/* Fenwick Tree */

```
template <typename T>
struct fenwick {
   int n;
   vector<T> bit; // give 0-based and operations are done 1-based
   fenwick(int n) : n(n), bit(n + 1, T()) {}
      void update(int i, T delta) {
        for (++i; i <= n; i += i & -i) {bit[i] = bit[i] + delta;}
   }
   T query(int i) {
       T res{};
       for (++i; i > 0; i -= i & -i) {res = res + bit[i];}
       return res;
   }
};
struct node {
   int a = 0; // don't forget the default value define the required operations
   inline node operator+(node& x) {return node(a - x.a);}
   inline bool operator<(node& x) {return a < x.a;}
};
```

/* Ordered Set */

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using oset = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
```

/* Sparse Table && LCA */

```
template <typename T> class SparseTable {
private:
   int n, log2dist;
   vector<vector<T>> st;
public:
   SparseTable(const vector<T>& v) {
       n = (int)v.size();
       log2dist = 1 + (int)log2(n);
       st.resize(log2dist);
       st[0] = v;
       for (int i = 1; i < log2dist; i++) {
           st[i].resize(n - (1 << i) + 1);
           for (int j = 0; j + (1 << i) <= n; j++) {
               st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
           }
       }
   }
    T query(int l, int r) {   /** @return minimum on the range [l, r] */
       int i = (int)log2(r - l + 1);
```

```cpp
        return min(st[i][l], st[i][r - (1 << i) + 1]);
    }
};
class LCA {
private:
    const int n;
    const vector<vector<int>>& adj;
    SparseTable<pair<int, int>> rmq;
    vector<int> tin, et, dep;
    int timer = 0;
      void dfs(int u, int p) {  /** prepares tin, et, dep arrays */
        tin[u] = timer;
        et[timer++] = u;
        for (int v : adj[u]) {
            if (v == p) { continue; }
            dep[v] = dep[u] + 1;
            dfs(v, u);
            et[timer++] = u;
        }
    }
public:
    // make sure the adjacency list is 0 indexed
    LCA(vector<vector<int>>& _adj)
        : n((int)_adj.size()), adj(_adj), tin(n), et(2 * n), dep(n),
        rmq(vector<pair<int, int>>(1)) {
        dfs(0, -1);
        vector<pair<int, int>> arr(2 * n);
        for (int i = 0; i < 2 * n; i++) { arr[i] = {dep[et[i]], et[i]}; }
        rmq = SparseTable<pair<int, int>>(arr);
    }
      int query(int a, int b) {  /** @return LCA of nodes a and b */
        if (tin[a] > tin[b]) { swap(a, b); }
        return rmq.query(tin[a], tin[b]).second;
    }
      int dist(int a, int b) { /** @return dist between node a and b */
        int c = query(a, b);
        return dep[a] + dep[b] - 2 * dep[c];
    }
};
/* Z-Algorithm */
vector<int> z_func(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 0; i < n; i++) {
        z[i] = max(0, min(z[i - x], y - i + 1));
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
```

```
            x = i, y = i + z[i], z[i]++;
        }
    }
    return z;
}
```
/* Trie */
```
struct Node {
    Node* links[26];
    bool eow;  // flag for marking the end of word
    int endCount = 0, prefixCount = 0;
    bool containsKey(char ch) {return links[ch - 'a'] != NULL;}
    // insert a new node with a specific key (letter) to the Trie
    void put(char ch, Node* node) {links[ch - 'a'] = node;}
    // get the node associated to a specific key (letter)
    Node* get(char ch) {return links[ch - 'a'];}
    // mark the end of the word
    void setEnd() {eow = true;}
    // check is the key is the end of the word or not
    bool isEnd() {return eow;}
};
class Trie {
private:
    Node* root;
public:
    Trie() {root = new Node();}
    // insert word into the Trie
    // time complexity : O(len) where len is length of the word
    void insert(string word) {
        Node* node = root;
        for (int i = 0; i < word.length(); i++) {
            if (!node->containsKey(word[i])) {
                node->put(word[i], new Node());
            }
            node = node->get(word[i]), node->prefixCount++;
        }
        node->setEnd(), node->endCount++;
    }
    bool search(string word) { // search for the word within the Trie
        Node* node = root;
        for (int i = 0; i < word.length(); i++) {
            if (!node->containsKey(word[i])) {return false;}
            node = node->get(word[i]);
        }
        return node->isEnd();
    }
    // return whether any word with the given prefix
    bool startsWith(string prefix) {
```

