));
  cpoints.clear();

  timer = 1;
  for (int i = 0; i < N; i++) {
    if (tin[i]) continue;
    int cnt = dfs(i);
    if (cnt == 1) cpoints.erase(i);
  }
}
```

4.18 Find Bridge Tree Components

label2CC(u, p) finds the 2-edge connected component of every node.
notes: 0 indexed, it also works with not simple graphs.
time: $O(n + m)$

```
const int maxn(3'00'000);
int tin[maxn], compId[maxn], qtdComps;
vi g[maxn], stck;
int n;
int dfs(int u, int p = -1) {
  int low = tin[u] = len(stck);
  stck.emplace_back(u);

  bool multEdge = false;
  for (auto v : g[u]) {
    if (v == p and !multEdge) {
      multEdge = 1;
      continue;
    }
    low = min(low, tin[v] == -1 ? dfs(v, u) : tin[v]);
  }

  if (low == tin[u]) {
    for (int i = tin[u]; i < len(stck); i++)
      compId[stck[i]] = qtdComps;
    stck.resize(tin[u]);
    qtdComps++;
  }

  return low;
}

void label2CC() {
  memset(compId, -1, sizeof(int) * n);
```

```
  memset(tin, -1, sizeof(int) * n);

  stck.reserve(n);
  for (int i = 0; i < n; i++) {
    if (tin[i] == -1) dfs(i);
  }
}
```

4.19 Find Bridges (online)

```
// O((n+m)*log(n))
struct BridgeFinder {
  // 2ecc = 2 edge conected component
  // cc = conected component
  vector<int> parent, dsu_2ecc, dsu_cc, dsu_cc_size;
  int bridges, lca_iteration;
  vector<int> last_visit;

  BridgeFinder(int n)
    : parent(n, -1),
      dsu_2ecc(n),
      dsu_cc(n),
      dsu_cc_size(n, 1),
      bridges(0),
      lca_iteration(0),
      last_visit(n) {
    for (int i = 0; i < n; i++) {
      dsu_2ecc[i] = i;
      dsu_cc[i] = i;
    }
  }

  int find_2ecc(int v) {
    if (v == -1) return -1;
    return dsu_2ecc[v] == v
             ? v
             : dsu_2ecc[v] = find_2ecc(dsu_2ecc[v]);
  }

  int find_cc(int v) {
    v = find_2ecc(v);
    return dsu_cc[v] == v ? v
                          : dsu_cc[v] = find_cc(dsu_cc[v]);
  }

void make_root(int v) {
  v = find_2ecc(v);
  int root = v;
  int child = -1;
  while (v != -1) {
    int p = find_2ecc(parent[v]);
    parent[v] = child;
    dsu_cc[v] = root;
    child = v;
    v = p;
  }
  dsu_cc_size[root] = dsu_cc_size[child];
}

void merge_path(int a, int b) {
  ++lca_iteration;
  vector<int> path_a, path_b;
  int lca = -1;
  while (lca == -1) {
    if (a != -1) {
      a = find_2ecc(a);
      path_a.push_back(a);
      if (last_visit[a] == lca_iteration) {
        lca = a;
        break;
      }
      last_visit[a] = lca_iteration;
      a = parent[a];
    }
    if (b != -1) {
      b = find_2ecc(b);
      path_b.push_back(b);
      if (last_visit[b] == lca_iteration) {
        lca = b;
        break;
      }
      last_visit[b] = lca_iteration;
      b = parent[b];
    }
  }

  for (auto v : path_a) {
    dsu_2ecc[v] = lca;
    if (v == lca) break;
    --bridges;
```

```cpp
    }
    for (auto v : path_b) {
      dsu_2ecc[v] = lca;
      if (v == lca) break;
      --bridges;
    }
  }

  void add_edge(int a, int b) {
    a = find_2ecc(a);
    b = find_2ecc(b);

    if (a == b) return;

    int ca = find_cc(a);
    int cb = find_cc(b);

    if (ca != cb) {
      ++bridges;
      if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
        swap(a, b);
        swap(ca, cb);
      }
      make_root(a);
      parent[a] = dsu_cc[a] = b;
      dsu_cc_size[cb] += dsu_cc_size[a];
    } else {
      merge_path(a, b);
    }
  }
};
```

4.20 Find Bridges

Find every bridge in a **undirected** connected graph.
bridge: A bridge is defined as an **edge** which, when removed, increases the number of connected components in the graph.
Remember to read the graph as pair where the second is the id of the edge !
Time: $O(N + M)$

```cpp
const int MAXN(10000), MAXM(100000);
int N, M, clk, tin[MAXN], low[MAXN], isBridge[MAXM];
vector<pii> G[MAXN];

void dfs(int u, int p = -1) {
  tin[u] = low[u] = clk++;
  for (auto [v, i] : G[u]) {
    if (v == p) continue;
    if (tin[v]) {
      low[u] = min(low[u], tin[v]);
    } else {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > tin[u]) {
        isBridge[i] = 1;
      }
    }
  }
}

void findBridges() {
  fill(tin, tin + N, 0);
  fill(low, low + N, 0);
  fill(isBridge, isBridge + M, 0);
  clk = 1;
  for (int i = 0; i < N; i++) {
    if (!tin[i]) dfs(i);
  }
}
```

4.21 Find Centroid

Given a tree (don't forget to make it 'undirected'), find it's centroids.
Time: $O(V)$

```cpp
void dfs(int u, int p, int n, vi2d &g, vi &sz,
         vi &centroid) {
  sz[u] = 1;

  bool iscentroid = true;
  for (auto v : g[u])
    if (v != p) {
      dfs(v, u, n, g, sz, centroid);
      if (sz[v] > n / 2) iscentroid = false;
      sz[u] += sz[v];
    }

  if (n - sz[u] > n / 2) iscentroid = false;
  if (iscentroid) centroid.eb(u);
}
vi getCentroid(vi2d &g, int n) {
  vi centroid;
```

```
  vi sz(n);
  dfs(0, -1, n, g, sz, centroid);
  return centroid;
}
```

4.22 Floyd Warshall

Simply finds the minimal distance for each node to every other node. $O(V^3)$

```
vector<vll> floyd_warshall(const vector<vll> &adj, ll n) {
  auto dist = adj;

  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < n; ++k) {
        dist[j][k] =
          min(dist[j][k], dist[j][i] + dist[i][k]);
      }
    }
  }
  return dist;
}
```

4.23 Functional/Successor Graph

Given a functional graph find the vertice after k moves starting at u and also the distance between u and v, if it's impossible to reach v starting at u returns -1.
Time: build $O(N \cdot MAXLOG2)$, kth $O(MAXLOG2)$, dist $O(MAXLOG2)$

```
const int MAXN(2'000'000), MAXLOG2(24);
int N;
vi2d succ(MAXN, vi(MAXLOG2 + 1));
vi dst(MAXN, 0);

int vis[MAXN];
void dfsbuild(int u) {
  if (vis[u]) return;
  vis[u] = 1;
  int v = succ[u][0];
  dfsbuild(v);
  dst[u] = dst[v] + 1;
}

void build() {
  for (int i = 0; i < N; i++) {
    if (not vis[i]) dfsbuild(i);
  }
  for (int k = 1; k <= MAXLOG2; k++) {
    for (int i = 0; i < N; i++) {
      succ[i][k] = succ[succ[i][k - 1]][k - 1];
    }
  }
}

int kth(int u, ll k) {
  if (k <= 0) return u;
  for (int i = 0; i <= MAXLOG2; i++)
    if ((1ll << i) & k) u = succ[u][i];
  return u;
}

int dist(int u, int v) {
  int cu = kth(u, dst[u]);
  if (kth(u, dst[u] - dst[v]) == v)
    return dst[u] - dst[v];
  else if (kth(cu, dst[cu] - dst[v]) == v)
    return dst[u] + (dst[cu] - dst[v]);
  else
    return -1;
}
```

4.24 Graph Cycle (directed)

Given a directed graph finds a cycle (or not).
Time : $O(E)$

```
bool dfs(int v, vi2d &adj, vc &visited, vi &parent,
         vc &color, int &cycle_start, int &cycle_end) {
  color[v] = 1;
  for (int u : adj[v]) {
    if (color[u] == 0) {
      parent[u] = v;
      if (dfs(u, adj, visited, parent, color, cycle_start,
              cycle_end))
        return true;
    } else if (color[u] == 1) {
      cycle_end = v;
      cycle_start = u;
      return true;
    }
  }
  color[v] = 2;
```

```
  return false;
}

vi find_cycle(vi2d &g, int n) {
  vc visited(n);
  vi parent(n);
  vc color(n);
  int cycle_start, cycle_end;
  color.assign(n, 0);
  parent.assign(n, -1);
  cycle_start = -1;

  for (int v = 0; v < n; v++) {
    if (color[v] == 0 && dfs(v, g, visited, parent, color,
                             cycle_start, cycle_end))
      break;
  }

  if (cycle_start == -1) {
    return {};
  } else {
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v])
      cycle.push_back(v);
    cycle.push_back(cycle_start);
    reverse(cycle.begin(), cycle.end());
    return cycle;
  }
}
```

4.25 Graph Cycle (undirected)

Detects if a graph contains a cycle. If path parameter is not null, it will contain the cycle if one exists.
Time: $O(V + E)$

```
void graph_cycles(const vector<vector<int>> &g, int u,
                  int p, vector<int> &ps,
                  vector<int> &color, int &cn,
                  vector<vector<int>> &cycles) {
  if (color[u] == 2) {
    return;
  }

  if (color[u] == 1) {
    cn++;
    int cur = p;
    cycles.emplace_back();
    auto &v = cycles.back();
    v.push_back(cur);
    while (cur != u) {
      cur = ps[cur];
      v.push_back(cur);
    }
    reverse(all(v));
    return;
  }

  ps[u] = p;
  color[u] = 1;
  for (auto v : g[u]) {
    if (v != p)
      graph_cycles(g, v, u, ps, color, cn, cycles);
  }

  color[u] = 2;
}

vector<vector<int>> graph_cycles(
  const vector<vector<int>> &g) {
  vector<int> ps(g.size(), -1), color(g.size());
  int cn = 0;
  vector<vector<int>> cycles;
  for (int i = 0; i < (int)g.size(); i++)
    graph_cycles(g, i, -1, ps, color, cn, cycles);
  return cycles;
}
```

4.26 Heavy Light Decomposition

```
struct HeavyLightDecomposition {
  vector<int> parent, depth, size, heavy, head, pos;

  using SegT = int;
  static SegT op(SegT a, SegT b) { return max(a, b); }
  SegTree<SegT, op> seg;

  HeavyLightDecomposition(const vector<vector<int>> &g,
                          const vector<int> &v,
                          int root = 0)
    : parent(g.size()),
```

```
    depth(g.size()),
    size(g.size()),
    heavy(g.size(), -1),
    head(g.size()),
    pos(g.size()),
    seg((int)g.size()) {
  dfs(g, root);
  int cur_pos = 0;
  decompose(g, root, root, cur_pos);

  for (int i = 0; i < (int)g.size(); i++) {
    seg.set(pos[i], v[i]);
  }
}

SegT query_path(int a, int b) const {
  int res = 0;
  for (; head[a] != head[b]; b = parent[head[b]]) {
    if (depth[head[a]] > depth[head[b]]) swap(a, b);
    res = op(res, seg.query(pos[head[b]], pos[b]));
  }
  if (depth[a] > depth[b]) swap(a, b);
  return op(res, seg.query(pos[a], pos[b]));
}

SegT query_subtree(int a) const {
  return seg.query(pos[a], pos[a] + size[a] - 1);
}

void set(int a, int x) { seg.set(pos[a], x); }

private:
void dfs(const vector<vector<int>> &g, int u) {
  size[u] = 1;
  int mx_child_size = 0;
  for (auto x : g[u])
    if (x != parent[u]) {
      parent[x] = u;
      depth[x] = depth[u] + 1;
      dfs(g, x);
      size[u] += size[x];
      if (size[x] > mx_child_size)
        mx_child_size = size[x], heavy[u] = x;
    }
}

  void decompose(const vector<vector<int>> &g, int u, int h,
                 int &cur_pos) {
    head[u] = h;
    pos[u] = cur_pos++;
    if (heavy[u] != -1) decompose(g, heavy[u], h, cur_pos);

    for (auto x : g[u])
      if (x != parent[u] and x != heavy[u]) {
        decompose(g, x, x, cur_pos);
      }
  }
};
```

4.27 Kruskal

Find the minimum spanning tree of a graph.
Time: $O(E \log E)$
can be used to find the maximum spanning tree by changing the comparison operator in the sort

```
struct UFDS {
  vector<int> ps, sz;
  int components;

  UFDS(int n) : ps(n + 1), sz(n + 1, 1), components(n) {
    iota(all(ps), 0);
  }

  int find_set(int x) {
    return (x == ps[x] ? x : (ps[x] = find_set(ps[x])));
  }

  bool same_set(int x, int y) {
    return find_set(x) == find_set(y);
  }

  void union_set(int x, int y) {
    x = find_set(x);
    y = find_set(y);

    if (x == y) return;

    if (sz[x] < sz[y]) swap(x, y);

    ps[y] = x;
    sz[x] += sz[y];
```

```cpp
      components--;
    }
};

vector<tuple<ll, int, int>> kruskal(
  int n, vector<tuple<ll, int, int>> &edges) {
  UFDS ufds(n);
  vector<tuple<ll, int, int>> ans;

  sort(all(edges));
  for (auto [a, b, c] : edges) {
    if (ufds.same_set(b, c)) continue;

    ans.emplace_back(a, b, c);
    ufds.union_set(b, c);
  }

  return ans;
}
```

4.28 Lowest Common Ancestor (Binary Lifting)

given a directed tree, finds the LCA between two nodes using binary lifting, and answer a few queries with it.
lca: returns the LCA between the two given nodes
on_path: fids if c is in the path from a to b
Time: build $O(N \cdot MAXLOG2)$ all queries $O(MAXLOG2)$

```cpp
struct LCA {
  int n;
  const int maxlog;
  vector<vector<int>> up;
  vector<int> depth;

  LCA(const vector<vector<int>> &tree)
    : n(tree.size()),
      maxlog(ceil(log2(n))),
      up(n, vector<int>(maxlog + 1)),
      depth(n, -1) {
    for (int i = 0; i < n; i++) {
      if (depth[i] == -1) {
        depth[i] = 0;
        dfs(i, -1, tree);
      }
    }
  }

  void dfs(int u, int p, const vector<vector<int>> &tree) {
    if (p != -1) {
      depth[u] = depth[p] + 1;
      up[u][0] = p;
      for (int i = 1; i <= maxlog; i++) {
        up[u][i] = up[up[u][i - 1]][i - 1];
      }
    }
    for (int v : tree[u]) {
      if (v == p) continue;
      dfs(v, u, tree);
    }
  }

  int kth_jump(int u, int k) {
    for (int i = maxlog; i >= 0; i--) {
      if ((1 << i) & k) {
        u = up[u][i];
      }
    }
    return u;
  }

  int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);
    int diff = depth[u] - depth[v];
    u = kth_jump(u, diff);
    if (u == v) return u;
    for (int i = maxlog; i >= 0; i--) {
      if (up[u][i] != up[v][i]) {
        u = up[u][i];
        v = up[v][i];
      }
    }
    return up[u][0];
  }

  bool on_path(int u, int v, int s) {
    int uv = lca(u, v), us = lca(u, s), vs = lca(v, s);
    return (uv == s or (us == uv and vs == s) or
            (vs == uv and us == s));
  }
};
```

4.29 Lowest Common Ancestor

Given two nodes of a tree find their lowest common ancestor, or their distance
Build : $O(V)$, Queries: $O(1)$

