Competitive Programming Notebook

Gian Alingog

Generated from `content/` — hash next to each snippet verifies you typed it correctly.

Contents

Data Structures

Fast Bitset

8f3cd1ba

Fixed-size bitset (compile-time N), faster than std::bitset for shift-heavy workloads (bitset-DP via <<=, |=) because it exposes find_next/find_prev directly instead of forcing a linear scan. Stress tested on Codeforces. Indices must be < N; out-of-range access is UB in release builds (asserted in debug). 2-page exception to the line budget.

Time: O(N / 64) per whole-bitset op (shift, and/or/xor, count); O(1) amortized per find_next/find_prev step. tested, stress tested on CF

```
template <size_t N>
struct FastBitset64 {
    // ensure unsigned and 64 bit; do not use 'long long',
word size may differ
    using u64 = uint64_t;

    static constexpr size_t POW_2 = 6;
    static constexpr size_t WORD_SIZE = u64(1) << POW_2;
    static constexpr u64 FULL_MASK = WORD_SIZE - 1;
    static constexpr size_t WORDS = (N + WORD_SIZE - 1) /
WORD_SIZE;

    array<u64, WORDS> a{};

    static constexpr u64 LAST_MASK = (N % WORD_SIZE == 0 ?
~u64(0) : ((u64(1) << (N % WORD_SIZE)) - 1));
    static inline size_t ctz(u64 x) { return
__builtin_ctzll(x); }
    static inline size_t clz(u64 x) { return
__builtin_clzll(x); }
    static inline size_t pop(u64 x) { return
__builtin_popcountll(x); }

    inline void fix_last() {
        if constexpr (N % WORD_SIZE) a[WORDS - 1] &=
LAST_MASK;
    }

    static constexpr size_t size() { return N; }
    inline void reset() { a.fill(0); }
    inline void set() {
        a.fill(~u64(0));
        fix_last();
    }

    inline void set(size_t i, bool v = true) {
        assert(i < N);
        size_t w = i >> POW_2, b = i & FULL_MASK;
        u64 m = u64(1) << b;
        if (v) a[w] |= m;
        else a[w] &= ~m;
    }

    inline bool test(size_t i) const {
        assert(i < N);
        size_t w = i >> POW_2, b = i & FULL_MASK;
        return (a[w] >> b) & u64(1);
    }

    inline void flip() {
        for (size_t i = 0; i < WORDS; i++) a[i] = ~a[i];
        fix_last();
    }

    inline void flip(size_t i) {
        assert(i < N);
        a[i >> POW_2] ^= (u64(1) << (i & FULL_MASK));
    }

    inline bool any() const {
        for (size_t i = 0; i < WORDS; i++) if (a[i]) return
true;
        return false;
    }

    inline bool none() const { return !any(); }

    inline bool all() const {
        for (size_t i = 0; i + 1 < WORDS; i++) if (~a[i])
return false;
        if constexpr (N % WORD_SIZE) return (a[WORDS - 1] &
LAST_MASK) == LAST_MASK;
        else return (a[WORDS - 1] == ~u64(0));
    }

    inline size_t count() const {
        size_t c = 0;
        for (size_t i = 0; i < WORDS; i++) c += pop(a[i]);
        return c;
    }

    inline u64 to_llong() const { return (WORDS ? a[0] :
u64(0)); }  // first word only

    string to_string() const {
        string s;
        s.reserve(N);
        for (size_t i = N; i --> 0;) s.push_back(char('0' +
test(i)));
        return s;
    }

    inline FastBitset64& operator&=(const FastBitset64 &o) {
        for (size_t i = 0; i < WORDS; i++) a[i] &= o.a[i];
        return *this;
    }

    inline FastBitset64& operator|=(const FastBitset64 &o) {
        for (size_t i = 0; i < WORDS; i++) a[i] |= o.a[i];
        return *this;
    }

    inline FastBitset64& operator^=(const FastBitset64 &o) {
        for (size_t i = 0; i < WORDS; i++) a[i] ^= o.a[i];
        return *this;
    }

    friend inline FastBitset64 operator&(FastBitset64 lhs,
const FastBitset64 &rhs) { lhs &= rhs; return lhs; }
    friend inline FastBitset64 operator|(FastBitset64 lhs,
const FastBitset64 &rhs) { lhs |= rhs; return lhs; }
    friend inline FastBitset64 operator^(FastBitset64 lhs,
const FastBitset64 &rhs) { lhs ^= rhs; return lhs; }
    friend inline bool operator==(const FastBitset64 &x,
const FastBitset64 &y) { return x.a == y.a; }
    friend inline bool operator!=(const FastBitset64 &x,
const FastBitset64 &y) { return !(x == y); }

    inline FastBitset64& operator<<=(size_t i) {
        if (i >= N) { reset(); return *this; }
        size_t w = i >> POW_2, b = i & FULL_MASK;
        if (b == 0) {
            for (size_t j = WORDS; j --> 0;) a[j] = (j >=
w) ? a[j - w] : 0;
        } else {
            for (size_t j = WORDS; j --> 0;) {
                u64 big = (j >= w) ? (a[j - w] << b) : 0;
                u64 sml = (j >= w + 1) ? (a[j - w - 1] >>
(WORD_SIZE - b)) : 0;
                a[j] = big | sml;
            }
        }
        fix_last();
        return *this;
    }

    inline FastBitset64& operator>>=(size_t i) {
        if (i >= N) { reset(); return *this; }
        size_t w = i >> POW_2, b = i & FULL_MASK;
        if (b == 0) {
            for (size_t j = 0; j < WORDS; j++) a[j] = (j + w
< WORDS) ? a[j + w] : 0;
        } else {
```

```
            // ascending: a[j] reads a[j+w]/a[j+w+1] (higher
indices), so low-to-high
            // avoids clobbering a source word before it's
read (opposite of operator<<=,
            // which reads lower indices and must go high-
to-low for the same reason).
            for (size_t j = 0; j < WORDS; j++) {
                u64 sml = (j + w < WORDS) ? (a[j + w] >>
b) : 0;
                u64 big = (j + w + 1 < WORDS) ? (a[j + w +
1] << (WORD_SIZE - b)) : 0;
                a[j] = sml | big;
            }
        }
        fix_last();
        return *this;
    }

    friend inline FastBitset64 operator<<(FastBitset64 x,
size_t i) { x <<= i; return x; }
    friend inline FastBitset64 operator>>(FastBitset64 x,
size_t i) { x >>= i; return x; }

    inline size_t find_first() const {
        for (size_t i = 0; i < WORDS; i++) if (a[i]) return
(i << POW_2) + ctz(a[i]);
        return N;
    }

    inline size_t find_next(size_t i) const {  //
upper_bound
        i++;
        if (i >= N) return N;
        size_t w = i >> POW_2, b = i & FULL_MASK;
        u64 x = a[w] & (~u64(0) << b);
        if (x) return (w << POW_2) + ctz(x);
        for (++w; w < WORDS; w++) if (a[w]) return (w <<
POW_2) + ctz(a[w]);
        return N;
    }

    inline size_t find_last() const {  // N means invalid,
same convention throughout
        for (size_t i = WORDS; i --> 0;) {
            u64 x = a[i];
            if constexpr (N % WORD_SIZE) if (i == WORDS - 1)
x &= LAST_MASK;
            if (x) return (i << POW_2) + (FULL_MASK -
clz(x));
        }
        return N;
    }

    inline size_t find_prev(size_t i) const {
        if (i == 0) return N;
        i--;
        size_t w = i >> POW_2, b = i & FULL_MASK;
        u64 m = (b == FULL_MASK) ? ~u64(0) : ((u64(1) << (b
+ 1)) - 1);
        u64 x = a[w] & m;
        if (x) return (w << POW_2) + (FULL_MASK - clz(x));
        while (w--) if (a[w]) return (w << POW_2) +
(FULL_MASK - clz(a[w]));
        return N;
    }
};
```

Fenwick

7c9ea88d

Fenwick tree (BIT). Point update, prefix/range sum, 0-indexed interface. T needs operator+, operator-, and a default-constructed zero.

Time: O(\log n) per operation. tested

```
template <typename T>
struct Fenwick {
    int n;
    vector<T> tree;

    Fenwick(int n) : n(n), tree(n + 1, T()) {}

    void add(int i, T v) {
        for (i++; i <= n; i += i & -i) tree[i] += v;
    }

    T sum(int i) {
        T res = T();
        for (i++; i; i -= i & -i) res += tree[i];
        return res;
    }

    T sum(int l, int r) { return sum(r) - sum(l - 1); }
};
```

Fenwick 2d

4d1b64cb

2D Fenwick tree. Point update, rectangle sum query, 1-indexed interface. T needs operator+, operator-, and a default-constructed zero.

Time: O(\log n \log m) per operation. tested

```
template <typename T>
struct Fenwick2D {
    int n, m;
    vector<vector<T>> tree;

    Fenwick2D(int n, int m) : n(n), m(m), tree(n + 1,
vector<T>(m + 1, T())) {}

    void add(int r, int c, T v) {
        for (; r <= n; r += r & -r)
            for (int x = c; x <= m; x += x & -x) tree[r][x]
+= v;
    }

    T sum(int r, int c) {
        T res = T();
        for (; r; r -= r & -r)
            for (int x = c; x; x -= x & -x) res += tree[r]
[x];
        return res;
    }

    T sum(int r1, int c1, int r2, int c2) {
        return sum(r2, c2) - sum(r1 - 1, c2) - sum(r2, c1 -
1) + sum(r1 - 1, c1 - 1);
    }
};
```

Link Cut Tree

baedc2bd

Link-cut tree over a fixed set of n labeled nodes (1-indexed; node 0 is a null placeholder). Supports link, cut, findRoot, and path aggregate (default sum) between two nodes via makeRoot + access. To reset between test cases, construct a new LinkCutTree(n). T needs operator+ and a default-constructed zero. Multi-page exception to the line budget.