```
        Node* node = root;
        for (int i = 0; i < prefix.length(); i++) {
            if (!node->containsKey(prefix[i])) {return false;}
            node = node->get(prefix[i]);
        }
        return true;
    }
    // return the count of the occurrences of the string word in the Trie
    int cntWord(string word) {
        Node* node = root;
        for (int i = 0; i < word.length(); i++) {
            if (!node->containsKey(word[i])) {return 0;}
            node = node->get(word[i]);
        }
        return node->endCount;
    }
    // return the count of words starting with the given prefix
    int cntPrefix(string word) {
        Node* node = root;
        // int res = 0;
        for (int i = 0; i < word.length(); i++) {
            if (!node->containsKey(word[i])) {return 0; // return res;}
            node = node->get(word[i]); // res += node->prefixCount;
        }
        return node->prefixCount; // return res;
    }
    // erase a word in the given trie
    void erase(string word) {
        Node* node = root;
        for (int i = 0; i < word.length(); i++) {
            node = node->get(word[i]);
            node->prefixCount--;
        }
        node->endCount--;
    }
};
/* Segment Tree */
template <class S, S(*op)(S, S), S(*e)()> struct segtree {
public:
    segtree() : segtree(0) {}
    segtree(int32_t n) : segtree(std::vector<S>(n, e())) {}
    segtree(const std::vector<S>& v) : _n(int32_t(v.size())) {
        log = 64 - __builtin_clzll(_n);
        size = 1 << log;
        d = std::vector<S>(2 * size, e());
        for (int32_t i = 0; i < _n; i++) d[size + i] = v[i];
        for (int32_t i = size - 1; i >= 1; i--) {update(i);}
```

```
    }
    void set(int32_t p, S x) {
        assert(0 <= p && p < _n);
        p += size;
        d[p] = x;
        for (int32_t i = 1; i <= log; i++) update(p >> i);
    }
    S get(int32_t p) {
        assert(0 <= p && p < _n);
        return d[p + size];
    }
    S prod(int32_t l, int32_t r) {
        assert(0 <= l && l <= r && r <= _n);
        S sml = e(), smr = e();
        l += size;
        r += size;

        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);
    }
    S all_prod() { return d[1]; }
    template <bool (*f)(S)> int32_t max_right(int32_t l) {
        return max_right(l, [](S x) { return f(x); });
    }
    template <class F> int32_t max_right(int32_t l, F f) {
        assert(0 <= l && l <= _n);
        assert(f(e()));
        if (l == _n) return _n;
        l += size;
        S sm = e();
        do {
            while (l % 2 == 0) l >>= 1;
            if (!f(op(sm, d[l]))) {
                while (l < size) {
                    l = (2 * l);
                    if (f(op(sm, d[l]))) {
                        sm = op(sm, d[l]);
                        l++;
                    }
                }
                return l - size;
            }
```

```cpp
            sm = op(sm, d[l]);
            l++;
        } while ((l & -l) != l);
        return _n;
    }
    template <bool (*f)(S)> int32_t min_left(int32_t r) {
        return min_left(r, [](S x) { return f(x); });
    }
    template <class F> int32_t min_left(int32_t r, F f) {
        assert(0 <= r && r <= _n);
        assert(f(e()));
        if (r == 0) return 0;
        r += size;
        S sm = e();
        do {
            r--;
            while (r > 1 && (r % 2)) r >>= 1;
            if (!f(op(d[r], sm))) {
                while (r < size) {
                    r = (2 * r + 1);
                    if (f(op(d[r], sm))) {
                        sm = op(d[r], sm);
                        r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(d[r], sm);
        } while ((r & -r) != r);
        return 0;
    }
private:
    int32_t _n, size, log;
    std::vector<S> d;
    void update(int32_t k) { d[k] = op(d[2 * k], d[2 * k + 1]); }
};
/* Lazy Segment Tree */
template <class S,
    S(*op)(S, S),
    S(*e)(),
    class F,
    S(*mapping)(F, S),
    F(*composition)(F, F),
    F(*id)()>
struct lazy_segtree {
public:
    lazy_segtree() : lazy_segtree(0) {}
```

```cpp
lazy_segtree(int32_t n) : lazy_segtree(std::vector<S>(n, e())) {}
lazy_segtree(const std::vector<S>& v) : _n(int32_t(v.size())) {
    log = 64 - __builtin_clzll(_n);
    size = 1 << log;
    d = std::vector<S>(2 * size, e());
    lz = std::vector<F>(size, id());
    for (int32_t i = 0; i < _n; i++) d[size + i] = v[i];
    for (int32_t i = size - 1; i >= 1; i--) {update(i);}
}
void set(int32_t p, S x) {
    assert(0 <= p && p < _n);
    p += size;
    for (int32_t i = log; i >= 1; i--) push(p >> i);
    d[p] = x;
    for (int32_t i = 1; i <= log; i++) update(p >> i);
}
S get(int32_t p) {
    assert(0 <= p && p < _n);
    p += size;
    for (int32_t i = log; i >= 1; i--) push(p >> i);
    return d[p];
}
S prod(int32_t l, int32_t r) {
    assert(0 <= l && l <= r && r <= _n);
    if (l == r) return e();
    l += size;
    r += size;
    for (int32_t i = log; i >= 1; i--) {
        if (((l >> i) << i) != l) push(l >> i);
        if (((r >> i) << i) != r) push(r >> i);
    }
    S sml = e(), smr = e();
    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);
}
S all_prod() { return d[1]; }
void apply(int32_t p, F f) {
    assert(0 <= p && p < _n);
    p += size;
    for (int32_t i = log; i >= 1; i--) push(p >> i);
    d[p] = mapping(f, d[p]);
    for (int32_t i = 1; i <= log; i++) update(p >> i);
}
```