```
template <typename T>
struct SparseTable {
  vector<T> v;
  int n;
  static const int b = 30;
  vi mask, t;

  int op(int x, int y) { return v[x] < v[y] ? x : y; }
  int msb(int x) {
    return __builtin_clz(1) - __builtin_clz(x);
  }
  SparseTable() {}
  SparseTable(const vector<T>& v_)
    : v(v_), n(v.size()), mask(n), t(n) {
    for (int i = 0, at = 0; i < n; mask[i++] = at |= 1) {
      at = (at << 1) & ((1 << b) - 1);
      while (at and op(i, i - msb(at & -at)) == i)
        at ^= at & -at;
    }
    for (int i = 0; i < n / b; i++)
      t[i] = b * i + b - 1 - msb(mask[b * i + b - 1]);
    for (int j = 1; (1 << j) <= n / b; j++)
      for (int i = 0; i + (1 << j) <= n / b; i++)
        t[n / b * j + i] =
          op(t[n / b * (j - 1) + i],
             t[n / b * (j - 1) + i + (1 << (j - 1))]);
  }
  int small(int r, int sz = b) {
    return r - msb(mask[r] & ((1 << sz) - 1));
  }
  T query(int l, int r) {
    if (r - l + 1 <= b) return small(r, r - l + 1);
    int ans = op(small(l + b - 1), small(r));
    int x = l / b + 1, y = r / b - 1;
    if (x <= y) {
      int j = msb(y - x + 1);
      ans = op(ans, op(t[n / b * j + x],
                      t[n / b * j + y - (1 << j) + 1]));
    }
    return ans;
  }
};

struct LCA {
  SparseTable<int> st;
  int n;
  vi v, pos, dep;

  LCA(const vi2d& g, int root) : n(len(g)), pos(n) {
    dfs(root, 0, -1, g);
    st = SparseTable<int>(vector<int>(all(dep)));
  }

  void dfs(int i, int d, int p, const vi2d& g) {
    v.eb(len(dep)) = i, pos[i] = len(dep), dep.eb(d);
    for (auto j : g[i])
      if (j != p) {
        dfs(j, d + 1, i, g);
        v.eb(len(dep)) = i, dep.eb(d);
      }
  }

  int lca(int a, int b) {
    int l = min(pos[a], pos[b]);
    int r = max(pos[a], pos[b]);
    return v[st.query(l, r)];
  }
  int dist(int a, int b) {
    return dep[pos[a]] + dep[pos[b]] -
           2 * dep[pos[lca(a, b)]];
  }
};
```

4.30 Maximum Flow (Dinic)

Finds the **maximum flow** in a graph network, given the **source** s and the **sink** t. Add edge from a to b with capcity c.
Complexity: time in general $O(E \cdot V^2)$, if every capacity is 1, and every vertice has in degree equal 1 or out degree equal 1 then $O(E \cdot \sqrt{V})$,
Suffle the edges list for every vertice may take you out of the worst case

```
struct Dinic {
  struct Edge {
    int to, rev;
    long long c, oc;
    long long flow() {
      return max(oc - c, 0LL);
    } // if you need flows
```

```
  };
  vector<int> lvl, ptr, q;
  vector<vector<Edge>> adj;
  Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
  void addEdge(int a, int b, long long c,
               long long rcap = 0) {
    adj[a].push_back({b, (int)adj[b].size(), c, c});
    adj[b].push_back(
      {a, (int)adj[a].size() - 1, rcap, rcap});
  }
  long long dfs(int v, int t, long long f) {
    if (v == t || !f) return f;
    for (int& i = ptr[v]; i < (int)adj[v].size(); i++) {
      Edge& e = adj[v][i];
      if (lvl[e.to] == lvl[v] + 1)
        if (long long p = dfs(e.to, t, min(f, e.c))) {
          e.c -= p, adj[e.to][e.rev].c += p;
          return p;
        }
    }
    return 0;
  }
  long long calc(int s, int t) {
    long long flow = 0;
    q[0] = s;
    for (int L = 0; L < 31; L++)
      do {  // 'int L=30' maybe faster for random data
        lvl = ptr = vector<int>(q.size());
        int qi = 0, qe = lvl[s] = 1;
        while (qi < qe && !lvl[t]) {
          int v = q[qi++];
          for (Edge e : adj[v])
            if (!lvl[e.to] && e.c >> (30 - L))
              q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
        }
        while (long long p = dfs(s, t, LLONG_MAX))
          flow += p;
      } while (lvl[t]);
    return flow;
  }
  bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

4.31 Maximum Flow (Edmonds-Karp)

Finds the **maximum flow** in a graph network, given the **source** s and the **sink** t.
Time: $O(V \cdot E^2)$

```
struct maxflow {
  int n;
  vi2d g;
  vll2d cps;
  vi ps;
  vector<vector<char>> isedge;

  maxflow(int _n)
    : n(_n),
      g(n),
      cps(n, vll(n)),
      ps(n),
      isedge(n, vc(n)) {}

  void add(int u, int v, ll c, bool set = true) {
    g[u].emplace_back(v);
    g[v].emplace_back(u);
    cps[u][v] = cps[u][v] * (!set) + c;
    isedge[u][v] = true;
  }

  ll bfs(int s, int t) {
    fill(all(ps), -1);
    ps[s] = -2;
    queue<pair<ll, int>> q;
    q.emplace(oo, s);

    while (!q.empty()) {
      auto [flow, cur] = q.front();
      q.pop();

      for (auto next : g[cur]) {
        if (ps[next] == -1 and cps[cur][next]) {
          ps[next] = cur;
          ll new_flow = min(flow, cps[cur][next]);
          if (next == t) return new_flow;
          q.emplace(new_flow, next);
        }
      }
    }
```

```cpp
    return 0ll;
  }

  ll flow(int s, int t) {
    ll flow = 0;
    ll new_flow;

    while ((new_flow = bfs(s, t))) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
        int prev = ps[cur];
        cps[prev][cur] -= new_flow;
        cps[cur][prev] += new_flow;
        cur = prev;
      }
    }

    return flow;
  }

  vector<pii> get_used() {
    vector<pii> used;
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (isedge[i][j] and cps[i][j] == 0)
          used.emplace_back(i, j);
      }
    }
    return used;
  }
};
```

4.32 Minimum Cost Flow

Given a network find the minimum cost to achieve a flow of at most f. Works with **directed** and **undirected** graphs

- **add(u, v, w, c):** adds an edge from u to v with capacity w and cost c.
- **flow(s, t, f):** return a pair $(flow, cost)$ with the maximum flow until f with source at s and sink at t, with the minimum cost possible.

Time : $O(N \cdot M + f \cdot m \log n)$

```cpp
template <typename T>
struct mcmf {
  struct edge {
    int to, rev, flow, cap;
    bool res;  // if it's a reverse edge
    T cost;    // cost per unity of flow
    edge()
      : to(0),
        rev(0),
        flow(0),
        cap(0),
        cost(0),
        res(false) {}
    edge(int to_, int rev_, int flow_, int cap_, T cost_,
         bool res_)
      : to(to_),
        rev(rev_),
        flow(flow_),
        cap(cap_),
        res(res_),
        cost(cost_) {}
  };

  vector<vector<edge>> g;
  vector<int> par_idx, par;
  T inf;
  vector<T> dist;

  mcmf(int n)
    : g(n),
      par_idx(n),
      par(n),
      inf(numeric_limits<T>::max() / 3) {}

  void add(int u, int v, int w, T cost) {
    edge a = edge(v, (int)g[v].size(), 0, w, cost, false);
    edge b = edge(u, (int)g[u].size(), 0, 0, -cost, true);

    g[u].emplace_back(a);
    g[v].emplace_back(b);
  }

  /* don't code this if there isn't negative cyles! */
  vector<T> spfa(int s) {
    deque<int> q;
    vector<char> is_inside(g.size(), 0);
    dist = vector<T>(g.size(), inf);

    dist[s] = 0;
```

```cpp
    q.push_back(s);
    is_inside[s] = true;

    while (!q.empty()) {
      int v = q.front();
      q.pop_front();
      is_inside[v] = false;

      for (int i = 0; i < (int)g[v].size(); i++) {
        auto [to, rev, flow, cap, res, cost] = g[v][i];
        if (flow < cap and dist[v] + cost < dist[to]) {
          dist[to] = dist[v] + cost;

          if (is_inside[to]) continue;
          if (!q.empty() and dist[to] > dist[q.front()])
            q.push_back(to);
          else
            q.push_front(to);
          is_inside[to] = true;
        }
      }
    }
    return dist;
  }

  bool dijkstra(int s, int t, vector<T>& pot) {
    priority_queue<pair<T, int>, vector<pair<T, int>>,
                   greater<>>
      q;
    dist = vector<T>(g.size(), inf);
    dist[s] = 0;
    q.emplace(0, s);
    while (q.size()) {
      auto [d, v] = q.top();
      q.pop();
      if (dist[v] < d) continue;
      for (int i = 0; i < (int)g[v].size(); i++) {
        auto [to, rev, flow, cap, res, cost] = g[v][i];
        cost += pot[v] - pot[to];
        if (flow < cap and dist[v] + cost < dist[to]) {
          dist[to] = dist[v] + cost;
          q.emplace(dist[to], to);
          par_idx[to] = i, par[to] = v;
        }
      }
    }

    return dist[t] < inf;
  }

  pair<int, T> min_cost_flow(int s, int t, int flow) {
    vector<T> pot((int)g.size(), 0);

    /* comment or remove this line if there isn't negative
     * cyles*/
    // pot = spfa(s);

    int f = 0;
    T ret = 0;
    while (f < flow and dijkstra(s, t, pot)) {
      for (int i = 0; i < (int)g.size(); i++)
        if (dist[i] < inf) pot[i] += dist[i];

      int mn_flow = flow - f, u = t;
      while (u != s) {
        mn_flow =
          min(mn_flow, g[par[u]][par_idx[u]].cap -
                         g[par[u]][par_idx[u]].flow);
        u = par[u];
      }

      ret += pot[t] * mn_flow;

      u = t;
      while (u != s) {
        g[par[u]][par_idx[u]].flow += mn_flow;
        g[u][g[par[u]][par_idx[u]].rev].flow -= mn_flow;
        u = par[u];
      }

      f += mn_flow;
    }

    return make_pair(f, ret);
  }
};
```

4.33 Minimum Cut (unweighted)

After build the **direct/undirected** graph find the minimum of edges needed to be removed to make the sink t unreachable from the source s.
Time: $O(V \cdot E^2)$

```
struct Mincut {
  int n;
  vi2d g;
  vii edges;
  vll2d capacity;
  vi ps, vis;

  Mincut(int _n)
    : n(_n), g(n), capacity(n, vll(n)), ps(n), vis(n) {}

  void add(int u, int v, ll c = 1, bool directed = false,
           bool set = false) {
    edges.emplace_back(u, v);
    g[u].emplace_back(v);

    if (not set)
      capacity[u][v] += c;
    else
      capacity[u][v] = c;

    if (not directed) {
      g[v].emplace_back(u);

      if (not set)
        capacity[v][u] += c;
      else
        capacity[v][u] = c;
    }
  }

  ll bfs(int s, int t) {
    fill(all(ps), -1);
    ps[s] = -2;
    queue<pair<ll, int>> q;
    q.push({oo, s});

    while (!q.empty()) {
      auto [flow, cur] = q.front();
      q.pop();

      for (auto next : g[cur]) {
        if (ps[next] == -1 and capacity[cur][next]) {
          ps[next] = cur;
          ll new_flow = min(flow, capacity[cur][next]);
          if (next == t) return new_flow;
          q.push({new_flow, next});
        }
      }
    }

    return 0ll;
  }

  ll maxflow(int s, int t) {
    ll flow = 0;
    ll new_flow;

    while ((new_flow = bfs(s, t))) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
        int prev = ps[cur];
        capacity[prev][cur] -= new_flow;
        capacity[cur][prev] += new_flow;
        cur = prev;
      }
    }

    return flow;
  }
  void dfs(int u) {
    vis[u] = true;

    for (auto v : g[u]) {
      if (capacity[u][v] > 0 and not vis[v]) {
        dfs(v);
      }
    }
  }

  vii mincut(int s, int t) {
    maxflow(s, t);
    fill(all(vis), 0);
    dfs(s);

    vii removed;
```

```
    for (auto &[u, v] : edges) {
      if ((vis[u] and not vis[v]) or
          (vis[v] and not vis[u]))
        removed.emplace_back(u, v);
    }

    return removed;
  }
};
```

4.34 Prim (MST)

Given a graph with N vertex finds the minimum spanning tree, if there is no such three returns inf, it starts using the edges that connect with each $s_i \in s$, if none is provided than it starts with the edges of node 0.
Time: $O(V \log E)$

```
const int MAXN(1'00'000);
int N;
vector<pair<ll, int>> G[MAXN];
ll prim(vi s = vi(1, 0)) {
  priority_queue<pair<ll, int>, vector<pair<ll, int>>,
                 greater<pair<ll, int>>>
    pq;
  vector<char> ingraph(MAXN);
  int ingraphcnt(0);
  for (auto si : s) {
    ingraphcnt++;
    ingraph[si] = true;
    for (auto &[w, v] : G[si]) pq.emplace(w, v);
  }

  ll mstcost = 0;
  while (ingraphcnt < N and !pq.empty()) {
    ll w;
    int v;

    do {
      tie(w, v) = pq.top();
      pq.pop();
    } while (not pq.empty() and ingraph[v]);

    mstcost += w, ingraph[v] = true, ingraphcnt++;

    for (auto &[w2, v2] : G[v]) {
      pq.emplace(w2, v2);
    }
  }

  return ingraphcnt == N ? mstcost : oo;
}
```

4.35 Small to Large

Answer queries of the form "How many vertices in the subtree of vertex v have property P?"
* this implementation answers how many distinct $values[i]$ are in the subtree starting at u.
Build: $O(N)$, Query: $O(N \log N)$

```
struct SmallToLarge {
  int n;
  vi2d tree, vis_childs;
  vi sizes, values, ans;
  set<int> cnt;

  SmallToLarge(vi2d &g, vi &v)
    : tree(g),
      vis_childs(len(g)),
      sizes(len(g)),
      values(v),
      ans(len(g)) {
    get_size(0);
    dfs(0);
  }

  inline void add_value(int u) { cnt.insert(values[u]); }

  inline void remove_value(int u) { cnt.erase(values[u]); }

  inline void update_ans(int u) { ans[u] = len(cnt); }

  void dfs(int u, int p = -1, bool keep = true) {
    int mx = -1;
    for (auto x : tree[u]) {
      if (x == p) continue;

      if (mx == -1 or sizes[mx] < sizes[x]) mx = x;
    }

    for (auto x : tree[u]) {
      if (x != p and x != mx) dfs(x, u, false);
    }

    if (mx != -1) {
```

```
      dfs(mx, u, true);
      swap(vis_childs[u], vis_childs[mx]);
    }

    vis_childs[u].push_back(u);
    add_value(u);

    for (auto x : tree[u]) {
      if (x != p and x != mx) {
        for (auto y : vis_childs[x]) {
          add_value(y);
          vis_childs[u].push_back(y);
        }
      }
    }

    update_ans(u);

    if (!keep) {
      for (auto x : vis_childs[u]) remove_value(x);
    }
  }

  void get_size(int u, int p = -1) {
    sizes[u] = 1;
    for (auto x : tree[u])
      if (x != p) {
        get_size(x, u);
        sizes[u] += sizes[x];
      }
  }
};
```