Time: O(\log n) amortized per operation. tested

```
template <typename T>
struct LinkCutTree {
    struct Node {
        int ch[2] = {0, 0}, fa = 0;
        bool rev = false;
        T val = T(), sum = T();
    };
    vector<Node> t;

    LinkCutTree(int n) { t.assign(n + 1, Node()); }

    bool isRoot(int x) { int f = t[x].fa; return f == 0 ||
(t[f].ch[0] != x && t[f].ch[1] != x); }

    void pushUp(int x) {
        t[x].sum = t[x].val + (t[x].ch[0] ?
t[t[x].ch[0]].sum : T()) + (t[x].ch[1] ? t[t[x].ch[1]].sum :
T());
    }
```

```
    void pushRev(int x) {
        if (!x) return;
        swap(t[x].ch[0], t[x].ch[1]);
        t[x].rev ^= 1;
    }

    void pushDown(int x) {
        if (t[x].rev) {
            pushRev(t[x].ch[0]);
            pushRev(t[x].ch[1]);
            t[x].rev = false;
        }
    }

    void pushDownAll(int x) {  // propagate lazy tags from the splay-tree root down to x
        if (!isRoot(x)) pushDownAll(t[x].fa);
        pushDown(x);
    }

    void rotate(int x) {
        int y = t[x].fa, z = t[y].fa;
        int dx = (t[y].ch[1] == x), dy = (t[z].ch[1] == y);
        if (!isRoot(y)) t[z].ch[dy] = x;
        t[x].fa = z;
        t[y].ch[dx] = t[x].ch[dx ^ 1];
        if (t[x].ch[dx ^ 1]) t[t[x].ch[dx ^ 1]].fa = y;
        t[x].ch[dx ^ 1] = y;
        t[y].fa = x;
        pushUp(y); pushUp(x);
    }

    void splay(int x) {
        pushDownAll(x);
        while (!isRoot(x)) {
            int y = t[x].fa;
            if (!isRoot(y)) {
                if ((t[t[y].fa].ch[0] == y) ^ (t[y].ch[0] == x)) rotate(x);
                else rotate(y);
            }
            rotate(x);
        }
        pushUp(x);
    }

    int access(int x) {  // makes the root-to-x path the preferred path; splays x to the top
        int last = 0;
        for (int y = x; y; y = t[y].fa) {
            splay(y);
            t[y].ch[1] = last;
            pushUp(y);
            last = y;
        }
        splay(x);
        return last;
    }

    void makeRoot(int x) { access(x); pushRev(x); }

    int findRoot(int x) {
        access(x);
        while (t[x].ch[0]) { pushDown(x); x = t[x].ch[0]; }
        splay(x);
        return x;
    }

    void link(int x, int y) {  // requires x, y in different trees
        makeRoot(x);
        if (findRoot(y) != x) t[x].fa = y;
    }

    void cut(int x, int y) {  // requires the edge (x, y) to exist
        makeRoot(x);
        access(y);
        if (t[y].ch[0] == x && t[x].fa == y && t[x].ch[1] == 0) {
            t[y].ch[0] = 0;
            t[x].fa = 0;
            pushUp(y);
        }
    }

    T pathSum(int x, int y) {  // requires x, y in the same tree
        makeRoot(x);
        access(y);
        return t[y].sum;
    }

    void setVal(int x, T v) {
        access(x);
        t[x].val = v;
        pushUp(x);
    }
};
```

Pbds Order Statistics Tree

92dca843

Order-statistics tree via GCC's built-in policy-based tree. find_by_order(k) gives the kth smallest element (0-indexed, as an iterator); order_of_key(x) gives the count of elements strictly less than x. Fastest option when it fits, but no split/merge and no direct erase-by-value (erase by iterator); for those, or for duplicate keys, use treap-ordered.h or splay-ordered-set.h instead.

Time: O(\log n) per operation. tested

```
#undef int  // pb_ds headers use 'int' internally; #define int long long corrupts them
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#define int long long
using namespace __gnu_pbds;

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

// usage:
//   ordered_set<int> s;
//   s.insert(5);
//   *s.find_by_order(0);   // smallest element
//   s.order_of_key(5);     // # elements < 5
//   s.erase(s.find_by_order(0));
```

Segment Tree Dynamic Lazy

eba76cf2

Dynamic (sparsely allocated) segment tree over an implicit range, e.g. [0, 1e9]. Nodes are created only along touched paths, so no initial array is needed and no coordinate compression is required. Range add + range sum; T needs operator+ and a default-constructed zero.

Time: O(\log(\text{range})) per operation; O(q \log(\text{range})) nodes total. tested

```
template <typename T>
struct DynamicSegmentTree {
    struct Node {
        T val = T(), lazy = T();
        Node *left = nullptr, *right = nullptr;
    };

    long long lo, hi;
    Node *root = new Node();

    DynamicSegmentTree(long long lo, long long hi) : lo(lo), hi(hi) {}

    void push(Node *n, long long l, long long r) {
        if (!n->left) { n->left = new Node(); n->right = new Node(); }
        if (n->lazy != T()) {
            long long m = (l + r) / 2;
```

```
            n->left->val += n->lazy * (m - l + 1);
            n->left->lazy += n->lazy;
            n->right->val += n->lazy * (r - m);
            n->right->lazy += n->lazy;
            n->lazy = T();
        }
    }

    void update(long long ul, long long ur, T v, Node *n =
nullptr, long long l = -1, long long r = -1) {
        if (!n) { n = root; l = lo; r = hi; }
        if (ur < l || r < ul) return;
        if (ul <= l && r <= ur) { n->val += v * (r - l + 1);
n->lazy += v; return; }
        push(n, l, r);
        long long m = (l + r) / 2;
        update(ul, ur, v, n->left, l, m);
        update(ul, ur, v, n->right, m + 1, r);
        n->val = n->left->val + n->right->val;
    }

    T query(long long ql, long long qr, Node *n = nullptr,
long long l = -1, long long r = -1) {
        if (!n) { n = root; l = lo; r = hi; }
        if (qr < l || r < ql) return T();
        if (ql <= l && r <= qr) return n->val;
        push(n, l, r);
        long long m = (l + r) / 2;
        return query(ql, qr, n->left, l, m) + query(ql, qr,
n->right, m + 1, r);
    }
};
```

Segment Tree Iterative

cee7ab5d

Iterative point-update segment tree, no recursion, no lazy propagation. Default op is max; edit comb()/ID for a different associative op.

Time: O(\log n) per operation. tested

```
template <typename T>
struct SegmentTree {
    int n;
    vector<T> tree;
    T ID = numeric_limits<T>::lowest();  // identity for
comb()

    T comb(T a, T b) { return max(a, b); }  // op: edit for
a different associative op

    SegmentTree(vector<T> &a) : n(a.size()), tree(2 * n) {
        for (int i = 0; i < n; i++) tree[i + n] = a[i];
        for (int i = n - 1; i > 0; i--) tree[i] =
comb(tree[2 * i], tree[2 * i + 1]);
    }

    void update(int i, T v) {
        tree[i += n] = v;
        for (i >>= 1; i > 0; i >>= 1) tree[i] = comb(tree[2
* i], tree[2 * i + 1]);
    }

    T query(int l, int r) {  // inclusive [l, r]
        T res = ID;
        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) res = comb(res, tree[l++]);
            if (r & 1) res = comb(res, tree[--r]);
        }
        return res;
    }
};
```

Segment Tree Lazy

1c016c3d

Recursive segment tree, eager array build, range update with lazy propagation. Default op is range-add + range-min; edit comb()/apply() for another op (see treap-implicit.h for the op-block pattern this follows). T needs operator+, a zero, and to be comparable with numeric_limits.

Time: O(\log n) per operation. tested

```
template <typename T>
struct LazySegmentTree {
    struct Node {
        int l, r;
        T val, lazy = T();
        Node *left = nullptr, *right = nullptr;
    };

    T ID() { return
numeric_limits<T>::max(); }              // op: identity for
comb()
    T comb(T a, T b) { return min(a,
b); }                    // op: combine two children
    void apply(Node *n, T v) { n->val += v; n->lazy +=
v; }   // op: push a lazy tag onto a node

    Node *root;

    LazySegmentTree(vector<T> &a) { root = build(0,
(int)a.size() - 1, a); }

    Node *build(int l, int r, vector<T> &a) {
        Node *n = new Node();
        n->l = l; n->r = r;
        if (l == r) { n->val = a[l]; return n; }
        int m = (l + r) / 2;
        n->left = build(l, m, a);
        n->right = build(m + 1, r, a);
        n->val = comb(n->left->val, n->right->val);
        return n;
    }

    void push(Node *n) {
        if (n->lazy != T()) {
            if (n->left) { apply(n->left, n->lazy); apply(n-
>right, n->lazy); }
            n->lazy = T();
        }
    }

    void update(int ul, int ur, T v, Node *n = nullptr) {
        if (!n) n = root;
        if (ur < n->l || n->r < ul) return;
        if (ul <= n->l && n->r <= ur) { apply(n, v);
return; }
        push(n);
        update(ul, ur, v, n->left);
        update(ul, ur, v, n->right);
        n->val = comb(n->left->val, n->right->val);
    }

    T query(int ql, int qr, Node *n = nullptr) {
        if (!n) n = root;
        if (qr < n->l || n->r < ql) return ID();
        if (ql <= n->l && n->r <= qr) return n->val;
        push(n);
        return comb(query(ql, qr, n->left), query(ql, qr, n-
>right));
    }
};
```

Segment Tree Max Subarray

6ca7ad01

Segment tree computing the maximum subarray sum (Kadane's) under point updates, via the merge trick: each node tracks sum / best-prefix / best-suffix / best-subarray. T needs operator+ and a very negative sentinel (NEG) with headroom below real value magnitudes.