```
void apply(int32_t l, int32_t r, F f) {
    assert(0 <= l && l <= r && r <= _n);
    if (l == r) return;
    l += size; r += size;
    for (int32_t i = log; i >= 1; i--) {
        if (((l >> i) << i) != l) push(l >> i);
        if (((r >> i) << i) != r) push((r - 1) >> i);
    }
    {
        int32_t 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 (int32_t i = 1; i <= log; i++) {
        if (((l >> i) << i) != l) update(l >> i);
        if (((r >> i) << i) != r) update((r - 1) >> i);
    }
}
template <bool (*g)(S)> int32_t max_right(int32_t l) {
    return max_right(l, [](S x) { return g(x); });
}
template <class G> int32_t max_right(int32_t l, G g) {
    assert(0 <= l && l <= _n);
    assert(g(e()));
    if (l == _n) return _n;
    l += size;
    for (int32_t i = log; i >= 1; i--) push(l >> i);
    S sm = e();
    do {
        while (l % 2 == 0) l >>= 1;
        if (!g(op(sm, d[l]))) {
            while (l < size) {
                push(l);
                l = (2 * l);
                if (g(op(sm, d[l]))) {
                    sm = op(sm, d[l]); l++;
                }
            }
            return l - size;
        }
        sm = op(sm, d[l]);
        l++;
    } while ((l & -l) != l);
```

```
        return _n;
    }
    template <bool (*g)(S)> int32_t min_left(int32_t r) {
        return min_left(r, [](S x) { return g(x); });
    }
    template <class G> int32_t min_left(int32_t r, G g) {
        assert(0 <= r && r <= _n);
        assert(g(e()));
        if (r == 0) return 0;
        r += size;
        for (int32_t i = log; i >= 1; i--) push((r - 1) >> i);
        S sm = e();
        do {
            r--;
            while (r > 1 && (r % 2)) r >>= 1;
            if (!g(op(d[r], sm))) {
                while (r < size) {
                    push(r);
                    r = (2 * r + 1);
                    if (g(op(d[r], sm))) {
                        sm = op(d[r], sm); r--;
                    }
                }
                return r + 1 - size;
            }
            sm = op(d[r], sm);
        } while ((r & -r) != r);
        return 0;
    }
private:
    int32_t _n, size, log;
    std::vector<S> d;
    std::vector<F> lz;
    void update(int32_t k) { d[k] = op(d[2 * k], d[2 * k + 1]); }
    void all_apply(int32_t k, F f) {
        d[k] = mapping(f, d[k]);
        if (k < size) lz[k] = composition(f, lz[k]);
    }
    void push(int32_t k) {
        all_apply(2 * k, lz[k]);
        all_apply(2 * k + 1, lz[k]);
        lz[k] = id();
    }
};
/* Disjoint Set Union */
struct DSU {
 private:
```

```cpp
  vector<int> par, sizes;

public:
  DSU(int n) : par(n), sizes(n, 1) { iota(par.begin(), par.end(), 0); }
  int find(int x) { return (par[x] == x ? x : par[x] = find(par[x])); }
  bool unite(int x, int y) {
      int x_root = find(x), y_root = find(y);
      if (x_root == y_root)
          return false;
      if (sizes[x_root] < sizes[y_root])
          swap(x_root, y_root);
      sizes[x_root] += sizes[y_root];
      par[y_root] = x_root;
      return true;
  }
  int tree_len(int x) { return sizes[find(x)]; }
};

int sumDigitsUpto(int n) {
 if (n <= 0)
   return 0;
 int res = 0, p = 1;
 while (p <= n) {
   int left = n / (p * 10); // higher part (digits left of current position)
   int cur = (n / p) % 10;  // current digit at position p
   int right = n % p;       // lower part (digits right of current position)

   res += left * 45 * p; // contribution of aint fuint cycles
   res += (cur * (cur - 1) / 2) *
     p; // contribution of partial cycle from 0..cur-1
   res += cur *
     (right +
      1); // contribution from the remaining part for the  current digit
   p *= 10;
 }
 return res;
}
```