4.36 Successor Graph-(struct)

```
struct SuccessorGraph {
  vector<vector<int>> paths;
  vector<int> path_num, pos;
  vector<char> is_cycle;

  SuccessorGraph(const vector<int> &v)
    : path_num(v.size()), pos(v.size()) {
    paths.reserve(v.size());
    is_cycle.reserve(v.size());
    vector<char> vis(v.size());
    for (auto i : topological_order(v)) {
      if (vis[i]) continue;

      vector<int> path;
      int cur;
      for (cur = i; not vis[cur]; cur = v[cur]) {
        path.push_back(cur);
        vis[cur] = 1;
      }

      int cycle_start = 0;
      for (; cycle_start < (int)path.size() and
             path[cycle_start] != cur;
           cycle_start++)
        ;

      if (cycle_start > 0) {
        paths.emplace_back();
        for (int j = 0; j < cycle_start; j++) {
          paths.back().push_back(path[j]);
          pos[path[j]] = j;
          path_num[path[j]] = (int)paths.size() - 1;
        }
        paths.back().push_back(cur);
        is_cycle.push_back(false);
      }

      if (cycle_start < (int)path.size()) {
        paths.emplace_back();
        for (int j = cycle_start; j < (int)path.size();
             j++) {
          paths.back().push_back(path[j]);
          pos[path[j]] = j - cycle_start;
          path_num[path[j]] = (int)paths.size() - 1;
        }
        is_cycle.push_back(true);
      }
    }
  }

  const vector<int> &path(int cur) const {
    return paths[path_num[cur]];
  }
```

```cpp
int kth_pos(int cur, ll k) const {
  while (not is_cycle[path_num[cur]]) {
    auto &p = path(cur);
    int remain = (int)p.size() - pos[cur] - 1;
    if (k <= remain) return p[pos[cur] + k];
    cur = p.back();
    k -= remain;
  }

  auto &p = path(cur);
  return p[(pos[cur] + k) % p.size()];
}

// {element, number_of_moves}
pair<int, int> go_to_cycle(int cur) const {
  int moves = 0;
  while (not is_cycle[path_num[cur]]) {
    auto &p = path(cur);
    moves += (int)p.size() - pos[cur] - 1;
    cur = p.back();
  }
  return {cur, moves};
}

// min cost to reach dest from cur
int reach(int cur, int dest) const {
  int moves = 0;
  while (not is_cycle[path_num[cur]] and
         path_num[cur] != path_num[dest]) {
    auto &p = path(cur);
    moves += (int)p.size() - pos[cur] - 1;
    cur = p.back();
  }

  if (path_num[cur] != path_num[dest]) return -1;

  if (pos[cur] <= pos[dest])
    return moves + pos[dest] - pos[cur];

  if (not is_cycle[path_num[cur]]) return -1;

  return moves + pos[dest] + (int)path(cur).size() -
         pos[cur];
}

 private:
  void topological_order(const vector<int> &g,
                         vector<char> &vis,
                         vector<int> &order, int u) {
    vis[u] = true;
    if (not vis[g[u]])
      topological_order(g, vis, order, g[u]);
    order.push_back(u);
  }

  vector<int> topological_order(const vector<int> &g) {
    vector<char> vis(g.size(), false);
    vector<int> order;
    for (auto i = 0; i < (int)g.size(); i++)
      if (not vis[i]) topological_order(g, vis, order, i);
    reverse(order.begin(), order.end());
    return order;
  }
};
```

4.37 Sum every node distance

Given a **tree**, for each node i find the sum of distance from i to every other node.
don't forget to set the tree as undirected, that's needed to choose an arbitrary root
Time: $O(N)$

```cpp
void getRoot(int u, int p, vi2d &g, vll &d, vll &cnt) {
  for (int i = 0; i < len(g[u]); i++) {
    int v = g[u][i];
    if (v == p) continue;
    getRoot(v, u, g, d, cnt);
    d[u] += d[v] + cnt[v];
    cnt[u] += cnt[v];
  }
}

void dfs(int u, int p, vi2d &g, vll &cnt, vll &ansd,
         int n) {
  for (int i = 0; i < len(g[u]); i++) {
    int v = g[u][i];
    if (v == p) continue;

    ansd[v] = ansd[u] - cnt[v] + (n - cnt[v]);
    dfs(v, u, g, cnt, ansd, n);
  }
}
```

```
vll fromToAll(vi2d &g, int n) {
  vll d(n);
  vll cnt(n, 1);
  getRoot(0, -1, g, d, cnt);

  vll ansdist(n);
  ansdist[0] = d[0];

  dfs(0, -1, g, cnt, ansdist, n);
  return ansdist;
}
```

4.38 Topological Labelling (Kahn)

The same thing as topological sorting but over every possible order gives lexicographically smaller
Time: $O(E + V \cdot \log V)$

```
const int MAXN(1'00'000);
int OUTCNT[MAXN];
vi2d GIN(MAXN);
int N;

vi toposort() {
  vi order;
  priority_queue<int> q;

  for (int i = 0; i < N; i++)
    if (!OUTCNT[i]) q.emplace(i);

  while (!q.empty()) {
    auto u = q.top();
    q.pop();
    order.emplace_back(u);
    for (auto v : GIN[u]) {
      OUTCNT[v]--;
      if (OUTCNT[v] == 0) q.emplace(v);
    }
  }

  reverse(all(order));
  return len(order) == N ? order : vi();
}
```

4.39 Topological Sorting (Kahn)

Finds the topological sorting in a **DAG**, if the given graph is not a **DAG** than an empty vector is returned, need to 'initialize' the **INCNT** as you build the graph.

Time: $O(V + E)$

```
const int MAXN(2'00'000);
int INCNT[MAXN];
vi2d GOUT(MAXN);
int N;

vi toposort() {
  vi order;
  queue<int> q;

  for (int i = 0; i < N; i++)
    if (!INCNT[i]) q.emplace(i);

  while (!q.empty()) {
    auto u = q.front();
    q.pop();
    order.emplace_back(u);
    for (auto v : GOUT[u]) {
      INCNT[v]--;
      if (INCNT[v] == 0) q.emplace(v);
    }
  }

  return len(order) == N ? order : vi();
}
```

4.40 Topological Sorting (Tarjan)

Finds a the topological order for the graph, if there is no such order it means the graph is cyclic, then it returns an empty vector
$O(V + E)$

```
const int maxn(1'00'000);
int n, m;
vi g[maxn];

int not_found = 0, found = 1, processed = 2;
int state[maxn];

bool dfs(int u, vi &order) {
  if (state[u] == processed) return true;
  if (state[u] == found) return false;

  state[u] = found;

  for (auto v : g[u]) {
```

```
    if (not dfs(v, order)) return false;
  }

  state[u] = processed;
  order.emplace_back(u);

  return true;
}

vi topo_sort() {
  vi order;
  memset(state, 0, sizeof state);

  for (int u = 0; u < n; u++) {
    if (state[u] == not_found and not dfs(u, order))
      return {};
  }

  reverse(all(order));
  return order;
}
```

4.41 Tree Diameter (DP)

```
const int MAXN(1'000'000);
int N;
vi G[MAXN];

int diameter, toLeaf[MAXN];
void calcDiameter(int u = 0, int p = -1) {
  int d1, d2;
  d1 = d2 = -1;

  for (auto v : G[u]) {
    if (v != p) {
      calcDiameter(v, u);
      d1 = max(d1, toLeaf[v]);
      tie(d1, d2) = minmax({d1, d2});
    }
  }
  toLeaf[u] = d2 + 1;
  diameter = max(diameter, d1 + d2 + 2);
}
```

4.42 Tree Isomorphism (not rooted)

Two trees are considered **isomorphic** if the hash given by *thash*() is the same.
Time: $O(V \cdot \log V)$

```
map<vi, int> mphash;

struct Tree {
  int n;
  vi2d g;
  vi sz, cs;

  Tree(int n_) : n(n_), g(n), sz(n) {}

  void add_edge(int u, int v) {
    g[u].emplace_back(v);
    g[v].emplace_back(u);
  }

  void dfs_centroid(int v, int p) {
    sz[v] = 1;
    bool cent = true;
    for (int u : g[v])
      if (u != p) {
        dfs_centroid(u, v);
        sz[v] += sz[u];
        cent &= not(sz[u] > n / 2);
      }
    if (cent and n - sz[v] <= n / 2) cs.push_back(v);
  }

  int fhash(int v, int p) {
    vi h;
    for (int u : g[v])
      if (u != p) h.push_back(fhash(u, v));
    sort(all(h));
    if (!mphash.count(h)) mphash[h] = mphash.size();
    return mphash[h];
  }

  ll thash() {
    cs.clear();
    dfs_centroid(0, -1);
    if (cs.size() == 1) return fhash(cs[0], -1);
    ll h1 = fhash(cs[0], cs[1]), h2 = fhash(cs[1], cs[0]);
    return (min(h1, h2) << 30ll) + max(h1, h2);
```

```
  }
};
```

4.43 Tree Isomorphism (rooted)

Given a rooted tree find the hash of each subtree, if two roots of two distinct trees have the same hash they are considered isomorphic
hash first time in $O(\log N_v \cdot N_v)$ where (N_v) is the of the subtree of v

```
map<vi, int> hasher;
int hs = 0;
struct RootedTreeIso {
  int n;
  vi2d adj;
  vi hashes;
  RootedTreeIso(int _n) : n(_n), adj(_n), hashes(_n, -1){};

  void add_edge(int u, int v) {
    adj[u].emplace_back(v);
    adj[v].emplace_back(u);
  }

  int hash(int u, int p = -1) {
    if (hashes[u] != -1) return hashes[u];

    vi children;
    for (auto v : adj[u])
      if (v != p) children.emplace_back(hash(v, u));

    sort(all(children));
    if (!hasher.count(children)) hasher[children] = hs++;

    return hashes[u] = hasher[children];
  }
};
```

4.44 Tree Maximum Distance

Returns the maximum distance from every node to any other node in the tree.
$O(6V) = O(V)$

```
pll mostDistantFrom(const vector<vll> &adj, ll n, ll root) {
  // O(V)
  // 0 indexed
  ll mostDistantNode = root;
  ll nodeDistance = 0;
  queue<pll> q;
  vector<char> vis(n);
  q.emplace(root, 0);
  vis[root] = true;
  while (!q.empty()) {
    auto [node, dist] = q.front();
    q.pop();
    if (dist > nodeDistance) {
      nodeDistance = dist;
      mostDistantNode = node;
    }
    for (auto u : adj[node]) {
      if (!vis[u]) {
        vis[u] = true;
        q.emplace(u, dist + 1);
      }
    }
  }
  return {mostDistantNode, nodeDistance};
}

ll twoNodesDist(const vector<vll> &adj, ll n, ll a, ll b) {
  queue<pll> q;
  vector<char> vis(n);
  q.emplace(a, 0);
  while (!q.empty()) {
    auto [node, dist] = q.front();
    q.pop();
    if (node == b) return dist;
    for (auto u : adj[node]) {
      if (!vis[u]) {
        vis[u] = true;
        q.emplace(u, dist + 1);
      }
    }
  }
  return -1;
}

tuple<ll, ll, ll> tree_diameter(const vector<vll> &adj,
                                ll n) {
  // returns two points of the diameter and the diameter
  // itself
  auto [node1, dist1] = mostDistantFrom(adj, n, 0);  // O(V)
  auto [node2, dist2] =
    mostDistantFrom(adj, n, node1);  // O(V)
```

```cpp
  auto diameter =
    twoNodesDist(adj, n, node1, node2);  // O(V)
  return make_tuple(node1, node2, diameter);
}

vll everyDistanceFromNode(const vector<vll> &adj, ll n,
                          ll root) {
  // Single Source Shortest Path, from a given root
  queue<pair<ll, ll>> q;
  vll ans(n, -1);
  ans[root] = 0;
  q.emplace(root, 0);
  while (!q.empty()) {
    auto [u, d] = q.front();
    q.pop();

    for (auto w : adj[u]) {
      if (ans[w] != -1) continue;
      ans[w] = d + 1;
      q.emplace(w, d + 1);
    }
  }
  return ans;
}

vll maxDistances(const vector<vll> &adj, ll n) {
  auto [node1, node2, diameter] =
    tree_diameter(adj, n);  // O(3V)
  auto distances1 =
    everyDistanceFromNode(adj, n, node1);  // O(V)
  auto distances2 =
    everyDistanceFromNode(adj, n, node2);  // O(V)
  vll ans(n);
  for (int i = 0; i < n; ++i)
    ans[i] = max(distances1[i], distances2[i]);  // O(V)
  return ans;
}
```

4.45 Tree Flatten

```cpp
void tree_flatten(const vector<vector<int>> &g, int u,
                  int p, vector<int> &pre, vector<int> &pos,
                  int &idx) {
  ++idx;
  pre.push_back(u);
  for (auto x : g[u])
    if (x != p) tree_flatten(g, x, u, pre, pos, idx);
  pos[u] = idx;
}

pair<vector<int>, vector<int>> tree_flatten(
  const vector<vector<int>> &g, int root = 0) {
  vector<int> first(g.size()), last(g.size()), pre;
  int timer = -1;
  tree_flatten(g, root, -1, pre, last, timer);
  for (int i = 0; i < (int)g.size(); i++) first[pre[i]] = i;
  return {first, last};
}
```

5 Math

5.1 GCD (with factorization)

$O(\sqrt{n})$ due to factorization.

```cpp
ll gcd_with_factorization(ll a, ll b) {
  map<ll, ll> fa = factorization(a);
  map<ll, ll> fb = factorization(b);
  ll ans = 1;
  for (auto fai : fa) {
    ll k = min(fai.second, fb[fai.first]);
    while (k--) ans *= fai.first;
  }
  return ans;
}
```

5.2 GCD

```cpp
ll gcd(ll a, ll b) { return b ? gcd(b, a % b) : a; }
```

5.3 LCM (with factorization)

$O(\sqrt{n})$ due to factorization.

```cpp
ll lcm_with_factorization(ll a, ll b) {
  map<ll, ll> fa = factorization(a);
  map<ll, ll> fb = factorization(b);
  ll ans = 1;
  for (auto fai : fa) {
    ll k = max(fai.second, fb[fai.first]);
    while (k--) ans *= fai.first;
  }
  return ans;
```

```
}
```

5.4 LCM

```
ll gcd(ll a, ll b) { return b ? gcd(b, a % b) : a; }
ll lcm(ll a, ll b) { return a / gcd(a, b) * b; }
```

5.5 Arithmetic Progression Sum

- s : first term
- d : common difference
- n : number of terms

```
ll arithmeticProgressionSum(ll s, ll d, ll n) {
  return (s + (s + d * (n - 1))) * n / 2ll;
}
```

5.6 Binomial MOD

Precompute every factorial until $maxn$ ($O(maxn)$) allowing to answer the $\binom{n}{k}$ in $O(\log mod)$ time, due to the fastpow. Note that it needs $O(maxn)$ in memory.

```
const ll MOD = 1e9 + 7;
const ll MAXN = 1'00'000;
ll FACT[MAXN + 1];
void precompute() {
#warning Remember to call precompute before use binommod !
  FACT[0] = 1;
  for (ll i = 1; i <= MAXN; i++) {
    FACT[i] = (FACT[i - 1] * i) % MOD;
  }
}

ll fpow(ll a, ll k) {
  ll ret = 1;
  while (k) {
    if (k & 1) ret = (ret * a) % MOD;
    a = (a * a) % MOD;
    k >>= 1;
  }
  return ret;
}

ll binommod(ll n, ll k) {
  ll upper = FACT[n];
  ll lower = (FACT[k] * FACT[n - k]) % MOD;
  return (upper * fpow(lower, MOD - 2ll)) % MOD;
}
```

5.7 Binomial

O(nm) time, O(m) space
Equal to n choose k

```
ll binom(ll n, ll k) {
  if (k > n) return 0;
  vll dp(k + 1, 0);
  dp[0] = 1;
  for (ll i = 1; i <= n; i++)
    for (ll j = k; j > 0; j--) dp[j] = dp[j] + dp[j - 1];
  return dp[k];
}
```