Time: O(\log n) per operation. tested

```
template <typename T>
struct MaxSubarrayTree {
    static constexpr T NEG = numeric_limits<T>::lowest() /
2;  // headroom avoids overflow

    struct Data {
        T sum = 0, pref = NEG, suff = NEG, best = NEG;
    };
```

```
    int n;
    vector<Data> tree;

    Data leaf(T v) { return {v, v, v, v}; }

    Data comb(const Data &a, const Data &b) {
        Data r;
        r.sum = a.sum + b.sum;
        r.pref = max(a.pref, a.sum + b.pref);
        r.suff = max(b.suff, b.sum + a.suff);
        r.best = max({a.best, b.best, a.suff + b.pref});
        return r;
    }

    MaxSubarrayTree(vector<T> &a) : n(a.size()), tree(2 * n)
{
        for (int i = 0; i < n; i++) tree[i + n] =
leaf(a[i]);
        for (int i = n - 1; i > 0; i--) tree[i] =
comb(tree[2 * i], tree[2 * i + 1]);
    }

    void update(int i, T v) {
        tree[i += n] = leaf(v);
        for (i >>= 1; i > 0; i >>= 1) tree[i] = comb(tree[2
* i], tree[2 * i + 1]);
    }

    T query(int l, int r) {  // inclusive [l, r]; returns
the best-subarray-sum
        Data resL, resR;
        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) resL = comb(resL, tree[l++]);
            if (r & 1) resR = comb(tree[--r], resR);
        }
        return comb(resL, resR).best;
    }
};
```

Sparse Table

0a74c0ca

Sparse table for static range queries over an idempotent, associative op (default min). Edit comb() for a different op.

Time: O(n \log n) build, O(1) query. tested

```
template <typename T>
struct SparseTable {
    vector<vector<T>> table;
    int n, logn;

    T comb(T a, T b) { return min(a, b); }  // op: edit for
a different idempotent op

    SparseTable(vector<T> &a) : n(a.size()) {
        logn = 1;
        while ((1 << logn) <= n) logn++;
        table.assign(logn, vector<T>(n));
        table[0] = a;
        for (int j = 1; j < logn; j++)
            for (int i = 0; i + (1 << j) <= n; i++)
                table[j][i] = comb(table[j - 1][i], table[j
- 1][i + (1 << (j - 1))]);
    }

    T query(int l, int r) {  // inclusive [l, r]
        int j = 31 - __builtin_clz(r - l + 1);
        return comb(table[j][l], table[j][r - (1 << j) +
1]);
    }
};
```

Splay Implicit

53ebabf0

Implicit (positional) splay tree: array-as-balanced-BST via splaying. Supports insert-at-position, erase-at-position, range reverse, and an "op" block (aggregate + lazy tag) you swap for the problem at hand — default here is an affine range update (x -> mul*x + add) + range-sum, a richer example than treap-implicit.h's additive one; same op-block pattern (see treap-implicit.h). T needs operator+, operator*, and default-constructed 0/1. Uses two internal sentinel positions. Multi-page exception to the line budget.

Time: O(\log n) amortized per operation. tested

```
template <typename T>
struct SplayImplicit {
    struct Node {
        // ---- op: edit for a different aggregate/lazy tag
(see treap-implicit.h) ----
        T val, agg;
        T mul = T(1), add = T();  // affine lazy: x -> mul*x
+ add
        // ---- end op ----
        int p = -1, ch[2] = {-1, -1}, sz = 1, rev = 0;
    };
    vector<Node> pool;
    int root = -1, n = 0;  // n = logical size, excludes the
2 sentinels

    void apply(int u, T mul, T add) {  // op: push an affine
tag onto u
        pool[u].val = mul * pool[u].val + add;
        pool[u].agg = mul * pool[u].agg + add * pool[u].sz;
        pool[u].mul = mul * pool[u].mul;
        pool[u].add = mul * pool[u].add + add;
    }

    int getSz(int u) { return u == -1 ? 0 : pool[u].sz; }
    T getAgg(int u) { return u == -1 ? T() : pool[u].agg; }
    int make(T x) { pool.push_back({x, x}); return
(int)pool.size() - 1; }

    void pull(int u) {
        pool[u].sz = 1 + getSz(pool[u].ch[0]) +
getSz(pool[u].ch[1]);
        pool[u].agg = getAgg(pool[u].ch[0]) + pool[u].val +
getAgg(pool[u].ch[1]);
    }

    void push(int u) {
        if (pool[u].rev) {
            swap(pool[u].ch[0], pool[u].ch[1]);
            if (pool[u].ch[0] != -1) pool[pool[u].ch[0]].rev
^= 1;
            if (pool[u].ch[1] != -1) pool[pool[u].ch[1]].rev
^= 1;
            pool[u].rev = 0;
        }
        if (pool[u].mul != T(1) || pool[u].add != T()) {
            if (pool[u].ch[0] != -1) apply(pool[u].ch[0],
pool[u].mul, pool[u].add);
            if (pool[u].ch[1] != -1) apply(pool[u].ch[1],
pool[u].mul, pool[u].add);
            pool[u].mul = T(1); pool[u].add = T();
        }
    }

    int dir(int u) { return pool[pool[u].p].ch[1] == u; }

    void rotate(int u) {
        int f = pool[u].p, g = pool[f].p, d = dir(u), w =
pool[u].ch[d ^ 1];
        pool[f].ch[d] = w;
        if (w != -1) pool[w].p = f;
        pool[u].ch[d ^ 1] = f;
        pool[f].p = u;
        pool[u].p = g;
        if (g != -1) {
            if (pool[g].ch[0] == f) pool[g].ch[0] = u;
            else if (pool[g].ch[1] == f) pool[g].ch[1] = u;
        }
        pull(f); pull(u);
    }

    void pushPath(int u) {
        if (pool[u].p != -1) pushPath(pool[u].p);
```

```
        push(u);
    }

    void splay(int u, int g = -1) {
        pushPath(u);
        while (pool[u].p != g) {
            int f = pool[u].p, gg = pool[f].p;
            if (gg != g) {
                if (dir(u) == dir(f)) rotate(f);
                else rotate(u);
            }
            rotate(u);
        }
        if (g == -1) root = u;
    }

    int findKth(int k, int g = -1) {  // 0-indexed over the internal (sentinel-padded) array
        int u = root;
        while (u != -1) {
            push(u);
            int szL = getSz(pool[u].ch[0]);
            if (k < szL) u = pool[u].ch[0];
            else if (k == szL) { splay(u, g); return u; }
            else { k -= szL + 1; u = pool[u].ch[1]; }
        }
        return -1;
    }

    int buildRec(vector<T> &a, int l, int r) {
        if (l > r) return -1;
        int m = (l + r) / 2;
        int u = make(a[m]);
        pool[u].ch[0] = buildRec(a, l, m - 1);
        pool[u].ch[1] = buildRec(a, m + 1, r);
        if (pool[u].ch[0] != -1) pool[pool[u].ch[0]].p = u;
        if (pool[u].ch[1] != -1) pool[pool[u].ch[1]].p = u;
        pull(u);
        return u;
    }

    void build(vector<T> &a) {  // adds 2 internal sentinels around a
        n = (int)a.size();
        vector<T> b(n + 2);
        for (int i = 0; i < n; i++) b[i + 1] = a[i];
        root = buildRec(b, 0, n + 1);
    }

    int isolate(int lo, int hi) {  // 0-indexed inclusive; returns subtree root for [lo, hi]
        int L = findKth(lo);
        int R = findKth(hi + 2, L);
        return pool[R].ch[0];
    }

    void insert(int pos, T x) {  // 0-indexed, shifts [pos, n) right
        int L = findKth(pos);
        int R = findKth(pos + 1, L);
        int m = make(x);
        pool[m].p = R;
        pool[R].ch[0] = m;
        pull(R); pull(L);
        n++;
    }

    void erase(int pos) {  // 0-indexed
        int L = findKth(pos);
        int R = findKth(pos + 2, L);
        pool[R].ch[0] = -1;
        pull(R); pull(L);
        n--;
    }

    void reverse(int lo, int hi) { pool[isolate(lo, hi)].rev ^= 1; }
    void update(int lo, int hi, T mul, T add) { apply(isolate(lo, hi), mul, add); }
    T query(int lo, int hi) { return getAgg(isolate(lo, hi)); }
};
```

Splay Ordered Set

9171fab9

Order-statistics multiset via splay tree: insert/erase/find/rank/kth/predecessor/successor. T needs operator< and operator==. findKth/predecessor/successor are UB if called out of bounds (empty result) — guard with size() first. Multi-page exception to the line budget.

Time: O(\log n) amortized per operation. tested

```
template <typename T>
struct Splay {
    struct Node {
        int p = -1, ch[2] = {-1, -1}, sz = 1, cnt = 1;
        T val;
    };
    vector<Node> pool;
    int root = -1;

    int size() { return getSz(root); }
    int getSz(int n) { return n == -1 ? 0 : pool[n].sz; }

    void pull(int n) {
        if (n == -1) return;
        pool[n].sz = pool[n].cnt + getSz(pool[n].ch[0]) + getSz(pool[n].ch[1]);
    }

    int make(T x) { pool.push_back(Node()); pool.back().val = x; return (int)pool.size() - 1; }
    int dir(int n) { return pool[pool[n].p].ch[1] == n; }

    void rotate(int n) {
        int u = pool[n].p, v = pool[u].p, d = dir(n), w = pool[n].ch[d ^ 1];
        pool[u].ch[d] = w;
        if (w != -1) pool[w].p = u;
        pool[n].ch[d ^ 1] = u;
        pool[u].p = n;
        pool[n].p = v;
        if (v != -1) {
            if (pool[v].ch[0] == u) pool[v].ch[0] = n;
            else if (pool[v].ch[1] == u) pool[v].ch[1] = n;
        }
        pull(u); pull(n);
    }

    void splay(int n, int g = -1) {
        while (pool[n].p != g) {
            int u = pool[n].p, v = pool[u].p;
            if (v != g) {
                if (dir(n) == dir(u)) rotate(u);
                else rotate(n);
            }
            rotate(n);
        }
        if (g == -1) root = n;
    }

    int find(T x) {
        int n = root;
        while (n != -1) {
            if (pool[n].val == x) { splay(n); return n; }
            n = pool[n].ch[pool[n].val < x];
        }
        return -1;
    }

    void insert(T x) {
        if (root == -1) { root = make(x); return; }
        int n = root, f = -1;
        while (n != -1) {
            f = n;
            if (pool[n].val == x) { pool[n].cnt++; pull(n);
```

```
splay(n); return; }
            n = pool[n].ch[pool[n].val < x];
        }
        n = make(x);
        pool[f].ch[pool[f].val < x] = n;
        pool[n].p = f;
        splay(n);
    }

    void erase(T x) {
        int n = find(x);
        if (n == -1) return;
        if (pool[n].cnt > 1) { pool[n].cnt--; pull(n);
return; }

        if (pool[n].ch[0] == -1 && pool[n].ch[1] == -1)
{ root = -1; return; }
        if (pool[n].ch[0] == -1) { root = pool[n].ch[1];
pool[root].p = -1; return; }
        if (pool[n].ch[1] == -1) { root = pool[n].ch[0];
pool[root].p = -1; return; }

        int L = pool[n].ch[0], R = pool[n].ch[1];
        pool[L].p = pool[R].p = -1;
        root = L;
        int nrt = L;
        while (pool[nrt].ch[1] != -1) nrt = pool[nrt].ch[1];
        splay(nrt);
        pool[nrt].ch[1] = R;
        pool[R].p = nrt;
        pull(nrt);
    }

    int findRank(T x) {
        int n = root, res = 1;
        while (n != -1) {
            if (pool[n].val == x) { res +=
getSz(pool[n].ch[0]); splay(n); return res; }
            if (!(x < pool[n].val)) res += pool[n].cnt +
getSz(pool[n].ch[0]);
            n = pool[n].ch[pool[n].val < x];
        }
        return res;
    }

    T findKth(int k) {  // 1-indexed
        int n = root;
        while (n != -1) {
            int szL = getSz(pool[n].ch[0]);
            if (k <= szL) n = pool[n].ch[0];
            else {
                k -= szL;
                if (k <= pool[n].cnt) { splay(n); return
pool[n].val; }
                k -= pool[n].cnt; n = pool[n].ch[1];
            }
        }
        return T();  // out of bounds
    }

    T predecessor(T x) {
        int n = root, f = -1;
        while (n != -1) {
            if (pool[n].val < x) { f = n; n =
pool[n].ch[1]; }
            else n = pool[n].ch[0];
        }
        if (f != -1) splay(f);
        return pool[f].val;
    }

    T successor(T x) {
        int n = root, f = -1;
        while (n != -1) {
            if (x < pool[n].val) { f = n; n =
pool[n].ch[0]; }
            else n = pool[n].ch[1];
        }
        if (f != -1) splay(f);
        return pool[f].val;
    }
};
```

Sqrt Decomposition

2bd92d13

Sqrt decomposition over a static-size array: range add + range sum via per-block lazy tags and precomputed block sums. Simpler and more flexible than a segment tree when the op is awkward to express as a tree merge (e.g. per-element rebuild rules) — adapt the two inner loops for a different op.

Time: O(\sqrt n) per operation. tested

```
template <typename T>
struct SqrtDecomp {
    int n, blockSz;
    vector<T> a, blockSum, lazy;

    SqrtDecomp(vector<T> &arr) : n((int)arr.size()), a(arr)
{
        blockSz = (int)sqrt((double)n);
        if (blockSz < 1) blockSz = 1;  // note: max(1, ...)
mismatches under #define int long long,
                                       // since the literal
1 stays a real int — see CONVENTIONS.md
        int nb = (n + blockSz - 1) / blockSz;
        blockSum.assign(nb, T());
        lazy.assign(nb, T());
        for (int i = 0; i < n; i++) blockSum[i / blockSz] +=
a[i];
    }

    void update(int l, int r, T v) {  // inclusive [l, r]
        int bl = l / blockSz, br = r / blockSz;
        if (bl == br) {
            for (int i = l; i <= r; i++) a[i] += v;
            blockSum[bl] += v * (r - l + 1);
            return;
        }
        for (int i = l; i < (bl + 1) * blockSz; i++) a[i] +=
v;
        blockSum[bl] += v * ((bl + 1) * blockSz - l);
        for (int b = bl + 1; b < br; b++) lazy[b] += v;
        for (int i = br * blockSz; i <= r; i++) a[i] += v;
        blockSum[br] += v * (r - br * blockSz + 1);
    }

    T query(int l, int r) {  // inclusive [l, r]
        int bl = l / blockSz, br = r / blockSz;
        if (bl == br) {
            T res = T();
            for (int i = l; i <= r; i++) res += a[i] +
lazy[bl];
            return res;
        }
        T res = T();
        for (int i = l; i < (bl + 1) * blockSz; i++) res +=
a[i] + lazy[bl];
        for (int b = bl + 1; b < br; b++) res += blockSum[b]
+ lazy[b] * blockSz;
        for (int i = br * blockSz; i <= r; i++) res += a[i]
+ lazy[br];
        return res;
    }
};
```

Treap Implicit

c5616e40

Implicit (positional) treap: array-as-balanced-BST via randomized split/merge by subtree size. Supports insert-at-position, range reverse, and an "op" block (aggregate + lazy tag) you swap for the problem at hand — default is range-add + range-sum. To change the op: edit Node's val/agg/lazy fields and comb()/apply() only; split/merge/push/pull never need to change. T needs operator+ and a default zero. Multi-page exception to the line budget.

Time: O(\log n) expected per operation. tested

```
template <typename T>
struct ImplicitTreap {
    struct Node {
        // ---- op: edit for a different aggregate/lazy tag
----
        T val, agg;
        T lazy = T();
        // ---- end op ----
        int l = -1, r = -1, sz = 1, rev = 0;
    };
    vector<Node> pool;
    mt19937 rng{random_device{}()};
    int root = -1;

    T comb(T a, T b) { return a + b; }  // op: combine two
subtree aggregates
    void apply(int n, T v) { pool[n].val += v; pool[n].agg
+= v * pool[n].sz; pool[n].lazy += v; }  // op: push a lazy
tag onto n

    int getSz(int n) { return n == -1 ? 0 : pool[n].sz; }
    T getAgg(int n) { return n == -1 ? T() : pool[n].agg; }
    int make(T x) { pool.push_back({x, x}); return
(int)pool.size() - 1; }

    void pull(int n) {
        pool[n].sz = 1 + getSz(pool[n].l) +
getSz(pool[n].r);
        pool[n].agg = comb(comb(getAgg(pool[n].l),
pool[n].val), getAgg(pool[n].r));
    }

    void push(int n) {
        if (pool[n].rev) {
            swap(pool[n].l, pool[n].r);
            if (pool[n].l != -1) pool[pool[n].l].rev ^= 1;
            if (pool[n].r != -1) pool[pool[n].r].rev ^= 1;
            pool[n].rev = 0;
        }
        if (pool[n].lazy != T()) {
            if (pool[n].l != -1) apply(pool[n].l,
pool[n].lazy);
            if (pool[n].r != -1) apply(pool[n].r,
pool[n].lazy);
            pool[n].lazy = T();
        }
    }

    int merge(int l, int r) {
        if (l == -1) return r;
        if (r == -1) return l;
        push(l); push(r);
        if ((rng() % (unsigned)(getSz(l) + getSz(r))) <
(unsigned)getSz(l)) {
            pool[l].r = merge(pool[l].r, r);
            pull(l);
            return l;
        } else {
            pool[r].l = merge(l, pool[r].l);
            pull(r);
            return r;
        }
    }

    void split(int n, int k, int &l, int &r) {  // first k
elements go to l
        if (n == -1) { l = r = -1; return; }
        push(n);
        if (k <= getSz(pool[n].l)) {
            split(pool[n].l, k, l, pool[n].l);
            r = n;
        } else {
            split(pool[n].r, k - getSz(pool[n].l) - 1,
pool[n].r, r);
            l = n;
        }
        pull(n);
    }

    void insert(int pos, T x) {  // 0-indexed
        int l, r; split(root, pos, l, r);
        root = merge(merge(l, make(x)), r);
    }

    void reverse(int lo, int hi) {  // inclusive [lo, hi],
0-indexed
        int l, m, r;
        split(root, lo, l, m);
        split(m, hi - lo + 1, m, r);
        pool[m].rev ^= 1;
        root = merge(merge(l, m), r);
    }

    void update(int lo, int hi, T v) {  // inclusive [lo,
hi]
        int l, m, r;
        split(root, lo, l, m);
        split(m, hi - lo + 1, m, r);
        apply(m, v);
        root = merge(merge(l, m), r);
    }

    T query(int lo, int hi) {  // inclusive [lo, hi]
        int l, m, r;
        split(root, lo, l, m);
        split(m, hi - lo + 1, m, r);
        T res = getAgg(m);
        root = merge(merge(l, m), r);
        return res;
    }
};
```

Treap Ordered

2f4cc70b

Key-ordered treap: order-statistics multiset via insert/erase/rank/kth/predecessor/successor, using split-by-key + priority-based merge (no rotations, no parent pointers). T needs operator< and operator==. findKth/predecessor/successor are UB if called out of bounds — guard with size() first. Companion to treap-implicit.h (that one is positional/array-indexed; this one is key-ordered). Multi-page exception to the line budget.