5.8 Chinese Remainder Theorem

Finds the solution x to the n modular equations.

$$\begin{gathered} x \equiv a_1 (mod m_1) \\ \dots \\ x \equiv a_n (mod m_n) \end{gathered} \tag{1}$$

The m_i don't need to be coprime, if there is no solution then it returns -1.

```
tuple<ll, ll, ll> ext_gcd(ll a, ll b) {
  if (!a) return {b, 0, 1};
  auto [g, x, y] = ext_gcd(b % a, a);
  return {g, y - b / a * x, x};
}

template <typename T = ll>
struct crt {
  T a, m;

  crt() : a(0), m(1) {}
  crt(T a_, T m_) : a(a_), m(m_) {}
  crt operator*(crt C) {
    auto [g, x, y] = ext_gcd(m, C.m);
    if ((a - C.a) % g != 0) a = -1;
    if (a == -1 or C.a == -1) return crt(-1, 0);
    T lcm = m / g * C.m;
    T ans = a + (x * (C.a - a) / g % (C.m / g)) * m;
    return crt((ans % lcm + lcm) % lcm, lcm);
  }
};
```

```cpp
template <typename T = ll>
struct Congruence {
  T a, m;
};

template <typename T = ll>
T chinese_remainder_theorem(
  const vector<Congruence<T>> &equations) {
  crt<T> ans;

  for (auto &[a_, m_] : equations) {
    ans = ans * crt<T>(a_, m_);
  }

  return ans.a;
}
```

5.9 Derangement / Matching Problem

Computes the derangement of N, which is given by the formula :
$D_N = N!\left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots + (-1)^N \frac{1}{N!}\right)$
time: $O(N)$

```cpp
#warning Remember to call precompute !
const ll MOD = 1e9 + 7;
const int MAXN(1'000'000);
ll fats[MAXN + 1];
void precompute() {
  fats[0] = 1;
  for (ll i = 1; i <= MAXN; i++) {
    fats[i] = (fats[i - 1] * i) % MOD;
  }
}

ll fastpow(ll a, ll p, ll m) {
  ll ret = 1;
  while (p) {
    if (p & 1) ret = (ret * a) % MOD;
    p >>= 1;
    a = (a * a) % MOD;
  }
  return ret;
}

ll divmod(ll a, ll b) {
  return (a * fastpow(b, MOD - 2, MOD)) % MOD;
}

ll derangement(const ll n) {
  ll ans = fats[n];
  for (ll i = 1; i <= n; i++) {
    ll k = divmod(fats[n], fats[i]);
    if (i & 1) {
      ans = (ans - k + MOD) % MOD;
    } else {
      ans = (ans + k) % MOD;
    }
  }
  return ans;
}
```

5.10 Euler phi $\varphi(n)$ (in range)

Computes the number of positive integers less than n that are coprimes with n, in the range $[1, n]$, in $O(N \log N)$.

```cpp
const int MAX = 1e6;
vi range_phi(int n) {
  bitset<MAX> sieve;
  vi phi(n + 1);

  iota(phi.begin(), phi.end(), 0);
  sieve.set();

  for (int p = 2; p <= n; p += 2) phi[p] /= 2;
  for (int p = 3; p <= n; p += 2) {
    if (sieve[p]) {
      for (int j = p; j <= n; j += p) {
        sieve[j] = false;
        phi[j] /= p;
        phi[j] *= (p - 1);
      }
    }
  }

  return phi;
}
```

5.11 Euler phi $\varphi(n)$

Computes the number of positive integers less than n that are coprimes with n, in $O(\sqrt{N})$.

```cpp
int phi(int n) {
  if (n == 1) return 1;

  auto fs = factorization(n);  // a vctor of pair or a map
  auto res = n;

  for (auto [p, k] : fs) {
    res /= p;
    res *= (p - 1);
  }

  return res;
}
```

5.12 Factorial Factorization

Computes the factorization of $n!$ in $\pi(N) * \log n$

```cpp
// O(logN)
ll E(ll n, ll p) {
  ll k = 0, b = p;
  while (b <= n) {
    k += n / b;
    b *= p;
  }
  return k;
}

// O(pi(N)*logN)
map<ll, ll> factorial_factorization(ll n,
                                    const vll &primes) {
  map<ll, ll> fs;
  for (const auto &p : primes) {
    if (p > n) break;
    fs[p] = E(n, p);
  }
  return fs;
}
```

5.13 Factorial

```cpp
const ll MAX = 18;
vll fv(MAX, -1);
ll factorial(ll n) {
  if (fv[n] != -1) return fv[n];
  if (n == 0) return 1;
  return n * factorial(n - 1);
}
```

5.14 Factorization (Pollard Rho)

Factorizes a number into its prime factors in $O(n^{(\frac{1}{4})} * \log(n))$.

```cpp
ll mul(ll a, ll b, ll m) {
  ll ret = a * b - (ll)((ld)1 / m * a * b + 0.5) * m;
  return ret < 0 ? ret + m : ret;
}

ll pow(ll a, ll b, ll m) {
  ll ans = 1;
  for (; b > 0; b /= 2ll, a = mul(a, a, m)) {
    if (b % 2ll == 1) ans = mul(ans, a, m);
  }
  return ans;
}

bool prime(ll n) {
  if (n < 2) return 0;
  if (n <= 3) return 1;
  if (n % 2 == 0) return 0;

  ll r = __builtin_ctzll(n - 1), d = n >> r;
  for (int a :
       {2, 325, 9375, 28178, 450775, 9780504, 795265022}) {
    ll x = pow(a, d, n);
    if (x == 1 or x == n - 1 or a % n == 0) continue;

    for (int j = 0; j < r - 1; j++) {
      x = mul(x, x, n);
      if (x == n - 1) break;
    }
    if (x != n - 1) return 0;
  }
  return 1;
}

ll rho(ll n) {
  if (n == 1 or prime(n)) return n;
  auto f = [n](ll x) { return mul(x, x, n) + 1; };

  ll x = 0, y = 0, t = 30, prd = 2, x0 = 1, q;
```

```
  while (t % 40 != 0 or gcd(prd, n) == 1) {
    if (x == y) x = ++x0, y = f(x);
    q = mul(prd, abs(x - y), n);
    if (q != 0) prd = q;
    x = f(x), y = f(f(y)), t++;
  }
  return gcd(prd, n);
}

vll fact(ll n) {
  if (n == 1) return {};
  if (prime(n)) return {n};
  ll d = rho(n);
  vll l = fact(d), r = fact(n / d);
  l.insert(l.end(), r.begin(), r.end());
  return l;
}
```

5.15 Factorization

Computes the factorization of n in $O(\sqrt{n})$.

```
map<ll, ll> factorization(ll n) {
  map<ll, ll> ans;
  for (ll i = 2; i * i <= n; i++) {
    ll count = 0;
    for (; n % i == 0; count++, n /= i)
      ;
    if (count) ans[i] = count;
  }
  if (n > 1) ans[n]++;
  return ans;
}
```

5.16 Fast pow

Computes $a^b \pmod{m}$ in $O(\log N)$.

```
ll fpow(ll a, ll b, ll m) {
  ll ret = 1;
  while (b) {
    if (b & 1) ret = (ret * a) % m;
    b >>= 1;
    a = (a * a) % m;
  }
  return ret;
}

ll fpow(ll a, ll b, ll m) {
  if (!b) return 1;
  ll ans = fpow2((a * a) % m, b / 2ll, m);
  return b & 1 ? (a * ans) % m : ans;
}
```

5.17 FFT Convolution

Performs convolution in a vector duh !

```
const ld PI = acos(-1);

/* change the ld to doulbe may increase performance =D */
struct num {
  ld a{0.0}, b{0.0};
  num() {}
  num(ld na) : a{na} {}
  num(ld na, ld nb) : a{na}, b{nb} {}

  const num operator+(const num& c) const {
    return num(a + c.a, b + c.b);
  }
  const num operator-(const num& c) const {
    return num(a - c.a, b - c.b);
  }
  const num operator*(const num& c) const {
    return num(a * c.a - b * c.b, a * c.b + b * c.a);
  }
  const num operator/(const ll& c) const {
    return num(a / c, b / c);
  }
};

void fft(vector<num>& a, bool invert) {
  int n = len(a);
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1) j ^= bit;
    j ^= bit;
    if (i < j) swap(a[i], a[j]);
  }
  for (int sz = 2; sz <= n; sz <<= 1) {
    ld ang = 2 * PI / sz * (invert ? -1 : 1);
    num wsz(cos(ang), sin(ang));
```

```
    for (int i = 0; i < n; i += sz) {
      num w(1);
      rep(j, 0, sz / 2) {
        num u = a[i + j], v = a[i + j + sz / 2] * w;
        a[i + j] = u + v;
        a[i + j + sz / 2] = u - v;
        w = w * wsz;
      }
    }
  }
  if (invert)
    for (num& x : a) x = x / n;
}

vi conv(vi const a, vi const b) {
  vector<num> fa(all(a));
  vector<num> fb(all(b));
  int n = 1;
  while (n < len(a) + len(b)) n <<= 1;
  fa.resize(n);
  fb.resize(n);
  fft(fa, false);
  fft(fb, false);
  rep(i, 0, n) fa[i] = fa[i] * fb[i];
  fft(fa, true);
  vi result(n);
  rep(i, 0, n) result[i] = round(fa[i].a);
  while (len(result) and result.back() == 0)
    result.pop_back();

  /* Unconment this line if you want a boolean convolution*/
  for (auto& xi : result) xi = min(xi, 1ll);

  return result;
}
vll poly_exp(vll& ps, int k) {
  vll ret(len(ps));
  auto base = ps;
  ret[0] = 1;

  while (k) {
    if (k & 1) ret = conv(ret, base);
    k >>= 1;
    base = conv(base, base);
  }
  return ret;
}
```

5.18 Find Multiplicative Inverse

```
ll inv(ll a, ll m) {
  return a > 1ll ? m - inv(m % a, a) * m / a : 1ll;
}
```

5.19 Linear Diophantine Equation: Find any solution

Given a b, c finds the solution to the equation $ax + by = c$, the result will be stored in the reference variables $x0$ and $y0$
time: $O(\log min(a, b))$

```
template <typename T>
tuple<T, T, T> ext_gcd(T a, T b) {
  if (b == 0) return {a, 1, 0};

  auto [d, x1, y1] = ext_gcd(b, a % b);

  return {d, y1, x1 - y1 * (a / b)};
}

template <typename T>
tuple<bool, T, T> find_any_solution(T a, T b, T c) {
  assert(a != 0 or b != 0);
#warning Be careful with overflow, use __int128 if needed !

  auto [d, x0, y0] =
    ext_gcd(a < 0 ? -a : a, b < 0 ? -b : b);

  if (c % d) return {false, 0, 0};

  x0 *= c / d;
  y0 *= c / d;
  if (a < 0) x0 = -x0;
  if (b < 0) y0 = -y0;

  return {true, x0, y0};
}

// optional if you want to use __int128
void print(__int128 x) {
  if (x < 0) {
    cout << '-';
```

```
    x = -x;
  }
  if (x > 9) print(x / 10);
  cout << (char)((x % 10) + '0');
}

__int128 read() {
  string s;
  cin >> s;
  __int128 x = 0;
  for (auto c : s) {
    if (c != '-') x += c - '0';
    x *= 10;
  }
  x /= 10;
  if (s[0] == '-') x = -x;
  return x;
}
```

5.20 Gauss Elimination

```
template <size_t Dim>
struct GaussianElimination {
  vector<ll> basis;
  size_t size;

  GaussianElimination() : basis(Dim + 1), size(0) {}

  void insert(ll x) {
    for (ll i = Dim; i >= 0; i--) {
      if ((x & 1ll << i) == 0) continue;

      if (!basis[i]) {
        basis[i] = x;
        size++;
        break;
      }

      x ^= basis[i];
    }
  }

  void normalize() {
    for (ll i = Dim; i >= 0; i--)
      for (ll j = i - 1; j >= 0; j--)
        if (basis[i] & 1ll << j) basis[i] ^= basis[j];
  }

  bool check(ll x) {
    for (ll i = Dim; i >= 0; i--) {
      if ((x & 1ll << i) == 0) continue;

      if (!basis[i]) return false;

      x ^= basis[i];
    }

    return true;
  }

  auto operator[](ll k) { return at(k); }

  ll at(ll k) {
    ll ans = 0;
    ll total = 1ll << size;
    for (ll i = Dim; ~i; i--) {
      if (!basis[i]) continue;

      ll mid = total >> 1ll;
      if ((mid < k and (ans & 1ll << i) == 0) ||
          (k <= mid and (ans & 1ll << i)))
        ans ^= basis[i];

      if (mid < k) k -= mid;

      total >>= 1ll;
    }
    return ans;
  }

  ll at_normalized(ll k) {
    ll ans = 0;
    k--;
    for (size_t i = 0; i <= Dim; i++) {
      if (!basis[i]) continue;
      if (k & 1) ans ^= basis[i];
      k >>= 1;
    }
    return ans;
  }
```

```
};
```

5.21 Integer Partition

Find the total of ways to partition a given number N in such way that none of the parts is greater than K.
Remember to memset everything to -1 before using it
time: $O(N \cdot min(N, K))$
memory: $O(N)$

```
const ll MOD = 1000000007;
const int MAXN(100);
ll memo[MAXN + 1];
ll dp(ll n, ll k = oo) {
  if (n == 0) return 1;
  ll &ans = memo[n];
  if (ans != -1) return ans;

  ans = 0;
  for (int i = 1; i <= min(n, k); i++) {
    ans = (ans + dp(n - i, k)) % MOD;
  }

  return ans;
}
```

5.22 Integer Mod

```
const ll INF = 1e18;
const ll mod = 998244353;
template <ll MOD = mod>
struct Modular {
  ll value;
  static const ll MOD_value = MOD;

  Modular(ll v = 0) {
    value = v % MOD;
    if (value < 0) value += MOD;
  }
  Modular(ll a, ll b) : value(0) {
    *this += a;
    *this /= b;
  }

  Modular& operator+=(Modular const& b) {
    value += b.value;
    if (value >= MOD) value -= MOD;
    return *this;
  }
  Modular& operator-=(Modular const& b) {
    value -= b.value;
    if (value < 0) value += MOD;
    return *this;
  }
  Modular& operator*=(Modular const& b) {
    value = (ll)value * b.value % MOD;
    return *this;
  }

  friend Modular mexp(Modular a, ll e) {
    Modular res = 1;
    while (e) {
      if (e & 1) res *= a;
      a *= a;
      e >>= 1;
    }
    return res;
  }
  friend Modular inverse(Modular a) {
    return mexp(a, MOD - 2);
  }

  Modular& operator/=(Modular const& b) {
    return *this *= inverse(b);
  }
  friend Modular operator+(Modular a, Modular const b) {
    return a += b;
  }
  Modular operator++(int) {
    return this->value = (this->value + 1) % MOD;
  }
  Modular operator++() {
    return this->value = (this->value + 1) % MOD;
  }
  friend Modular operator-(Modular a, Modular const b) {
    return a -= b;
  }
  friend Modular operator-(Modular const a) {
    return 0 - a;
  }
  Modular operator--(int) {
    return this->value = (this->value - 1 + MOD) % MOD;
  }
```

```cpp
  Modular operator--() {
    return this->value = (this->value - 1 + MOD) % MOD;
  }
  friend Modular operator*(Modular a, Modular const b) {
    return a *= b;
  }
  friend Modular operator/(Modular a, Modular const b) {
    return a /= b;
  }
  friend std::ostream& operator<<(std::ostream& os,
                                  Modular const& a) {
    return os << a.value;
  }
  friend bool operator==(Modular const& a,
                         Modular const& b) {
    return a.value == b.value;
  }
  friend bool operator!=(Modular const& a,
                         Modular const& b) {
    return a.value != b.value;
  }
};
```