Time: O(\log n) expected per operation. tested

```
template <typename T>
struct Treap {
    struct Node {
        T val;
        unsigned pri;
        int cnt = 1, sz = 1, l = -1, r = -1;
    };
    vector<Node> pool;
    mt19937 rng{random_device{}()};
    int root = -1;

    int size() { return getSz(root); }
    int getSz(int n) { return n == -1 ? 0 : pool[n].sz; }
    void pull(int n) { pool[n].sz = pool[n].cnt +
getSz(pool[n].l) + getSz(pool[n].r); }
    int make(T x) { pool.push_back({x, (unsigned)rng(), 1,
1, -1, -1}); return (int)pool.size() - 1; }

    int merge(int l, int r) {
        if (l == -1) return r;
        if (r == -1) return l;
        if (pool[l].pri > pool[r].pri) { pool[l].r =
merge(pool[l].r, r); pull(l); return l; }
        else { pool[r].l = merge(l, pool[r].l); pull(r);
return r; }
    }

    void split(int n, T key, int &l, int &r) {  // l: val <
key, r: val >= key
        if (n == -1) { l = r = -1; return; }
        if (pool[n].val < key) { split(pool[n].r, key,
pool[n].r, r); l = n; }
        else { split(pool[n].l, key, l, pool[n].l); r = n; }
```

```
        pull(n);
    }

    int insertRec(int n, int nn) {
        if (n == -1) return nn;
        if (pool[n].val == pool[nn].val) { pool[n].cnt++;
pull(n); return n; }
        if (pool[nn].pri > pool[n].pri) {
            split(n, pool[nn].val, pool[nn].l, pool[nn].r);
            pull(nn);
            return nn;
        }
        if (pool[nn].val < pool[n].val) pool[n].l =
insertRec(pool[n].l, nn);
        else pool[n].r = insertRec(pool[n].r, nn);
        pull(n);
        return n;
    }
    void insert(T x) { root = insertRec(root, make(x)); }

    int eraseRec(int n, T x) {
        if (n == -1) return -1;
        if (pool[n].val == x) {
            if (pool[n].cnt > 1) { pool[n].cnt--; pull(n);
return n; }
            return merge(pool[n].l, pool[n].r);
        }
        if (x < pool[n].val) pool[n].l = eraseRec(pool[n].l,
x);
        else pool[n].r = eraseRec(pool[n].r, x);
        pull(n);
        return n;
    }
    void erase(T x) { root = eraseRec(root, x); }

    bool contains(T x) {
        int n = root;
        while (n != -1) {
            if (pool[n].val == x) return true;
            n = pool[n].val < x ? pool[n].r : pool[n].l;
        }
        return false;
    }

    int findRank(T x) {
        int n = root, res = 1;
        while (n != -1) {
            if (pool[n].val == x) return res +
getSz(pool[n].l);
            if (!(x < pool[n].val)) res += pool[n].cnt +
getSz(pool[n].l);
            n = pool[n].val < x ? pool[n].r : pool[n].l;
        }
        return res;
    }

    T findKth(int k) {  // 1-indexed
        int n = root;
        while (n != -1) {
            int szL = getSz(pool[n].l);
            if (k <= szL) n = pool[n].l;
            else if (k <= szL + pool[n].cnt) return
pool[n].val;
            else { k -= szL + pool[n].cnt; n = pool[n].r; }
        }
        return T();
    }

    T predecessor(T x) {
        int n = root, f = -1;
        while (n != -1) {
            if (pool[n].val < x) { f = n; n = pool[n].r; }
            else n = pool[n].l;
        }
        return pool[f].val;
    }
    T successor(T x) {
        int n = root, f = -1;
        while (n != -1) {
            if (x < pool[n].val) { f = n; n = pool[n].l; }
            else n = pool[n].r;
        }
        return pool[f].val;
    }
};
```

Xor Trie

b2a92c88

Binary trie over BITS-bit integers, multiset semantics via ref-counted soft erase (no physical deletion). query(x) needs >=1 element currently inserted; insert(0) once up front as a sentinel if the set might otherwise be empty.

Time: O(BITS) per operation. tested

```
template <int BITS = 31>
struct XorTrie {
    struct Node {
        int cnt = 0, ch[2] = {-1, -1};
    };
    vector<Node> pool{Node()};  // pool[0] is the root

    void insert(int x) {
        int at = 0;
        pool[at].cnt++;
        for (int i = BITS - 1; i >= 0; i--) {
            int b = (x >> i) & 1;
            if (pool[at].ch[b] == -1) { pool[at].ch[b] =
pool.size(); pool.push_back(Node()); }
            at = pool[at].ch[b];
            pool[at].cnt++;
        }
    }

    void erase(int x) {  // x must currently be present
        int at = 0;
        pool[at].cnt--;
        for (int i = BITS - 1; i >= 0; i--) {
            at = pool[at].ch[(x >> i) & 1];
            pool[at].cnt--;
        }
    }

    int query(int x) {  // max (x ^ elem) over currently
inserted elements
        int at = 0, res = 0;
        for (int i = BITS - 1; i >= 0; i--) {
            int b = (x >> i) & 1, want = b ^ 1;
            if (pool[at].ch[want] != -1 &&
pool[pool[at].ch[want]].cnt > 0) {
                res |= (1 << i);
                at = pool[at].ch[want];
            } else {
                at = pool[at].ch[b];
            }
        }
        return res;
    }
};
```

Graphs

Bipartite Check

0aa26d17

Bipartiteness check via BFS 2-coloring, handles disconnected graphs. Returns {color, ok}; color[i] is 1 or -1 when ok, undefined otherwise.

Time: O(V + E). tested

```
pair<vector<int>, bool> bipartiteCheck(int n,
vector<vector<int>> &adj) {
    vector<int> color(n, 0);
    for (int s = 0; s < n; s++) {
        if (color[s]) continue;
        color[s] = 1;
        queue<int> q; q.push(s);
        while (!q.empty()) {
```

```
            int u = q.front(); q.pop();
            for (int v : adj[u]) {
                if (!color[v]) { color[v] = -color[u];
q.push(v); }
                else if (color[v] == color[u]) return {{},
false};
            }
        }
    }
    return {color, true};
}
```

Bridges

39a7ffb5

Tarjan bridge-finding via tin/low DFS, handles disconnected graphs. Returns the list of bridge edge indices, given adj[u] = {(v, edgeIndex), ...} listing both directions of each edge.

Time: O(V + E). tested

```
vector<int> findBridges(int n, vector<vector<pair<int,
int>>> &adj) {
    vector<int> tin(n, -1), low(n), bridges;
    int timer = 0;
    auto dfs = [&](auto &&dfs, int u, int pe) -> void {
        tin[u] = low[u] = timer++;
        for (auto &[v, id] : adj[u]) {
            if (id == pe) continue;
            if (tin[v] != -1) {
                low[u] = min(low[u], tin[v]);
            } else {
                dfs(dfs, v, id);
                low[u] = min(low[u], low[v]);
                if (low[v] > tin[u]) bridges.push_back(id);
            }
        }
    };
    for (int i = 0; i < n; i++) if (tin[i] == -1) dfs(dfs,
i, -1);
    return bridges;
}
```

Centroid Decomposition

de137c00

Centroid decomposition solving "min #edges among paths of length exactly targetLen" (IOI Race). Shows the general pattern (subtree size -> find centroid -> combine paths through the centroid, one child branch at a time so two halves from the same subtree never combine -> recurse on the remaining pieces); adapt solve()'s merge step for a different centroid problem, subtreeSize()/findCentroid() never need to change.

Time: O(n \log n). tested

```
struct CentroidDecomp {
    int n;
    ll targetLen;
    vector<vector<pair<int, ll>>> adj;
    vector<bool> alive;
    vector<int> sz;

    CentroidDecomp(int n, ll targetLen,
vector<vector<pair<int, ll>>> &adj)
        : n(n), targetLen(targetLen), adj(adj), alive(n,
true), sz(n) {}

    int subtreeSize(int u, int p) {
        sz[u] = 1;
        for (auto &[v, w] : adj[u])
            if (v != p && alive[v]) sz[u] += subtreeSize(v,
u);
        return sz[u];
    }

    int findCentroid(int target, int u, int p) {
        for (auto &[v, w] : adj[u])
            if (v != p && alive[v] && 2 * sz[v] > target)
return findCentroid(target, v, u);
        return u;
    }

    void collect(vector<pair<ll, int>> &out, ll dist, int
edges, int u, int p) {
        if (dist > targetLen) return;
        out.push_back({dist, edges});
        for (auto &[v, w] : adj[u])
            if (v != p && alive[v]) collect(out, dist + w,
edges + 1, v, u);
    }

    int solve(int u) {
        subtreeSize(u, -1);
        int cen = findCentroid(sz[u], u, -1);
        alive[cen] = false;

        int res = INT_MAX;
        unordered_map<ll, int> best;
        best[0] = 0;
        for (auto &[v, w] : adj[cen]) {
            if (!alive[v]) continue;
            vector<pair<ll, int>> branch;
            collect(branch, w, 1, v, cen);
            for (auto &[d, c] : branch)
                if (targetLen - d >= 0 &&
best.count(targetLen - d)) res = min(res, best[targetLen -
d] + c);
            for (auto &[d, c] : branch) {
                auto it = best.find(d);
                if (it == best.end() || it->second > c)
best[d] = c;
            }
        }

        for (auto &[v, w] : adj[cen]) if (alive[v]) res =
min(res, solve(v));
        return res;
    }
};
```

Dijkstra

7695da1b

Dijkstra's algorithm on a non-negative-weight directed graph given as an adjacency list. W needs operator+ and operator<.

Time: O((V + E) \log V). tested

```
template <typename W>
vector<W> dijkstra(int n, vector<vector<pair<int, W>>> &adj,
int src, W INF) {
    vector<W> dist(n, INF);
    priority_queue<pair<W, int>, vector<pair<W, int>>,
greater<>> pq;
    dist[src] = W();
    pq.push({dist[src], src});
    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;
        for (auto &[v, w] : adj[u]) {
            if (d + w < dist[v]) {
                dist[v] = d + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

Dsu

191d69c2

Disjoint set union with path compression + union by size.

Time: O(\alpha(n)) amortized per operation. tested

```
struct DSU {
    vector<int> par, sz;

    DSU(int n) : par(n), sz(n, 1) { iota(par.begin(),
par.end(), 0); }

    int find(int u) { return par[u] == u ? u : par[u] =
find(par[u]); }
```

```
    bool unite(int u, int v) {
        u = find(u), v = find(v);
        if (u == v) return false;
        if (sz[u] < sz[v]) swap(u, v);
        par[v] = u;
        sz[u] += sz[v];
        return true;
    }
};
```

Eulerian Circuit

5074d807

Hierholzer's algorithm for an Eulerian circuit on an undirected multigraph, starting/ending at node 0. adj[u] = {(v, edgeIndex), ...} with both directions of each edge listed (edgeIndex shared by both); mutated in place. Returns {circuit, ok}. For a directed graph: check in-degree == out-degree per node instead of even degree, everything else is the same.

Time: O(V + E). tested

```
pair<vector<int>, bool> eulerianCircuit(int n, int m,
vector<vector<pair<int, int>>> &adj) {
    for (int u = 0; u < n; u++) if ((int)adj[u].size() % 2)
return {{}, false};

    vector<int> path;
    vector<bool> vis(m);
    auto dfs = [&](auto &&dfs, int u) -> void {
        while (!adj[u].empty()) {
            auto [v, i] = adj[u].back(); adj[u].pop_back();
            if (vis[i]) continue;
            vis[i] = true;
            dfs(dfs, v);
        }
        path.push_back(u);
    };
    dfs(dfs, 0);

    if ((int)path.size() != m + 1) return {{}, false};
    return {path, true};
}
```

Floyd Warshall

bd8743ee

Floyd-Warshall all-pairs shortest paths on a dense distance matrix (dist[i][j] = INF if no edge, dist[i][i] = 0). Mutates dist in place. Negative-cycle

Time: O(V^3). tested

```
template <typename W>
void floydWarshall(vector<vector<W>> &dist, W INF) {
    int n = (int)dist.size();
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (dist[i][k] < INF && dist[k][j] <
INF)  // guards INF+INF overflow
                    dist[i][j] = min(dist[i][j], dist[i][k]
+ dist[k][j]);
}
```

Floyd Warshall Incremental

021d326d

Add a single edge (u, v, w) to an all-pairs-shortest-path matrix already computed by floyd-warshall.h, in O(n^2) instead of recomputing full O(n^3) Floyd-Warshall. Treats the new edge as one extra intermediate relaxation, same trick as the k-loop in Floyd-Warshall itself. Call once per edge added, in any order.

Time: O(n^2) per edge added. tested

```
template <typename W>
void addEdgeFW(vector<vector<W>> &dist, int u, int v, W w) {
    int n = (int)dist.size();
    dist[u][v] = min(dist[u][v], w);
    dist[v][u] = dist[u][v];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dist[i][j] = min({dist[i][j], dist[i][u] +
dist[u][v] + dist[v][j], dist[i][v] + dist[v][u] + dist[u]
[j]});
}
```

Heavy Light Decomposition

312fcf6e

Heavy-light decomposition: flattens a tree into O(\log n) contiguous ranges per root-to-node path. Pair with any segment tree (segment-tree-lazy.h, segment-tree-iterative.h, ...) keyed by pos[]; forPath(u, v, f) calls f(l, r) for each maximal contiguous [l, r] range (in pos[] order, both inclusive) along the path u-v; forSubtree(u) gives the single [l, r] covering u's subtree. Recursive dfsSize/ dfsDecompose; may need a larger stack for very deep/ skewed trees.

Time: O(n) build; a path decomposes into O(\log n) ranges. tested

```
struct HLD {
    int n, timer = 0;
    vector<vector<int>> adj;
    vector<int> par, dep, heavy, head, pos, sz;

    HLD(int n, vector<vector<int>> &adj, int root = 0)
        : n(n), adj(adj), par(n, -1), dep(n), heavy(n, -1),
head(n), pos(n), sz(n) {
        dfsSize(root, -1);
        head[root] = root;
        dfsDecompose(root, -1);
    }

    int dfsSize(int u, int p) {
        par[u] = p; sz[u] = 1;
        int maxSz = 0;
        for (int v : adj[u]) {
            if (v == p) continue;
            dep[v] = dep[u] + 1;
            sz[u] += dfsSize(v, u);
            if (sz[v] > maxSz) { maxSz = sz[v]; heavy[u] =
v; }
        }
        return sz[u];
    }

    void dfsDecompose(int u, int p) {
        pos[u] = timer++;
        if (heavy[u] != -1) {
            head[heavy[u]] = head[u];
            dfsDecompose(heavy[u], u);
        }
        for (int v : adj[u]) {
            if (v == p || v == heavy[u]) continue;
            head[v] = v;
            dfsDecompose(v, u);
        }
    }

    void forPath(int u, int v, function<void(int, int)> f) {
        while (head[u] != head[v]) {
            if (dep[head[u]] < dep[head[v]]) swap(u, v);
            f(pos[head[u]], pos[u]);
            u = par[head[u]];
        }
        if (dep[u] > dep[v]) swap(u, v);
        f(pos[u], pos[v]);
    }

    pair<int, int> forSubtree(int u) { return {pos[u],
pos[u] + sz[u] - 1}; }

    int lca(int u, int v) {
        while (head[u] != head[v]) {
            if (dep[head[u]] < dep[head[v]]) swap(u, v);
            u = par[head[u]];
        }
        return dep[u] < dep[v] ? u : v;
    }
};
```

Lca Binary Lifting

3615c86b

Lowest common ancestor via Euler tour (tin/tout) + binary lifting. Recursive Euler tour; may need a larger stack for very deep/skewed trees.

Time: O(n \log n) build, O(\log n) per query. tested

```
struct LCA {
    int timer = 0, n, LOGN;
    vector<int> tin, tout;
    vector<vector<int>> up;

    LCA(int n, vector<vector<int>> &adj, int root = 0) :
n(n) {
        LOGN = n > 1 ? (int)ceil(log2(n)) : 1;
        tin.resize(n);
        tout.resize(n);
        up.assign(n, vector<int>(LOGN + 1));
        dfs(root, root, adj);
    }

    void dfs(int u, int p, vector<vector<int>> &adj) {
        tin[u] = timer++;
        up[u][0] = p;
        for (int i = 1; i <= LOGN; i++) up[u][i] = up[up[u]
[i - 1]][i - 1];
        for (int v : adj[u]) if (v != p) dfs(v, u, adj);
        tout[u] = timer - 1;
    }

    bool isAncestor(int u, int v) { return tin[u] <= tin[v]
&& tout[v] <= tout[u]; }

    int lca(int u, int v) {
        if (isAncestor(u, v)) return u;
        if (isAncestor(v, u)) return v;
        for (int i = LOGN; i >= 0; i--)
            if (!isAncestor(up[u][i], v)) u = up[u][i];
        return up[u][0];
    }
};
```

Lex Smallest Path

3174cc67

Lexicographically smallest walk from x to y in an unweighted graph: greedily extend to the smallest-labeled unvisited neighbor that still has a path to y avoiding already-visited nodes.

Time: O(V(V + E)) worst case (one reachability BFS per step taken). tested

```
vector<int> lexSmallestPath(int n, vector<vector<int>> &adj,
int x, int y) {
    for (auto &nbrs : adj) sort(nbrs.begin(), nbrs.end());
    vector<bool> vis(n, false);
    vector<int> path = {x};
    vis[x] = true;
    int at = x;

    auto reachable = [&](int st, int en) {
        vector<bool> vis2(n, false);
        queue<int> q;
        vis2[st] = true;
        q.push(st);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (u == en) return true;
            for (int v : adj[u]) if (!vis[v] && !vis2[v])
{ vis2[v] = true; q.push(v); }
        }
        return false;
    };

    while (at != y) {
        for (int v : adj[at]) {
            if (!vis[v] && reachable(v, y)) {
                vis[v] = true;
                path.push_back(v);
                at = v;
                break;
            }
        }
    }
    return path;
}
```

Small To Large

108ae9a3

Small-to-large merging ("DSU on tree"): merge a child's set into the larger of the two and re-insert the smaller one's elements, giving O(n log^2 n) total instead of O(n^2) for per-node subtree aggregate queries. Worked example: count of distinct colors in each node's subtree, given colors a[] and adjacency adj (rooted at 0).

Time: O(n \log^2 n). tested

```
vector<int> distinctColorCounts(int n, vector<int> &a,
vector<vector<int>> &adj) {
    vector<set<int>> col(n);
    vector<int> ans(n);
    auto dfs = [&](auto &&dfs, int u, int p) -> void {
        for (int v : adj[u]) {
            if (v == p) continue;
            dfs(dfs, v, u);
            if (col[u].size() < col[v].size()) swap(col[u],
col[v]);
            for (int x : col[v]) col[u].insert(x);
        }
        col[u].insert(a[u]);
        ans[u] = (int)col[u].size();
    };
    dfs(dfs, 0, -1);
    return ans;
}
```

Topo Sort

1db4c9c1

DFS topological sort with cycle detection on a directed graph, handles disconnected graphs. Returns {order, ok}; ok is false if the graph has a cycle (order is invalid in that case). Longest path in a DAG: relax edges once per node, in order.

Time: O(V + E). tested