5.23 Matrix Exponentiation

```cpp
ll MOD = 1'000'000'007;

template <typename T>
vector<vector<T>> prod(vector<vector<T>> &a,
                       vector<vector<T>> &b) {
  int n = len(a);
  vector<vector<T>> c(n, vector<T>(n));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        c[i][j] =
          (c[i][j] + ((a[i][k] * b[k][j]) % MOD)) % MOD;
      }
    }
  }

  return c;
}
```

```cpp
template <typename T>
vector<vector<T>> fpow(vector<vector<T>> &xs, ll p) {
  vector<vector<T>> ans(len(xs), vector<T>(len(xs)));
  for (int i = 0; i < len(xs); i++) ans[i][i] = 1;

  auto b = xs;
  while (p) {
    if (p & 1) ans = prod(ans, b);
    p >>= 1;
    b = prod(b, b);
  }
  return ans;
}
```

5.24 N Choose K (elements)

process every possible combination of K elements from N elements, thoose index marked as 1 in the index vector says which elments are choosed at that moment.
Time : O($\binom{N}{K} \cdot O(process)$)

```cpp
void process(vi &index) {
  for (int i = 0; i < len(index); i++) {
    if (index[i]) cout << i << " \n"[i == len(index) - 1];
  }
}

void n_choose_k(int n, in k) {
  vi index(n);
  fill(index.end() - k, index.end(), 1);

  do {
    process(index);
  } while (next_permutation(all(index)));
}
```

5.25 NTT integer convolution and exponentiation

Convolution finds the product a and b, and exp finds a^k
time: convolution $O(N \cdot \log N)$, exponentiation: $O(\log K \cdot N \cdot \log N)$

```cpp
/*
    ================================================================
    */
/*===================== MINT ===================== */
template <int _mod>
struct mint {
  ll expo(ll b, ll e) {
```

```
  ll ret = 1;
  while (e) {
    if (e % 2) ret = ret * b % _mod;
    e /= 2, b = b * b % _mod;
  }
  return ret;
}
ll inv(ll b) { return expo(b, _mod - 2); }

using m = mint;
ll v;
mint() : v(0) {}
mint(ll v_) {
  if (v_ >= _mod or v_ <= -_mod) v_ %= _mod;
  if (v_ < 0) v_ += _mod;
  v = v_;
}
m& operator+=(const m& a) {
  v += a.v;
  if (v >= _mod) v -= _mod;
  return *this;
}
m& operator-=(const m& a) {
  v -= a.v;
  if (v < 0) v += _mod;
  return *this;
}
m& operator*=(const m& a) {
  v = v * ll(a.v) % _mod;
  return *this;
}
m& operator/=(const m& a) {
  v = v * inv(a.v) % _mod;
  return *this;
}
m operator-() { return m(-v); }
m& operator^=(ll e) {
  if (e < 0) {
    v = inv(v);
    e = -e;
  }
  v = expo(v, e);
  // possivel otimizacao:
  // cuidado com 0^0
  // v = expo(v, e%(p-1));
    return *this;
  }
  bool operator==(const m& a) { return v == a.v; }
  bool operator!=(const m& a) { return v != a.v; }

  friend istream& operator>>(istream& in, m& a) {
    ll val;
    in >> val;
    a = m(val);
    return in;
  }
  friend ostream& operator<<(ostream& out, m a) {
    return out << a.v;
  }
  friend m operator+(m a, m b) { return a += b; }
  friend m operator-(m a, m b) { return a -= b; }
  friend m operator*(m a, m b) { return a *= b; }
  friend m operator/(m a, m b) { return a /= b; }
  friend m operator^(m a, ll e) { return a ^= e; }
};
/*=================== END MINT ===================== */

/*================= ntt int convolution =============*/
const ll MOD1 = 998244353;
const ll MOD2 = 754974721;
const ll MOD3 = 167772161;

template <int _mod>
void ntt(vector<mint<_mod>>& a, bool rev) {
  int n = len(a);
  auto b = a;
  assert(!(n & (n - 1)));
  mint<_mod> g = 1;
  while ((g ^ (_mod / 2)) == 1) g += 1;
  if (rev) g = 1 / g;

  for (int step = n / 2; step; step /= 2) {
    mint<_mod> w = g ^ (_mod / (n / step)), wn = 1;
    for (int i = 0; i < n / 2; i += step) {
      for (int j = 0; j < step; j++) {
        auto u = a[2 * i + j], v = wn * a[2 * i + j + step];
        b[i + j] = u + v;
        b[i + n / 2 + j] = u - v;
      }
      wn = wn * w;
```

```
    }
    swap(a, b);
  }
  if (rev) {
    auto n1 = mint<_mod>(1) / n;
    for (auto& x : a) x *= n1;
  }
}

template <ll _mod>
vector<mint<_mod>> convolution(
  const vector<mint<_mod>>& a,
  const vector<mint<_mod>>& b) {
  vector<mint<_mod>> l(all(a)), r(all(b));
  int N = len(l) + len(r) - 1, n = 1;
  while (n <= N) n *= 2;
  l.resize(n), r.resize(n);
  ntt(l, false), ntt(r, false);
  for (int i = 0; i < n; i++) l[i] *= r[i];
  ntt(l, true);
  l.resize(N);

  // Uncommnent for a boolean convolution :)
  /*

  for (auto& li : l) {
    li.v = min(li.v, 1ll);
  }
  */

  return l;
}

template <ll _mod>
vector<mint<_mod>> poly_exp(vector<mint<_mod>>& ps, int k) {
  vector<mint<_mod>> ret(len(ps));
  auto base = ps;
  ret[0] = 1;

  while (k) {
    if (k & 1) ret = convolution(ret, base);
    k >>= 1;
    base = convolution(base, base);
  }
  return ret;
}
```

5.26 NTT Integer Convolution (combine 2 modules)

Computes the convolution between polynomials (vectors) a and b
This is pure magic !
time: $O(N \log N)$

```
/*
    ==================================================================
    */
/*====================== MINT ====================== */
template <int _mod>
struct mint {
  ll expo(ll b, ll e) {
    ll ret = 1;
    while (e) {
      if (e % 2) ret = ret * b % _mod;
      e /= 2, b = b * b % _mod;
    }
    return ret;
  }
  ll inv(ll b) { return expo(b, _mod - 2); }

  using m = mint;
  ll v;
  mint() : v(0) {}
  mint(ll v_) {
    if (v_ >= _mod or v_ <= -_mod) v_ %= _mod;
    if (v_ < 0) v_ += _mod;
    v = v_;
  }
  m& operator+=(const m& a) {
    v += a.v;
    if (v >= _mod) v -= _mod;
    return *this;
  }
  m& operator-=(const m& a) {
    v -= a.v;
    if (v < 0) v += _mod;
    return *this;
  }
  m& operator*=(const m& a) {
    v = v * ll(a.v) % _mod;
    return *this;
```

```
  }
  m& operator/=(const m& a) {
    v = v * inv(a.v) % _mod;
    return *this;
  }
  m operator-() { return m(-v); }
  m& operator^=(ll e) {
    if (e < 0) {
      v = inv(v);
      e = -e;
    }
    v = expo(v, e);
    // possivel otimizacao:
    // cuidado com 0^0
    // v = expo(v, e%(p-1));
    return *this;
  }
  bool operator==(const m& a) { return v == a.v; }
  bool operator!=(const m& a) { return v != a.v; }

  friend istream& operator>>(istream& in, m& a) {
    ll val;
    in >> val;
    a = m(val);
    return in;
  }
  friend ostream& operator<<(ostream& out, m a) {
    return out << a.v;
  }
  friend m operator+(m a, m b) { return a += b; }
  friend m operator-(m a, m b) { return a -= b; }
  friend m operator*(m a, m b) { return a *= b; }
  friend m operator/(m a, m b) { return a /= b; }
  friend m operator^(m a, ll e) { return a ^= e; }
};
/*==================== END MINT ====================== */

/*================== ntt int convolution ==============*/
const ll MOD1 = 998244353;
const ll MOD2 = 754974721;
const ll MOD3 = 167772161;

template <int _mod>
void ntt(vector<mint<_mod>>& a, bool rev) {
  int n = len(a);
  auto b = a;
  assert(!(n & (n - 1)));
  mint<_mod> g = 1;
  while ((g ^ (_mod / 2)) == 1) g += 1;
  if (rev) g = 1 / g;

  for (int step = n / 2; step; step /= 2) {
    mint<_mod> w = g ^ (_mod / (n / step)), wn = 1;
    for (int i = 0; i < n / 2; i += step) {
      for (int j = 0; j < step; j++) {
        auto u = a[2 * i + j], v = wn * a[2 * i + j + step];
        b[i + j] = u + v;
        b[i + n / 2 + j] = u - v;
      }
      wn = wn * w;
    }
    swap(a, b);
  }
  if (rev) {
    auto n1 = mint<_mod>(1) / n;
    for (auto& x : a) x *= n1;
  }
}

tuple<ll, ll, ll> ext_gcd(ll a, ll b) {
  if (!a) return {b, 0, 1};
  auto [g, x, y] = ext_gcd(b % a, a);
  return {g, y - b / a * x, x};
}

template <typename T = ll>
struct crt {
  T a, m;

  crt() : a(0), m(1) {}
  crt(T a_, T m_) : a(a_), m(m_) {}
  crt operator*(crt C) {
    auto [g, x, y] = ext_gcd(m, C.m);
    if ((a - C.a) % g != 0) a = -1;
    if (a == -1 or C.a == -1) return crt(-1, 0);
    T lcm = m / g * C.m;
    T ans = a + (x * (C.a - a) / g % (C.m / g)) * m;
    return crt((ans % lcm + lcm) % lcm, lcm);
  }
};
```

```cpp
template <typename T = ll>
struct Congruence {
  T a, m;
};

template <typename T = ll>
T chinese_remainder_theorem(
  const vector<Congruence<T>>& equations) {
  crt<T> ans;

  for (auto& [a_, m_] : equations) {
    ans = ans * crt<T>(a_, m_);
  }

  return ans.a;
}

#define int long long
template <ll m1, ll m2>
vll merge_two_mods(const vector<mint<m1>>& a,
                   const vector<mint<m2>>& b) {
  int n = len(a);
  vll ans(n);
  for (int i = 0; i < n; i++) {
    auto cur = crt<ll>();
    auto ai = a[i].v;
    auto bi = b[i].v;
    cur = cur * crt<ll>(ai, m1);
    cur = cur * crt<ll>(bi, m2);
    ans[i] = cur.a;
  }

  return ans;
}

vll convolution_2mods(const vll& a, const vll& b) {
  vector<mint<MOD1>> l(all(a)), r(all(b));
  int N = len(l) + len(r) - 1, n = 1;
  while (n <= N) n *= 2;
  l.resize(n), r.resize(n);
  ntt(l, false), ntt(r, false);
  for (int i = 0; i < n; i++) l[i] *= r[i];
  ntt(l, true);
  l.resize(N);
  vector<mint<MOD2>> l2(all(a)), r2(all(b));
  l2.resize(n), r2.resize(n);
  ntt(l2, false), ntt(r2, false);
  rep(i, 0, n) l2[i] *= r2[i];
  ntt(l2, true);
  l2.resize(N);

  return merge_two_mods(l, l2);
}

vll poly_exp(const vll& xs, ll k) {
  vll ret(len(xs));
  ret[0] = 1;
  auto base = xs;
  while (k) {
    if (k & 1) ret = convolution_2mods(ret, base);
    k >>= 1;
    base = convolution_2mods(base, base);
  }

  return ret;
}
```

5.27 Number Of Divisors (sieve)

```cpp
ll divisors(ll n) {
  ll ans = 1;
  for (auto p : primes) {
    if (p * p * p > n) break;

    int count = 1;
    while (n % p == 0) {
      n /= p;
      count++;
    }

    ans *= count;
  }

  if (is_prime[n])
    ans *= 2;
  else if (is_prime_square[n])
    ans *= 3;
  else if (n != 1)
```

```
    ans *= 4;

  return ans;
}
```

5.28 Number of Divisors $\tau(n)$

Find the total of divisors of N in $O(\sqrt{N})$

```
ll number_of_divisors(ll n) {
  ll res = 0;

  for (ll d = 1; d * d <= n; ++d) {
    if (n % d == 0) res += (d == n / d ? 1 : 2);
  }

  return res;
}
```

5.29 Power Sum

Calculates $K^0 + K^1 + ... + K^n$

```
ll powersum(ll n, ll k) {
  return (fastpow(n, k + 1) - 1) / (n - 1);
}
```

5.30 Sieve list primes

List every prime until MAXN, $O(N \log N)$ in time and $O(MAXN)$ in memory.

```
const ll MAXN = 2e5;
vll list_primes(ll n = MAXN) {
  vll ps;
  bitset<MAXN + 1> sieve;
  sieve.set();
  sieve.reset(1);
  for (ll i = 2; i <= n; ++i) {
    if (sieve[i]) ps.push_back(i);
    for (ll j = i * 2; j <= n; j += i) {
      sieve.reset(j);
    }
  }
  return ps;
}
```

5.31 Sum of Divisors $\sigma(n)$

Computes the sum of each divisor of n in $O(\sqrt{n})$.

```
ll sum_of_divisors(long long n) {
  ll res = 0;

  for (ll d = 1; d * d <= n; ++d) {
    if (n % d == 0) {
      ll k = n / d;

      res += (d == k ? d : d + k);
    }
  }

  return res;
}
```

5.32 To Any Base

```
vll to_otherbase(ll x, ll b) {
  vll result;

  while (x) {
    auto [quot, rem] = std::div(x, b);

    x = rem < 0 ? quot + 1 : quot;
    rem = rem < 0 ? rem + -b : rem;

    result.eb(rem);
  }

  if (!len(result)) return {0ll};

  reverse(all(result));

  // [msb, ..., lsb]
  return result;
}
```

6 Primitives

6.1 Bigint

```
const int maxn = 1e2 + 14, lg = 15;
const int base = 1000000000;
```

```cpp
const int base_digits = 9;
struct bigint {
  vi a;
  int sign;

  int size() {
    if (a.empty()) return 0;
    int ans = (a.size() - 1) * base_digits;
    int ca = a.back();
    while (ca) ans++, ca /= 10;
    return ans;
  }
  bigint operator^(const bigint &v) {
    bigint ans = 1, a = *this, b = v;
    while (!b.isZero()) {
      if (b % 2) ans *= a;
      a *= a, b /= 2;
    }
    return ans;
  }
  string to_string() {
    stringstream ss;
    ss << *this;
    string s;
    ss >> s;
    return s;
  }
  int sumof() {
    string s = to_string();
    int ans = 0;
    for (auto c : s) ans += c - '0';
    return ans;
  }
  /*</arpa>*/
  bigint() : sign(1) {}

  bigint(long long v) { *this = v; }

  bigint(const string &s) { read(s); }

  void operator=(const bigint &v) {
    sign = v.sign;
    a = v.a;
  }

  void operator=(long long v) {
    sign = 1;
    a.clear();
    if (v < 0) sign = -1, v = -v;
    for (; v > 0; v = v / base) a.push_back(v % base);
  }

  bigint operator+(const bigint &v) const {
    if (sign == v.sign) {
      bigint res = v;

      for (int i = 0, carry = 0;
           i < (int)max(a.size(), v.a.size()) || carry;
           ++i) {
        if (i == (int)res.a.size()) res.a.push_back(0);
        res.a[i] += carry + (i < (int)a.size() ? a[i] : 0);
        carry = res.a[i] >= base;
        if (carry) res.a[i] -= base;
      }
      return res;
    }
    return *this - (-v);
  }

  bigint operator-(const bigint &v) const {
    if (sign == v.sign) {
      if (abs() >= v.abs()) {
        bigint res = *this;
        for (int i = 0, carry = 0;
             i < (int)v.a.size() || carry; ++i) {
          res.a[i] -=
            carry + (i < (int)v.a.size() ? v.a[i] : 0);
          carry = res.a[i] < 0;
          if (carry) res.a[i] += base;
        }
        res.trim();
        return res;
      }
      return -(v - *this);
    }
    return *this + (-v);
  }

  void operator*=(int v) {
    if (v < 0) sign = -sign, v = -v;
```

```cpp
  for (int i = 0, carry = 0; i < (int)a.size() || carry;
       ++i) {
    if (i == (int)a.size()) a.push_back(0);
    long long cur = a[i] * (long long)v + carry;
    carry = (int)(cur / base);
    a[i] = (int)(cur % base);
    // asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) :
    // "A"(cur), "c"(base));
  }
  trim();
}

bigint operator*(int v) const {
  bigint res = *this;
  res *= v;
  return res;
}

void operator*=(long long v) {
  if (v < 0) sign = -sign, v = -v;
  if (v > base) {
    *this =
      *this * (v / base) * base + *this * (v % base);
    return;
  }
  for (int i = 0, carry = 0; i < (int)a.size() || carry;
       ++i) {
    if (i == (int)a.size()) a.push_back(0);
    long long cur = a[i] * (long long)v + carry;
    carry = (int)(cur / base);
    a[i] = (int)(cur % base);
    // asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) :
    // "A"(cur), "c"(base));
  }
  trim();
}

bigint operator*(long long v) const {
  bigint res = *this;
  res *= v;
  return res;
}

friend pair<bigint, bigint> divmod(const bigint &a1,
                                   const bigint &b1) {
  int norm = base / (b1.a.back() + 1);
  bigint a = a1.abs() * norm;
  bigint b = b1.abs() * norm;
  bigint q, r;
  q.a.resize(a.a.size());

  for (int i = a.a.size() - 1; i >= 0; i--) {
    r *= base;
    r += a.a[i];
    int s1 =
      r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
    int s2 = r.a.size() <= b.a.size() - 1
               ? 0
               : r.a[b.a.size() - 1];
    int d = ((long long)base * s1 + s2) / b.a.back();
    r -= b * d;
    while (r < 0) r += b, --d;
    q.a[i] = d;
  }

  q.sign = a1.sign * b1.sign;
  r.sign = a1.sign;
  q.trim();
  r.trim();
  return make_pair(q, r / norm);
}

bigint operator/(const bigint &v) const {
  return divmod(*this, v).first;
}

bigint operator%(const bigint &v) const {
  return divmod(*this, v).second;
}

void operator/=(int v) {
  if (v < 0) sign = -sign, v = -v;
  for (int i = (int)a.size() - 1, rem = 0; i >= 0; --i) {
    long long cur = a[i] + rem * (long long)base;
    a[i] = (int)(cur / v);
    rem = (int)(cur % v);
  }
  trim();
}
```

```cpp
bigint operator/(int v) const {
  bigint res = *this;
  res /= v;
  return res;
}

int operator%(int v) const {
  if (v < 0) v = -v;
  int m = 0;
  for (int i = a.size() - 1; i >= 0; --i)
    m = (a[i] + m * (long long)base) % v;
  return m * sign;
}

void operator+=(const bigint &v) { *this = *this + v; }
void operator-=(const bigint &v) { *this = *this - v; }
void operator*=(const bigint &v) { *this = *this * v; }
void operator/=(const bigint &v) { *this = *this / v; }

bool operator<(const bigint &v) const {
  if (sign != v.sign) return sign < v.sign;
  if (a.size() != v.a.size())
    return a.size() * sign < v.a.size() * v.sign;
  for (int i = a.size() - 1; i >= 0; i--)
    if (a[i] != v.a[i])
      return a[i] * sign < v.a[i] * sign;
  return false;
}

bool operator>(const bigint &v) const {
  return v < *this;
}
bool operator<=(const bigint &v) const {
  return !(v < *this);
}
bool operator>=(const bigint &v) const {
  return !(*this < v);
}
bool operator==(const bigint &v) const {
  return !(*this < v) && !(v < *this);
}
bool operator!=(const bigint &v) const {
  return *this < v || v < *this;
}
```

```cpp
void trim() {
  while (!a.empty() && !a.back()) a.pop_back();
  if (a.empty()) sign = 1;
}

bool isZero() const {
  return a.empty() || (a.size() == 1 && !a[0]);
}

bigint operator-() const {
  bigint res = *this;
  res.sign = -sign;
  return res;
}

bigint abs() const {
  bigint res = *this;
  res.sign *= res.sign;
  return res;
}

long long longValue() const {
  long long res = 0;
  for (int i = a.size() - 1; i >= 0; i--)
    res = res * base + a[i];
  return res * sign;
}

friend bigint gcd(const bigint &a, const bigint &b) {
  return b.isZero() ? a : gcd(b, a % b);
}
friend bigint lcm(const bigint &a, const bigint &b) {
  return a / gcd(a, b) * b;
}

void read(const string &s) {
  sign = 1;
  a.clear();
  int pos = 0;
  while (pos < (int)s.size() &&
         (s[pos] == '-' || s[pos] == '+')) {
    if (s[pos] == '-') sign = -sign;
    ++pos;
  }
  for (int i = s.size() - 1; i >= pos; i -= base_digits) {
```

```
    int x = 0;
    for (int j = max(pos, i - base_digits + 1); j <= i;
         j++)
      x = x * 10 + s[j] - '0';
    a.push_back(x);
  }
  trim();
}

friend istream &operator>>(istream &stream, bigint &v) {
  string s;
  stream >> s;
  v.read(s);
  return stream;
}

friend ostream &operator<<(ostream &stream,
                           const bigint &v) {
  if (v.sign == -1) stream << '-';
  stream << (v.a.empty() ? 0 : v.a.back());
  for (int i = (int)v.a.size() - 2; i >= 0; --i)
    stream << setw(base_digits) << setfill('0') << v.a[i];
  return stream;
}

static vector<int> convert_base(const vector<int> &a,
                                int old_digits,
                                int new_digits) {
  vector<long long> p(max(old_digits, new_digits) + 1);
  p[0] = 1;
  for (int i = 1; i < (int)p.size(); i++)
    p[i] = p[i - 1] * 10;
  vector<int> res;
  long long cur = 0;
  int cur_digits = 0;
  for (int i = 0; i < (int)a.size(); i++) {
    cur += a[i] * p[cur_digits];
    cur_digits += old_digits;
    while (cur_digits >= new_digits) {
      res.push_back(int(cur % p[new_digits]));
      cur /= p[new_digits];
      cur_digits -= new_digits;
    }
  }
  res.push_back((int)cur);
  while (!res.empty() && !res.back()) res.pop_back();
  return res;
}

typedef vector<long long> vll;

static vll karatsubaMultiply(const vll &a, const vll &b) {
  int n = a.size();
  vll res(n + n);
  if (n <= 32) {
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        res[i + j] += a[i] * b[j];
    return res;
  }

  int k = n >> 1;
  vll a1(a.begin(), a.begin() + k);
  vll a2(a.begin() + k, a.end());
  vll b1(b.begin(), b.begin() + k);
  vll b2(b.begin() + k, b.end());

  vll a1b1 = karatsubaMultiply(a1, b1);
  vll a2b2 = karatsubaMultiply(a2, b2);

  for (int i = 0; i < k; i++) a2[i] += a1[i];
  for (int i = 0; i < k; i++) b2[i] += b1[i];

  vll r = karatsubaMultiply(a2, b2);
  for (int i = 0; i < (int)a1b1.size(); i++)
    r[i] -= a1b1[i];
  for (int i = 0; i < (int)a2b2.size(); i++)
    r[i] -= a2b2[i];

  for (int i = 0; i < (int)r.size(); i++)
    res[i + k] += r[i];
  for (int i = 0; i < (int)a1b1.size(); i++)
    res[i] += a1b1[i];
  for (int i = 0; i < (int)a2b2.size(); i++)
    res[i + n] += a2b2[i];
  return res;
}

bigint operator*(const bigint &v) const {
  vector<int> a6 = convert_base(this->a, base_digits, 6);
```

```
    vector<int> b6 = convert_base(v.a, base_digits, 6);
    vll a(a6.begin(), a6.end());
    vll b(b6.begin(), b6.end());
    while (a.size() < b.size()) a.push_back(0);
    while (b.size() < a.size()) b.push_back(0);
    while (a.size() & (a.size() - 1))
      a.push_back(0), b.push_back(0);
    vll c = karatsubaMultiply(a, b);
    bigint res;
    res.sign = sign * v.sign;
    for (int i = 0, carry = 0; i < (int)c.size(); i++) {
      long long cur = c[i] + carry;
      res.a.push_back((int)(cur % 1000000));
      carry = (int)(cur / 1000000);
    }
    res.a = convert_base(res.a, 6, base_digits);
    res.trim();
    return res;
  }
};
```

6.2 Integer Mod

```
const ll MOD = 1'000'000'000 + 7;
template <ll _mod = MOD>
struct mint {
  ll value;
  static const ll MOD_value = _mod;

  mint(ll v = 0) {
    value = v % _mod;
    if (value < 0) value += _mod;
  }
  mint(ll a, ll b) : value(0) {
    *this += a;
    *this /= b;
  }

  mint& operator+=(mint const& b) {
    value += b.value;
    if (value >= _mod) value -= _mod;
    return *this;
  }
  mint& operator-=(mint const& b) {
    value -= b.value;
    if (value < 0) value += _mod;
    return *this;
  }
  mint& operator*=(mint const& b) {
    value = (ll)value * b.value % _mod;
    return *this;
  }

  friend mint mexp(mint a, ll e) {
    mint res = 1;
    while (e) {
      if (e & 1) res *= a;
      a *= a;
      e >>= 1;
    }
    return res;
  }
  friend mint inverse(mint a) { return mexp(a, _mod - 2); }

  mint& operator/=(mint const& b) {
    return *this *= inverse(b);
  }
  friend mint operator+(mint a, mint const b) {
    return a += b;
  }
  mint operator++(int) {
    return this->value = (this->value + 1) % _mod;
  }
  mint operator++() {
    return this->value = (this->value + 1) % _mod;
  }
  friend mint operator-(mint a, mint const b) {
    return a -= b;
  }
  friend mint operator-(mint const a) { return 0 - a; }
  mint operator--(int) {
    return this->value = (this->value - 1 + _mod) % _mod;
  }

  mint operator--() {
    return this->value = (this->value - 1 + _mod) % _mod;
  }
  friend mint operator*(mint a, mint const b) {
    return a *= b;
  }
```

```
  friend mint operator/(mint a, mint const b) {
    return a /= b;
  }
  friend std::ostream& operator<<(std::ostream& os,
                                  mint const& a) {
    return os << a.value;
  }
  friend bool operator==(mint const& a, mint const& b) {
    return a.value == b.value;
  }
  friend bool operator!=(mint const& a, mint const& b) {
    return a.value != b.value;
  }
};
```

6.3 Matrix

```
template <typename T>
struct Matrix {
  vector<vector<T>> d;

  Matrix() : Matrix(0) {}
  Matrix(int n) : Matrix(n, n) {}
  Matrix(int n, int m)
    : Matrix(vector<vector<T>>(n, vector<T>(m))) {}
  Matrix(const vector<vector<T>> &v) : d(v) {}

  constexpr int n() const { return (int)d.size(); }
  constexpr int m() const {
    return n() ? (int)d[0].size() : 0;
  }

  void rotate() { *this = rotated(); }

  Matrix<T> rotated() const {
    Matrix<T> res(m(), n());
    for (int i = 0; i < m(); i++) {
      for (int j = 0; j < n(); j++) {
        res[i][j] = d[n() - j - 1][i];
      }
    }
    return res;
  }

  Matrix<T> pow(int power) const {
    assert(n() == m());

    auto res = Matrix<T>::identity(n());
    auto b = *this;
    while (power) {
      if (power & 1) res *= b;
      b *= b;
      power >>= 1;
    }
    return res;
  }

  Matrix<T> submatrix(int start_i, int start_j,
                      int rows = INT_MAX,
                      int cols = INT_MAX) const {
    rows = min(rows, n() - start_i);
    cols = min(cols, m() - start_j);
    if (rows <= 0 or cols <= 0) return {};

    Matrix<T> res(rows, cols);
    for (int i = 0; i < rows; i++)
      for (int j = 0; j < cols; j++)
        res[i][j] = d[i + start_i][j + start_j];
    return res;
  }

  Matrix<T> translated(int x, int y) const {
    Matrix<T> res(n(), m());
    for (int i = 0; i < n(); i++) {
      for (int j = 0; j < m(); j++) {
        if (i + x < 0 or i + x >= n() or j + y < 0 or
            j + y >= m())
          continue;
        res[i + x][j + y] = d[i][j];
      }
    }
    return res;
  }

  static Matrix<T> identity(int n) {
    Matrix<T> res(n);
    for (int i = 0; i < n; i++) res[i][i] = 1;
    return res;
  }
```

```
vector<T> &operator[](int i) { return d[i]; }
const vector<T> &operator[](int i) const { return d[i]; }
Matrix<T> &operator+=(T value) {
  for (auto &row : d) {
    for (auto &x : row) x += value;
  }
  return *this;
}
Matrix<T> operator+(T value) const {
  auto res = *this;
  for (auto &row : res) {
    for (auto &x : row) x = x + value;
  }
  return res;
}
Matrix<T> &operator-=(T value) {
  for (auto &row : d) {
    for (auto &x : row) x -= value;
  }
  return *this;
}
Matrix<T> operator-(T value) const {
  auto res = *this;
  for (auto &row : res) {
    for (auto &x : row) x = x - value;
  }
  return res;
}
Matrix<T> &operator*=(T value) {
  for (auto &row : d) {
    for (auto &x : row) x *= value;
  }
  return *this;
}
Matrix<T> operator*(T value) const {
  auto res = *this;
  for (auto &row : res) {
    for (auto &x : row) x = x * value;
  }
  return res;
}
Matrix<T> &operator/=(T value) {
  for (auto &row : d) {
    for (auto &x : row) x /= value;
  }
  return *this;
}
Matrix<T> operator/(T value) const {
  auto res = *this;
  for (auto &row : res) {
    for (auto &x : row) x = x / value;
  }
  return res;
}
Matrix<T> &operator+=(const Matrix<T> &o) {
  assert(n() == o.n() and m() == o.m());
  for (int i = 0; i < n(); i++) {
    for (int j = 0; j < m(); j++) {
      d[i][j] += o[i][j];
    }
  }
  return *this;
}
Matrix<T> operator+(const Matrix<T> &o) const {
  assert(n() == o.n() and m() == o.m());
  auto res = *this;
  for (int i = 0; i < n(); i++) {
    for (int j = 0; j < m(); j++) {
      res[i][j] = res[i][j] + o[i][j];
    }
  }
  return res;
}
Matrix<T> &operator-=(const Matrix<T> &o) {
  assert(n() == o.n() and m() == o.m());
  for (int i = 0; i < n(); i++) {
    for (int j = 0; j < m(); j++) {
      d[i][j] -= o[i][j];
    }
  }
  return *this;
}
Matrix<T> operator-(const Matrix<T> &o) const {
  assert(n() == o.n() and m() == o.m());
  auto res = *this;
  for (int i = 0; i < n(); i++) {
    for (int j = 0; j < m(); j++) {
      res[i][j] = res[i][j] - o[i][j];
    }
  }
```

```
  return res;
}
Matrix<T> &operator*=(const Matrix<T> &o) {
  *this = *this * o;
  return *this;
}
Matrix<T> operator*(const Matrix<T> &o) const {
  assert(m() == o.n());
  Matrix<T> res(n(), o.m());
  for (int i = 0; i < res.n(); i++) {
    for (int j = 0; j < res.m(); j++) {
      auto &x = res[i][j];
      for (int k = 0; k < m(); k++) {
        x += (d[i][k] * o[k][j]);
      }
    }
  }
  return res;
}

friend istream &operator>>(istream &is, Matrix<T> &mat) {
  for (auto &row : mat)
    for (auto &x : row) is >> x;
  return is;
}
friend ostream &operator<<(ostream &os,
                           const Matrix<T> &mat) {
  bool frow = 1;
  for (auto &row : mat) {
    if (not frow) os << '\n';
    bool first = 1;
    for (auto &x : row) {
      if (not first) os << ' ';
      os << x;
      first = 0;
    }

    frow = 0;
  }
  return os;
}

auto begin() { return d.begin(); }
auto end() { return d.end(); }
auto rbegin() { return d.rbegin(); }
auto rend() { return d.rend(); }

auto begin() const { return d.begin(); }
auto end() const { return d.end(); }
auto rbegin() const { return d.rbegin(); }
auto rend() const { return d.rend(); }
};
```