```
pair<vector<int>, bool> topoSort(int n, vector<vector<int>>
&adj) {
    vector<int> vis(n, 0), order;  // 0 = unvisited, 1 = on
stack, 2 = done
    bool ok = true;
    auto dfs = [&](auto &&dfs, int u) -> void {
        vis[u] = 1;
        for (int v : adj[u]) {
            if (vis[v] == 1) ok = false;
            else if (!vis[v]) dfs(dfs, v);
        }
        vis[u] = 2;
        order.push_back(u);
    };
    for (int i = 0; i < n; i++) if (!vis[i]) dfs(dfs, i);
    reverse(order.begin(), order.end());
    return {order, ok};
}
```

Tree Center

a515b4b4

Tree center(s) via leaf-peeling (repeatedly strip degree-1 nodes). Returns 1 node if the tree's diameter is even, 2 adjacent nodes if odd.

Time: O(n). tested

```
vector<int> treeCenters(vector<vector<int>> &adj) {
    int n = (int)adj.size();
    vector<int> degree(n);
    queue<int> leaves;
    for (int i = 0; i < n; i++) {
        degree[i] = (int)adj[i].size();
        if (degree[i] <= 1) leaves.push(i);
    }

    int left = n;
    while (left > 2) {
        int cnt = (int)leaves.size();
```

```
        left -= cnt;
        for (int i = 0; i < cnt; i++) {
            int leaf = leaves.front(); leaves.pop();
            for (int v : adj[leaf]) {
                degree[v]--;
                if (degree[v] == 1) leaves.push(v);
            }
        }
    }

    vector<int> centers;
    while (!leaves.empty())
{ centers.push_back(leaves.front()); leaves.pop(); }
    return centers;
}
```

Dynamic Programming

Bitset Subset Sum

59cda75d

Subset-sum reachability via bitset shifts: which totals in [0, cap] are achievable using a subset of a[]. Each item processed in O(cap / 64) instead of O(cap) — the same shift trick generalizes to 2D reachability (track a bitset per value of a second resource, shift each one) when there are two constrained sums, as in the classic "knapsack with two capacities" variant.

Time: O(n \cdot cap / 64). tested

```
template <size_t CAP>
bitset<CAP> subsetSumReachable(vector<int> &a) {
    bitset<CAP> reachable;
    reachable[0] = 1;
    for (int x : a) reachable |= reachable << x;
    return reachable;
}
```

Digit Dp

a2d99e57

Digit DP skeleton: count integers in [0, x] satisfying some digit-local constraint. State is [pos, extra, tight, leadingZero]; extra is problem-specific side state (e.g. "previous digit" for a no-adjacent-equal-digits constraint, sized here as an int up to 11 — 0-9 plus a 10 = "no previous digit yet" sentinel — adjust EXTRA and the transition condition for your problem). tight means the prefix so far equals x's prefix exactly (so the next digit is capped); leadingZero means every digit so far has been a leading zero (not yet "really started"). countUpTo(r) - countUpTo(l-1) gives [l, r]; countUpTo(-1) = 0 by convention (call with l-1 possibly negative).

Time: O(digits \times EXTRA \times 2 \times 2 \times 10) states. tested

```
struct DigitDP {
    static constexpr int EXTRA = 11;  // problem-specific
side-state cardinality
    vector<int> d;
    ll dp[20][EXTRA][2][2];
    bool vis[20][EXTRA][2][2];

    ll calc(int pos, int extra, bool tight, bool
leadingZero) {
        if (pos == (int)d.size()) return 1;
        if (vis[pos][extra][tight][leadingZero]) return
dp[pos][extra][tight][leadingZero];
        vis[pos][extra][tight][leadingZero] = true;

        int ub = tight ? d[pos] : 9;
        ll res = 0;
        for (int c = 0; c <= ub; c++) {
            if (leadingZero && c == 0) {
                res += calc(pos + 1, EXTRA - 1, tight && c
== d[pos], true);
            } else if (c != extra) {  // op: the actual
digit-local constraint goes here
                res += calc(pos + 1, c, tight && c ==
d[pos], false);
            }
        }
        return dp[pos][extra][tight][leadingZero] = res;
    }

    ll countUpTo(ll x) {
        if (x < 0) return 0;
        memset(dp, 0, sizeof(dp));
        memset(vis, 0, sizeof(vis));
        d.clear();
        if (x == 0) return 1;
        while (x) { d.push_back((int)(x % 10)); x /= 10; }
        reverse(d.begin(), d.end());
        return calc(0, EXTRA - 1, true, true);
    }
};
```

Edit Distance

370fe1ee

Levenshtein edit distance (insert/delete/substitute, unit cost) via top-down memoization. For |edit distance| <= k specifically, edit-distance-bounded.h is faster (O(nk) instead of O(nm)).

Time: O(nm). tested

```
int editDistanceRec(string &s, string &t, int n, int m,
vector<vector<int>> &dp) {
    if (n == 0) return m;
    if (m == 0) return n;
    if (dp[n][m] != -1) return dp[n][m];
    if (s[n - 1] == t[m - 1]) return dp[n][m] =
editDistanceRec(s, t, n - 1, m - 1, dp);
    return dp[n][m] = 1 + min({
        editDistanceRec(s, t, n - 1, m, dp),
        editDistanceRec(s, t, n, m - 1, dp),
        editDistanceRec(s, t, n - 1, m - 1, dp)
    });
}

int editDistance(string &s, string &t) {
    int n = (int)s.size(), m = (int)t.size();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, -1));
    return editDistanceRec(s, t, n, m, dp);
}
```

Edit Distance Bounded

870c9d9c

"Is edit distance <= k?" in O(nk) instead of O(nm), by restricting the DP to the diagonal band |i - j| <= k (any optimal alignment outside the band would need > k edits already). Returns true iff editDistance(s, t) <= k.

Time: O(nk). tested

```
bool editDistanceWithinK(string &s, string &t, int k) {
    int n = (int)s.size(), m = (int)t.size();
    if (abs(n - m) > k) return false;
    const int BIG = k + 1;

    vector<int> prev(m + 1), curr(m + 1);
    for (int j = 0; j <= m; j++) prev[j] = (j <= k ? j :
BIG);

    for (int i = 1; i <= n; i++) {
        int lo = max((ll)1, i - k), hi = min((ll)m, i + k);
        curr[0] = (i <= k ? i : BIG);
        // reset curr[lo-1] BEFORE the loop: j=lo reads
curr[j-1], and curr is reused
        // (ping-ponged with prev) across rounds, so it
otherwise holds a stale value
        // from 2 rounds ago instead of "out of band" — this
was a real bug, caught by
        // stress-testing against editDistance() on a random
case.
        if (lo > 1) curr[lo - 1] = BIG;
        for (int j = lo; j <= hi; j++) {
            int cost = (s[i - 1] == t[j - 1] ? 0 : 1);
            int ins = curr[j - 1] + 1, del = prev[j] + 1,
sub = prev[j - 1] + cost;
            curr[j] = min({ins, del, sub});
        }
        if (hi < m) curr[hi + 1] = BIG;
        swap(prev, curr);
    }

    return prev[m] <= k;
}
```

Knapsack 01 ab5927fc

0/1 knapsack (each item used at most once), 1D rolling array. The classic bug: capacity loop must go high-to-low, or an item gets reused within the same iteration (that's the unbounded variant, knapsack-unbounded.h).

Time: O(nW). tested

```
ll knapsack01(vector<ll> &w, vector<ll> &v, ll cap) {
    int n = (int)w.size();
    vector<ll> dp(cap + 1, -1);
    dp[0] = 0;
    for (int i = 0; i < n; i++)
        for (ll j = cap; j >= w[i]; j--)
            if (dp[j - w[i]] != -1) dp[j] = max(dp[j], dp[j
- w[i]] + v[i]);

    ll ans = 0;
    for (ll j = 0; j <= cap; j++) ans = max(ans, dp[j]);
    return ans;
}
```

Knapsack Unbounded 980f153e

Unbounded knapsack / coin-change count-of-ways (each item usable any number of times), 1D rolling array. Capacity loop goes low-to-high — the opposite direction from knapsack-01.h, which is exactly why that direction matters.

Time: O(n \cdot target). tested

```
ll coinWays(vector<ll> &coins, ll target, ll MOD) {
    vector<ll> dp(target + 1, 0);
    dp[0] = 1;
    for (ll c : coins)
        for (ll w = c; w <= target; w++) dp[w] = (dp[w] +
dp[w - c]) % MOD;
    return dp[target];
}
```

Lis 5c58d50e

Longest strictly-increasing subsequence via patience sorting, with reconstruction. For longest non-decreasing, change the lower_bound to upper_bound.

Time: O(n \log n). tested

```
template <typename T>
vector<T> lis(vector<T> &a) {
    int n = (int)a.size();
    vector<T> tails;                // tails[i] = smallest
possible tail value of an increasing subsequence of length
i+1
    vector<int> tailIdx, par(n, -1);
    for (int i = 0; i < n; i++) {
        int p = (int)(lower_bound(tails.begin(),
tails.end(), a[i]) - tails.begin());
        if (p == (int)tails.size()) { tails.push_back(a[i]);
tailIdx.push_back(i); }
        else { tails[p] = a[i]; tailIdx[p] = i; }
        par[i] = (p > 0 ? tailIdx[p - 1] : -1);
    }

    vector<T> res;
    int at = tailIdx.empty() ? -1 : tailIdx.back();
    while (at != -1) { res.push_back(a[at]); at = par[at]; }
    reverse(res.begin(), res.end());
    return res;
}
```

Strings

Aho Corasick b60006fe

Aho-Corasick automaton over lowercase letters: add() each pattern (id < 32, tracked via bitmask), then build(). After build(), next[c] is a full O(1) transition table (no need to follow fail links while matching); out is a bitmask of pattern ids ending at this state or any of its suffix-link ancestors.

Time: O(sum(|pattern|) \cdot 26) build; O(|text|) to run. tested