7 Problems

7.1 Hanoi Tower

Let T_n be the total of moves to solve a hanoi tower, we know that $T_n >= 2 \cdot T_{n-1} + 1$, for $n > 0$, and $T_0 = 0$. By induction it's easy to see that $T_n = 2^n - 1$, for $n > 0$.
The following algorithm finds the necessary steps to solve the game for 3 stacks and n disks.

```
void move(int a, int b) { cout << a << ' ' << b << endl; }
void solve(int n, int s, int e) {
  if (n == 0) return;
  if (n == 1) {
    move(s, e);
    return;
  }
  solve(n - 1, s, 6 - s - e);
  move(s, e);
  solve(n - 1, 6 - s - e, e);
}
```

8 Searching

8.1 Meet in the middle

Answers the query how many subsets of the vector xs have sum equal x.
Time: $O(N \cdot 2^{\frac{N}{2}})$

```
vll get_subset_sums(int l, int r, vll &a) {
  int len = r - l + 1;
  vll res;

  for (int i = 0; i < (1 << len); i++) {
    ll sum = 0;
    for (int j = 0; j < len; j++) {
      if (i & (1 << j)) {
        sum += a[l + j];
      }
    }
    res.push_back(sum);
```

```
  }
  return res;
};

ll count(vll &xs, ll x) {
  int n = len(xs);
  vll left = get_subset_sums(0, n / 2 - 1, xs);
  vll right = get_subset_sums(n / 2, n - 1, xs);
  sort(all(left));
  sort(all(right));
  ll ans = 0;
  for (ll i : left) {
    auto start_index =
      lower_bound(right.begin(), right.end(), x - i) -
      right.begin();
    auto end_index =
      upper_bound(right.begin(), right.end(), x - i) -
      right.begin();
    ans += end_index - start_index;
  }
  return ans;
}
```

8.2 Ternary Search Recursive

```
const double eps = 1e-6;

// IT MUST BE AN UNIMODAL FUNCTION
double f(int x) { return x * x + 2 * x + 4; }

double ternary_search(double l, double r) {
  if (fabs(f(l) - f(r)) < eps)
    return f((l + (r - l) / 2.0));

  auto third = (r - l) / 3.0;
  auto m1 = l + third;
  auto m2 = r - third;

  // change the signal to find the maximum point.
  return m1 < m2 ? ternary_search(m1, r)
                 : ternary_search(l, m2);
}
```

9 Strings

9.1 Count Distinct Anagrams

```
const ll MOD = 1e9 + 7;
const int maxn = 1e6;
vll fs(maxn + 1);
void precompute() {
  fs[0] = 1;
  for (ll i = 1; i <= maxn; i++) {
    fs[i] = (fs[i - 1] * i) % MOD;
  }
}

ll fpow(ll a, int n, ll mod = LLONG_MAX) {
  if (n == 0) return 1;
  if (n == 1) return a;
  ll x = fpow(a, n / 2, mod) % mod;
  return ((x * x) % mod * (n & 1 ? a : 1ll)) % mod;
}

ll distinctAnagrams(const string &s) {
  precompute();
  vi hist('z' - 'a' + 1, 0);
  for (auto &c : s) hist[c - 'a']++;
  ll ans = fs[len(s)];
  for (auto &q : hist) {
    ans = (ans * fpow(fs[q], MOD - 2, MOD)) % MOD;
  }
  return ans;
}
```

9.2 Double Hash Range Query

```
using ll = long long;
using vll = vector<ll>;
using pll = pair<ll, ll>;

const int MAXN(1'000'000);
const ll MOD = 1000027957;
const ll MOD2 = 1000015187;
const ll P = 31;

ll p[MAXN + 1], p2[MAXN + 1];
void precompute() {
  p[0] = p2[0] = 1;
```

```
  for (int i = 1; i <= MAXN; i++)
    p[i] = (P * p[i - 1]) % MOD,
    p2[i] = (P * p2[i - 1]) % MOD2;
}

struct Hash {
  int n;
  vll h, h2, hi, hi2;
  Hash() {}
  Hash(const string& s)
    : n(s.size()), h(n), h2(n), hi(n), hi2(n) {
    h[0] = h2[0] = s[0];
    for (int i = 1; i < n; i++)
      h[i] = (s[i] + h[i - 1] * P) % MOD,
      h2[i] = (s[i] + h2[i - 1] * P) % MOD2;

    hi[n - 1] = hi2[n - 1] = s[n - 1];
    for (int i = n - 2; i >= 0; i--)
      hi[i] = (s[i] + hi[i + 1] * P) % MOD,
      hi2[i] = (s[i] + hi2[i + 1] * P) % MOD2;
  }
  pll query(int l, int r) {
    ll hash =
      (h[r] - (l ? h[l - 1] * p[r - l + 1] % MOD : 0));
    ll hash2 =
      (h2[r] - (l ? h2[l - 1] * p2[r - l + 1] % MOD2 : 0));

    return {(hash < 0 ? hash + MOD : hash),
            (hash2 < 0 ? hash2 + MOD2 : hash2)};
  }
  pll query_inv(int l, int r) {
    ll hash =
      (hi[l] -
       (r + 1 < n ? hi[r + 1] * p[r - l + 1] % MOD : 0));
    ll hash2 =
      (hi2[l] -
       (r + 1 < n ? hi2[r + 1] * p2[r - l + 1] % MOD2 : 0));
    return {(hash < 0 ? hash + MOD : hash),
            (hash2 < 0 ? hash2 + MOD2 : hash2)};
  }
};
```

9.3 Hash Interal mod $2^{64} - 1$

Arithmetic mod $2^{64} - 1$. 2x slower than mod 2^{64} and more code, but works on evil test data (e.g. Thue-Morse, where ABBA... and BAAB... of length 2^{10} hash the same mod 2^{64}).
"typedef ull H;" instead if you think test data is random.

```
typedef uint64_t ull;
struct H {
  ull x;
  H(ull x = 0) : x(x) {}
  H operator+(H o) { return x + o.x + (x + o.x < x); }
  H operator-(H o) { return *this + ~o.x; }
  H operator*(H o) {
    auto m = (__uint128_t)x * o.x;
    return H((ull)m) + (ull)(m >> 64);
  }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};
static const H C =
  (long long)1e11 + 3;  // (order ~ 3e9; random also ok)

struct Hash {
  int n;
  vector<H> ha, pw;
  Hash(string& str)
    : n(str.size()), ha((int)str.size() + 1), pw(ha) {
    pw[0] = 1;
    for (int i = 0; i < (int)str.size(); i++)
      ha[i + 1] = ha[i] * C + str[i], pw[i + 1] = pw[i] * C;
  }
  H query(int a, int b) {  // hash [a, b]
    b++;
    return ha[b] - ha[a] * pw[b - a];
  }
};

vector<H> getHashes(string& str, int length) {
  if ((int)str.size() < length) return {};
  H h = 0, pw = 1;
  for (int i = 0; i < length; i++)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  for (int i = length; i < (int)str.size(); i++)
    ret.push_back(h =
                  h * C + str[i] - pw * str[i - length]);
  return ret;
}
```

```cpp
H hashString(string& s) {
  H h{};
  for (char c : s) h = h * C + c;
  return h;
}
```

9.4 Hash Range Query

```cpp
const ll P = 31;
const ll MOD = 1e9 + 9;
const int MAXN(1e6);

ll ppow[MAXN + 1];
void pre_calc() {
  ppow[0] = 1;
  for (int i = 1; i <= MAXN; i++)
    ppow[i] = (ppow[i - 1] * P) % MOD;
}

struct Hash {
  int n;
  vll h, hi;
  Hash(const string &s) : n(s.size()), h(n), hi(n) {
    h[0] = s[0];
    hi[n - 1] = s[n - 1];
    for (int i = 1; i < n; i++) {
      h[i] = (s[i] + h[i - 1] * P) % MOD;
      hi[n - i - 1] =
        (s[n - i - 1] + hi[n - i - 1] * P) % MOD;
    }
  }

  ll qry(int l, int r) {
    ll hash =
      (h[r] - (l ? h[l - 1] * ppow[r - l + 1] % MOD : 0));
    return hash < 0 ? hash + MOD : hash;
  }

  ll qry_inv(int l, int r) {
    ll hash =
      (hi[l] -
       (r + 1 < n ? hi[r + 1] * ppow[r - l + 1] % MOD : 0));
    return hash < 0 ? hash + MOD : hash;
  }
};
```

9.5 Hash Ull

```cpp
using ull = unsigned long long;

const int MAXN(1'000'000);

const ull P = 31;
ull p[MAXN + 1];
void precompute() {
  p[0] = 1;
  for (int i = 1; i <= MAXN; i++) p[i] = (P * p[i - 1]);
}

struct Hash {
  int n;
  vector<ull> h;
  // vector<ull> hi;
  Hash() {}

  Hash(const string& s) : n(s.size()), h(n) /*, hi(n) */ {
    h[0] = s[0];
    for (int i = 1; i < n; i++)
      h[i] = (s[i] + h[i - 1] * P);

    // hi[n - 1] = s[n - 1];
    // for (int i = n - 2; i >= 0; i--)
    // ␣hi[i] = (s[i] + hi[i + 1] * P);
  }

  ull query(int l, int r) {
    ull hash = (h[r] - (l ? h[l - 1] * p[r - l + 1] : 0));
    return hash;
  }

  // ull query_inv(int l, int r) {
  // ␣ull hash = (hi[l] - (r + 1 < n ? hi[r + 1] * p[r - l +
  // 1] : 0)); ␣return hash;
  // }
};
```

9.6 K-th digit in digit string

Find the k-th digit in a *digit string*, only works for $1 <= k <= 10^{18}$!
Time: precompute $O(1)$, query $O(1)$

```cpp
using vull = vector<ull>;
vull pow10;
vector<array<ull, 4>> memo;
void precompute(int maxpow = 18) {
  ull qtd = 1;
  ull start = 1;
  ull end = 9;
  ull curlenght = 9;
  ull startstr = 1;
  ull endstr = 9;

  for (ull i = 0, j = 1ll; (int)i < maxpow; i++, j *= 10ll)
    pow10.eb(j);

  for (ull i = 0; i < maxpow - 1ull; i++) {
    memo.push_back({start, end, startstr, endstr});

    start = end + 1ll;
    end = end + (9ll * pow10[qtd]);
    curlenght = end - start + 1ull;

    qtd++;
    startstr = endstr + 1ull;
    endstr = (endstr + 1ull) + (curlenght)*qtd - 1ull;
  }
}
char kthDigit(ull k) {
  int qtd = 1;
  for (auto [s, e, ss, es] : memo) {
    if (k >= ss and k <= es) {
      ull pos = k - ss;
      ull index = pos / qtd;
      ull nmr = s + index;
      int i = k - ss - qtd * index;

      return ((nmr / pow10[qtd - i - 1]) % 10) + '0';
    }
    qtd++;
  }

  return 'X';
}
```

9.7 Longest Palindrome Substring (Manacher)

Finds the longest palindrome substring, manacher returns a vector where the i-th position is how much is possible to grow the string to the left and the right of i and keep it a palindrome.
Time: $O(N)$

```cpp
vi manacher(const string &s) {
  int n = len(s) - 2;
  vi p(n + 2);
  int l = 1, r = 1;
  for (int i = 1; i <= n; i++) {
    p[i] = max(0, min(r - i, p[l + (r - i)]));
    while (s[i - p[i]] == s[i + p[i]]) p[i]++;
    if (i + p[i] > r) l = i - p[i], r = i + p[i];
    p[i]--;
  }
  return p;
}
string longest_palindrome(const string &s) {
  string t("$#");
  for (auto c : s) t.push_back(c), t.push_back('#');
  t.push_back('^');
  vi xs = manacher(t);
  int mpos = max_element(all(xs)) - xs.begin();
  string p;
  for (int k = xs[mpos], i = mpos - k; i <= mpos + k; i++)
    if (t[i] != '#') p.push_back(t[i]);
  return p;
}
```

9.8 Longest Palindrome

```cpp
string longest_palindrome(const string &s) {
  int n = (int)s.size();
  vector<array<int, 2>> dp(n);

  pii odd(0, -1), even(0, -1);
  pii ans;
  for (int i = 0; i < n; i++) {
    int k = 0;
    if (i > odd.second)
      k = 1;
    else
      k = min(dp[odd.first + odd.second - i][0],
              odd.second - i + 1);
    while (i - k >= 0 and i + k < n and
```

```
          s[i - k] == s[i + k])
      k++;
    dp[i][0] = k--;
    if (i + k > odd.second) odd = {i - k, i + k};
    if (2 * dp[i][0] - 1 > ans.second)
      ans = {i - k, 2 * dp[i][0] - 1};

    k = 0;
    if (i <= even.second)
      k = min(dp[even.first + even.second - i + 1][1],
              even.second - i + 1);
    while (i - k - 1 >= 0 and i + k < n and
          s[i - k - 1] == s[i + k])
      k++;
    dp[i][1] = k--;
    if (i + k > even.second) even = {i - k - 1, i + k};
    if (2 * dp[i][1] > ans.second)
      ans = {i - k - 1, 2 * dp[i][1]};
  }
  return s.substr(ans.first, ans.second);
}
```

9.9 Rabin Karp

```
size_t rabin_karp(const string &s, const string &p) {
  if (s.size() < p.size()) return 0;

  auto n = s.size(), m = p.size();
  const ll p1 = 31, p2 = 29, q1 = 1e9 + 7, q2 = 1e9 + 9;
  const ll p1_1 = fpow(p1, q1 - 2, q1),
           p1_2 = fpow(p1, m - 1, q1);
  const ll p2_1 = fpow(p2, q2 - 2, q2),
           p2_2 = fpow(p2, m - 1, q2);

  pair<ll, ll> hs, hp;
  for (int i = (int)m - 1; ~i; --i) {
    hs.first = (hs.first * p1) % q1;
    hs.first = (hs.first + (s[i] - 'a' + 1)) % q1;
    hs.second = (hs.second * p2) % q2;
    hs.second = (hs.second + (s[i] - 'a' + 1)) % q2;

    hp.first = (hp.first * p1) % q1;
    hp.first = (hp.first + (p[i] - 'a' + 1)) % q1;
    hp.second = (hp.second * p2) % q2;
    hp.second = (hp.second + (p[i] - 'a' + 1)) % q2;
  }

  size_t occ = 0;
  for (size_t i = 0; i < n - m; i++) {
    occ += (hs == hp);

    int fi = s[i] - 'a' + 1;
    int fm = s[i + m] - 'a' + 1;

    hs.first = (hs.first - fi + q1) % q1;
    hs.first = (hs.first * p1_1) % q1;
    hs.first = (hs.first + fm * p1_2) % q1;
    hs.second = (hs.second - fi + q2) % q2;
    hs.second = (hs.second * p2_1) % q2;
    hs.second = (hs.second + fm * p2_2) % q2;
  }
  occ += hs == hp;

  return occ;
}
```

9.10 String Psum

```
struct strPsum {
  ll n;
  ll k;
  vector<vll> psum;
  strPsum(const string &s)
    : n(s.size()), k(100), psum(k, vll(n + 1)) {
    for (ll i = 1; i <= n; ++i) {
      for (ll j = 0; j < k; ++j) {
        psum[j][i] = psum[j][i - 1];
      }
      psum[s[i - 1]][i]++;
    }
  }

  ll qtd(ll l, ll r, char c) {  // [0,n-1]
    return psum[c][r + 1] - psum[c][l];
  }
}
```

9.11 Suffix Automaton (complete)

```
struct state {
  int len, link, cnt, firstpos;
```

```cpp
  // this can be optimized using a vector with the alphabet
  // size
  map<char, int> next;
  vi inv_link;
};
struct SuffixAutomaton {
  vector<state> st;
  int sz = 0;
  int last;
  vc cloned;

  SuffixAutomaton(const string &s, int maxlen)
    : st(maxlen * 2), cloned(maxlen * 2) {
    st[0].len = 0;
    st[0].link = -1;
    sz++;
    last = 0;
    for (auto &c : s) add_char(c);

    // precompute for count occurences
    for (int i = 1; i < sz; i++) {
      st[i].cnt = !cloned[i];
    }
    vector<pair<state, int>> aux;
    for (int i = 0; i < sz; i++) {
      aux.push_back({st[i], i});
    }

    sort(all(aux), [](const pair<state, int> &a,
                      const pair<state, int> &b) {
      return a.fst.len > b.fst.len;
    });

    for (auto &[stt, id] : aux) {
      if (stt.link != -1) {
        st[stt.link].cnt += st[id].cnt;
      }
    }

    // for find every occurende position
    for (int v = 1; v < sz; v++) {
      st[st[v].link].inv_link.push_back(v);
    }
  }

  void add_char(char c) {
    int cur = sz++;
    st[cur].len = st[last].len + 1;
    st[cur].firstpos = st[cur].len - 1;
    int p = last;
    // follow the suffix link until find a transition to c
    while (p != -1 and !st[p].next.count(c)) {
      st[p].next[c] = cur;
      p = st[p].link;
    }
    // there was no transition to c so create and leave
    if (p == -1) {
      st[cur].link = 0;
      last = cur;
      return;
    }

    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) {
      st[cur].link = q;
    } else {
      int clone = sz++;
      cloned[clone] = true;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      st[clone].firstpos = st[q].firstpos;
      while (p != -1 and st[p].next[c] == q) {
        st[p].next[c] = clone;
        p = st[p].link;
      }
      st[q].link = st[cur].link = clone;
    }
    last = cur;
  }

  bool checkOccurrence(const string &t) {  // O(len(t))
    int cur = 0;
    for (auto &c : t) {
      if (!st[cur].next.count(c)) return false;
      cur = st[cur].next[c];
    }
    return true;
  }
  ll totalSubstrings() {  // distinct, O(len(s))
```

```
    ll tot = 0;
    for (int i = 1; i < sz; i++) {
      tot += st[i].len - st[st[i].link].len;
    }
    return tot;
  }

  // count occurences of a given string t
  int countOccurences(const string &t) {
    int cur = 0;
    for (auto &c : t) {
      if (!st[cur].next.count(c)) return 0;
      cur = st[cur].next[c];
    }
    return st[cur].cnt;
  }

  // find the first index where t appears a substring
  // O(len(t))
  int firstOccurence(const string &t) {
    int cur = 0;
    for (auto c : t) {
      if (!st[cur].next.count(c)) return -1;
      cur = st[cur].next[c];
    }
    return st[cur].firstpos - len(t) + 1;
  }

  vi everyOccurence(const string &t) {
    int cur = 0;
    for (auto c : t) {
      if (!st[cur].next.count(c)) return {};
      cur = st[cur].next[c];
    }
    vi ans;
    getEveryOccurence(cur, len(t), ans);
    return ans;
  }

  void getEveryOccurence(int v, int P_length, vi &ans) {
    if (!cloned[v]) ans.pb(st[v].firstpos - P_length + 1);
    for (int u : st[v].inv_link)
      getEveryOccurence(u, P_length, ans);
  }
};
```

9.12 Trie

- build with the size of the alphabet (*sigma*) and the first char (*norm*)
- $insert(s)$ insert the string in the trie $O(|s| * sigma)$
- $erase(s)$ remove the string from the trie $O(|s|)$
- $find(s)$ return the last node from the string s, 0 if not found $O(|s|)$

```
struct trie {
  vi2d to;
  vi end, pref;
  int sigma;
  char norm;

  trie(int sigma_ = 26, char norm_ = 'a')
    : sigma(sigma_), norm(norm_) {
    to = {vector<int>(sigma)};
    end = {0}, pref = {0};
  }

  int next(int node, char key) {
    return to[node][key - norm];
  }

  void insert(const string &s) {
    int x = 0;
    for (auto c : s) {
      int &nxt = to[x][c - norm];
      if (!nxt) {
        nxt = len(to);
        to.push_back(vi(sigma));
        end.emplace_back(0), pref.emplace_back(0);
      }
      x = nxt, pref[x]++;
    }
    end[x]++, pref[0]++;
  }
  void erase(const string &s) {
    int x = 0;
    for (char c : s) {
      int &nxt = to[x][c - norm];
      x = nxt, pref[x]--;
      if (!pref[x]) nxt = 0;
    }
    end[x]--, pref[0]--;
  }
  int find(const string &s) {
```

```
    int x = 0;
    for (auto c : s) {
      x = to[x][c - norm];
      if (!x) return 0;
    }
    return x;
  }
};
```

9.13 Z-function get occurence positions

$O(len(s) + len(p))$

```
vi getOccPos(string &s, string &p) {
  // Z-function
  char delim = '#';
  string t{p + delim + s};
  vi zs(len(t));

  int l = 0, r = 0;
  for (int i = 1; i < len(t); i++) {
    if (i <= r) zs[i] = min(zs[i - l], r - i + 1);
    while (zs[i] + i < len(t) and t[zs[i]] == t[i + zs[i]])
      zs[i]++;
    if (r < i + zs[i] - 1) l = i, r = i + zs[i] - 1;
  }

  // Iterate over the results of Z-function to get ranges
  vi ans;
  int start = len(p) + 1 + 1 - 1;
  for (int i = start; i < len(zs); i++) {
    if (zs[i] == len(p)) {
      int l = i - start;
      ans.emplace_back(l);
    }
  }
  return ans;
}
```

10 Settings and macros

10.1 gen.cpp

```
#include <bits/stdc++.h>
using namespace std;
#ifdef LOCAL
#include "debug.cpp"
#else
#define dbg(...)
#endif
#define endl '\n'
#define fastio                          \
  ios_base::sync_with_stdio(0); \
  cin.tie(0);
// #define int long long
#define len(__x) (int)__x.size()
using ll = long long;
using ull = unsigned long long;
using ld = long double;
using vll = vector<ll>;
using pll = pair<ll, ll>;
using vll2d = vector<vll>;
using vi = vector<int>;
using vi2d = vector<vi>;
using pii = pair<int, int>;
using vii = vector<pii>;
using vc = vector<char>;
#define all(a) a.begin(), a.end()
#define rall(a) a.rbegin(), a.rend()
#define pb push_back
#define eb emplace_back
#define ff first
#define ss second
#define rep(i, begin, end)                               \
  for (__typeof(begin) i = (begin) - ((begin) > (end)); \
       i != (end) - ((begin) > (end));                  \
       i += 1 - 2 * ((begin) > (end)))

int lg2(ll x) {
  return __builtin_clzll(1) - __builtin_clzll(x);
}

// vector<string> dir({"LU", "U", "RU", "R", "RD", "D",
// "LD", "L"}); int dx[] = {-1, -1, -1, 0, 1, 1, 1, 0}; int
// dy[] = {-1, 0, 1, 1, 1, 0, -1, -1};
vector<string> dir({"U", "R", "D", "L"});
int dx[] = {-1, 0, 1, 0};
int dy[] = {0, 1, 0, -1};

const ll oo = 1e18;
```

```
int T(1);


ll randum() {
  std::random_device
    rd;  // Obtain a random number from hardware
  std::mt19937 gen(rd());  // Seed the generator
                           //
  std::uniform_int_distribution<> distribution(1, 4);
  return distribution(gen);
}

auto run() {
}

int32_t main(void) {
  srand(time(NULL));
#ifndef LOCAL
  fastio;
#endif

  // cin >> T;

  for (int t = 1; t <= T; t++) {
    run();
  }
}
```

10.2 macro.cpp

```
#include <bits/stdc++.h>
using namespace std;
#ifdef LOCAL
#include "debug.cpp"
#else
#define dbg(...)
#endif
#define endl '\n'
#define fastio                    \
  ios_base::sync_with_stdio(0); \
  cin.tie(0);
// #define int long long
#define len(__x) (int)__x.size()
using ll = long long;
using ull = unsigned long long;
using ld = long double;
using vll = vector<ll>;
using pll = pair<ll, ll>;
using vll2d = vector<vll>;
using vi = vector<int>;
using vi2d = vector<vi>;
using pii = pair<int, int>;
using vii = vector<pii>;
using vc = vector<char>;
#define all(a) a.begin(), a.end()
#define rall(a) a.rbegin(), a.rend()
#define pb push_back
#define eb emplace_back
#define ff first
#define ss second
#define rep(i, begin, end)                                 \
  for (__typeof(begin) i = (begin) - ((begin) > (end)); \
       i != (end) - ((begin) > (end));                    \
       i += 1 - 2 * ((begin) > (end)))

int lg2(ll x) {
  return __builtin_clzll(1) - __builtin_clzll(x);
}

// vector<string> dir({"LU", "U", "RU", "R", "RD", "D",
// "LD", "L"}); int dx[] = {-1, -1, -1, 0, 1, 1, 1, 0}; int
// dy[] = {-1, 0, 1, 1, 1, 0, -1, -1};
vector<string> dir({"U", "R", "D", "L"});
int dx[] = {-1, 0, 1, 0};
int dy[] = {0, 1, 0, -1};

const ll OO = 1e18;
const int oo = 1e9;

int T(1);

auto run() {

}

int32_t main(void) {
#ifndef LOCAL
  fastio;
#endif
```

```
  // cin >> T;

  for (int t = 1; t <= T; t++) {
    run();
  }
}
```

10.3 short-macro.cpp

```
#include <bits/stdc++.h>
using namespace std;
#define fastio                        \
  ios_base::sync_with_stdio(0); \
  cin.tie(0);

void run() {

}

int32_t main(void) {
  fastio;
  int t;
  t = 1;
  // cin >> t;
  while (t--) run();
}
```

10.4 debug.cpp

```
#include <bits/stdc++.h>
using namespace std;
/******** Debug Code *******/
template <typename T>
concept Printable = requires(T t) {
    { std::cout << t } -> std::same_as<std::ostream &>;
};
template <Printable T>
void __print(const T &x) {
    cerr << x;
}
template <size_t T>
void __print(const bitset<T> &x) {
    cerr << x;
}
template <typename A, typename B>
void __print(const pair<A, B> &p);
template <typename... A>
void __print(const tuple<A...> &t);
template <typename T>
void __print(stack<T> s);
template <typename T>
void __print(queue<T> q);
template <typename T, typename... U>
void __print(priority_queue<T, U...> q);
template <typename A>
void __print(const A &x) {
    bool first = true;
    cerr << '{';
    for (const auto &i : x) {
        cerr << (first ? "" : ","), __print(i);
        first = false;
    }
    cerr << '}';
}
template <typename A, typename B>
void __print(const pair<A, B> &p) {
    cerr << '(';
    __print(p.first);
    cerr << ',';
    __print(p.second);
    cerr << ')';
}
template <typename... A>
void __print(const tuple<A...> &t) {
    bool first = true;
    cerr << '(';
    apply(
        [&first](const auto &...args) {
            ((cerr << (first ? "" : ","), __print(args), first
   = false), ...);
        },
        t);
    cerr << ')';
}
template <typename T>
void __print(stack<T> s) {
    vector<T> debugVector;
    while (!s.empty()) {
        T t = s.top();
        debugVector.push_back(t);
        s.pop();
```

```
    }
    reverse(debugVector.begin(), debugVector.end());
    __print(debugVector);
}
template <typename T>
void __print(queue<T> q) {
    vector<T> debugVector;
    while (!q.empty()) {
        T t = q.front();
        debugVector.push_back(t);
        q.pop();
    }
    __print(debugVector);
}
template <typename T, typename... U>
void __print(priority_queue<T, U...> q) {
    vector<T> debugVector;
    while (!q.empty()) {
        T t = q.top();
        debugVector.push_back(t);
        q.pop();
    }
    __print(debugVector);
}
void _print() { cerr << "]\n"; }
template <typename Head, typename... Tail>
void _print(const Head &H, const Tail &...T) {
    __print(H);
    if (sizeof...(T)) cerr << ", ";
    _print(T...);
}

#define dbg(x...)                    \
    cerr << "[" << #x << "] = ["; \
    _print(x)
```

10.5 .bashrc

```
#copy first argument to clipborad ! ONLY WORK ON XORG !
alias clip="xclip -sel clip"

# compile the $1 parameter, if a $2 is provided
# the name will be the the binary output, if
# none is provided the binary name will be
# 'a.out'
comp() {
  echo ">> COMPILING $1 <<" 1>&2

  if [ $# -gt 1 ]; then
    outfile="${2}"
  else
    outfile="a.out"
  fi

  time g++ -std=c++20 \
      -O2 \
      -g3 \
      -Wall \
      -fsanitize=address,undefined \
      -fno-sanitize-recover \
      -D LOCAL \
      -o "${outfile}" \
      "$1"

  if [ $? -ne 0 ]; then
      echo ">> FAILED <<" 1>&2
      return 1
  fi
  echo ">> DONE << " 1>&2
}

# run the binary given in $1, if none is
# given it will try to run the 'a.out'
# binary
run() {
        to_run=./a.out
        if [ -n "$1" ]; then
                to_run="$1"
        fi
        time $to_run
}

# just comp and run your cpp file
# accpets <in1 >out and everything else
comprun() {
        comp "$1" "a"
        run ./a ${@:2}
}

testall() {
```

```
comp "$1" generator
comp "$2" brute
comp "$3" main

input_counter=1

while true; do
    echo "$input_counter"
    run ./generator > input
    run ./main < input > main_output.txt
    run ./brute < input > brute_output.txt

    diff brute_output.txt main_output.txt
    if [ $? -ne 0 ]; then
        echo "Outputs differ at input $input_counter"
        echo "Brute file output:"
        cat brute_output.txt
        echo "Main file output:"
            cat main_output.txt
␣     echo "input used: "
␣     cat input
            break
        fi

        ((input_counter++))
    done
}
```

10.6 .vimrc

```
set ts=4 sw=4 sta nu rnu sc cindent
set bg=dark ruler clipboard=unnamed,unnamedplus, timeoutlen=100
colorscheme default

nnoremap <C-j> :botright belowright term bash <CR>
syntax on
```