```
struct AhoCorasick {
    struct Node {
        int next[26], link = 0, out = 0;
        Node() { fill_n(next, 26, -1); }
    };
    vector<Node> t;
    AhoCorasick() { t.emplace_back(); }

    void add(string &s, int id) {
        int at = 0;
        for (char ch : s) {
            int c = ch - 'a';
            if (t[at].next[c] == -1) { t[at].next[c] =
(int)t.size(); t.emplace_back(); }
            at = t[at].next[c];
        }
        t[at].out |= (1 << id);
    }

    void build() {
        queue<int> q;
        for (int c = 0; c < 26; c++) {
            int at = t[0].next[c];
            if (at != -1) { t[at].link = 0; q.push(at); }
            else t[0].next[c] = 0;
        }
        while (!q.empty()) {
            int at = q.front(); q.pop();
            t[at].out |= t[t[at].link].out;
            for (int c = 0; c < 26; c++) {
                int to = t[at].next[c];
                if (to != -1) { t[to].link =
t[t[at].link].next[c]; q.push(to); }
                else t[at].next[c] = t[t[at].link].next[c];
            }
        }
    }
};
```

Kmp 0b220928

KMP prefix function pi[i] = length of the longest proper prefix of s[0..i] that is also a suffix. findOccurrences(text, pattern) finds every occurrence of pattern in text via pi over pattern + '#' + text (# must not appear in either).

Time: O(n). tested

```
vector<int> prefixFunction(string &s) {
    int n = (int)s.size();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    return pi;
}

vector<int> findOccurrences(string &text, string &pattern) {
    string s = pattern + '#' + text;
    auto pi = prefixFunction(s);
    int m = (int)pattern.size();
    vector<int> res;
    for (int i = m + 1; i < (int)s.size(); i++)
        if (pi[i] == m) res.push_back(i - 2 * m);
    return res;
}
```

Manacher 4b8194e7

Manacher's algorithm: longest palindromic substring centered at every position, both odd and even length, in linear time. d1[i] = radius of the longest odd-length palindrome centered at i (length 2*d1[i]-1); d2[i] = radius of the longest even-length palindrome centered between i-1 and i (length 2*d2[i]).

Time: O(n). tested

```
pair<vector<int>, vector<int>> manacher(string &s) {
    int n = (int)s.size();
    vector<int> d1(n), d2(n);
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
        while (i - k >= 0 && i + k < n && s[i - k] == s[i +
k]) k++;
        d1[i] = k;
        if (i + k - 1 > r) { l = i - k + 1; r = i + k - 1; }
    }
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i +
1);
        while (i - k - 1 >= 0 && i + k < n && s[i - k - 1]
== s[i + k]) k++;
        d2[i] = k;
        if (i + k - 1 > r) { l = i - k; r = i + k - 1; }
    }
    return {d1, d2};
}
```

String Hashing

7f2ead7c

Polynomial string hashing, arithmetic mod 2^64-1 (not mod 2^64 — plain mod-2^64 overflow hashing is broken by well-known adversarial test data, e.g. Thue-Morse strings, and mod-1e9+7-style double hashing is routinely hacked when the mod/base are common template defaults). H wraps the mod-(2^64-1) arithmetic; compare hashes via == and <. HashInterval(s).hashInterval(a, b) hashes s[a, b). Source: kactl (CC0), see NOTICE.md.

Time: O(n) build, O(1) per interval hash. kactl-derived, tested

```
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 = (ll)1e11 + 3;  // order ~3e9; a random
base also works

struct HashInterval {
    vector<H> ha, pw;
    HashInterval(string &s) : ha(s.size() + 1), pw(ha) {
        pw[0] = 1;
        for (int i = 0; i < (int)s.size(); i++) {
            ha[i + 1] = ha[i] * C + s[i];
            pw[i + 1] = pw[i] * C;
        }
    }
    H hashInterval(int a, int b) { return ha[b] - ha[a] *
pw[b - a]; }  // hash s[a, b)
};

vector<H> slidingHashes(string &s, int length) {
    if ((int)s.size() < length) return {};
    H h = 0, pw = 1;
    for (int i = 0; i < length; i++) { h = h * C + s[i]; pw
= pw * C; }
    vector<H> res = {h};
    for (int i = length; i < (int)s.size(); i++)
res.push_back(h = h * C + s[i] - pw * s[i - length]);
    return res;
}
```

Suffix Array

1a145bb6

Suffix array via radix-sort doubling, with the matching O(n) LCP pass woven into the same construction (kactl's SuffixArray). sa[i] is the starting index of the i'th suffix in sorted order; sa.size() == s.size()+1 and sa[0] == s.size() (the appended sentinel, sorts first). lcp[i] = longest common prefix of sa[i] and sa[i-1] (lcp[0] = 0). s must not contain a NUL character. Source: kactl (CC0), see NOTICE.md.

Time: O(n \log n). kactl-derived, tested

```
struct SuffixArray {
    vector<int> sa, lcp;

    SuffixArray(string s, int lim = 256) {
        s.push_back(0);
        int n = (int)s.size(), k = 0, a, b;
        vector<int> x(s.begin(), s.end()), y(n), ws(max(n,
lim));
        sa = lcp = y;
        iota(sa.begin(), sa.end(), 0);
        for (int j = 0, p = 0; p < n; j = max((ll)1, j * 2),
lim = p) {
            p = j;
            iota(y.begin(), y.end(), n - j);
            rep(i, 0, n) if (sa[i] >= j) y[p++] = sa[i] - j;
            fill(ws.begin(), ws.end(), 0);
            rep(i, 0, n) ws[x[i]]++;
            rep(i, 1, lim) ws[i] += ws[i - 1];
            for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
            swap(x, y);
            p = 1;
            x[sa[0]] = 0;
            rep(i, 1, n) {
                a = sa[i - 1], b = sa[i];
                x[b] = (y[a] == y[b] && y[a + j] == y[b +
j]) ? p - 1 : p++;
            }
        }
        for (int i = 0, j; i < n - 1; lcp[x[i++]] = k)
            for (k && k--, j = sa[x[i] - 1]; s[i + k] == s[j
+ k]; k++);
    }
};
```

Z Function

db694547

Z-function z[i] = length of the longest common prefix of s and s[i..]. z[0] is conventionally left 0 (undefined/unused). For substring search, same trick as KMP: run on pattern + '#' + text and scan for z[i] == |pattern|.

Time: O(n). tested

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

Math

Euler Phi

a1aaf3e4

Euler's totient function phi(n) (count of integers in [1,n] coprime to n), single-query via trial division, or a sieve for all of [1, maxn).

Time: phi() O(\sqrt n); eulerPhiSieve() O(n \log\log n). tested

```
ll phi(ll n) {
    ll ans = n;
    for (ll i = 2; i * i <= n; i++) {
        if (n % i) continue;
        while (n % i == 0) n /= i;
        ans -= ans / i;
    }
    if (n > 1) ans -= ans / n;
    return ans;
}

vector<ll> eulerPhiSieve(int maxn) {
```

```
    vector<ll> phi(maxn);
    iota(phi.begin(), phi.end(), 0);
    for (int i = 2; i < maxn; i++)
        if (phi[i] == i)  // i is prime
            for (int j = i; j < maxn; j += i) phi[j] -=
phi[j] / i;
    return phi;
}
```

Fft

e23a0cfd

Fast Fourier transform (in-place, iterative) and real-valued polynomial multiplication via the trick of packing two real sequences into one complex FFT.

Time: O(n \log n). kactl-derived, tested

```
typedef complex<double> C;
typedef vector<double> vd;

void fft(vector<C> &a) {
    int n = (int)a.size(), L = 31 - __builtin_clz(n);
    static vector<complex<long double>> R(2, 1);
    static vector<C> rt(2, 1);
    for (static int k = 2; k < n; k *= 2) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0L, acos(-1.0L) / k);
        for (int i = k; i < 2 * k; i++) rt[i] = R[i] = i &
1 ? R[i / 2] * x : R[i / 2];
    }
    vector<int> rev(n);
    for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] | (i &
1) << L) / 2;
    for (int i = 0; i < n; i++) if (i < rev[i]) swap(a[i],
a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k)
            for (int j = 0; j < k; j++) {
                C z = rt[j + k] * a[i + j + k];
                a[i + j + k] = a[i + j] - z;
                a[i + j] += z;
            }
}

vd conv(const vd &a, const vd &b) {
    if (a.empty() || b.empty()) return {};
    vd res((int)a.size() + (int)b.size() - 1);
    int L = 32 - __builtin_clz((int)res.size()), n = 1 << L;
    vector<C> in(n), out(n);
    copy(a.begin(), a.end(), in.begin());
    for (int i = 0; i < (int)b.size(); i++)
in[i].imag(b[i]);
    fft(in);
    for (C &x : in) x *= x;
    for (int i = 0; i < n; i++) out[i] = in[-i & (n - 1)] -
conj(in[i]);
    fft(out);
    for (int i = 0; i < (int)res.size(); i++) res[i] =
imag(out[i]) / (4 * n);
    return res;
}
```

Floor Sum Block Trick

9f51b00b

Block/quotient iteration: floor(n/i) takes O(\sqrt n) distinct values as i ranges over [1, n], each held over a contiguous block [i, floor(n / floor(n/i))]. Iterate blocks instead of individual i whenever summing something that depends only on floor(n/i) — this file shows the idiom computing Sum_{i=1}^{n} floor(n/i) mod MOD.

Time: O(\sqrt n). tested

```
ll floorSum(ll n, ll MOD) {
    ll ans = 0;
    for (ll i = 1, hi; i <= n; i = hi + 1) {
        ll q = n / i;
        hi = n / q;
        ll cnt = (hi - i + 1) % MOD;
        ans = (ans + cnt * (q % MOD)) % MOD;
    }
    return ans;
}
```

Inclusion Exclusion

b16dc065

Bitmask inclusion-exclusion: count of integers in [1, x] divisible by at least one of a small list of divisors (e.g. count numbers not coprime to a set of primes). Only practical for a small k (~20) — O(2^k) per query.

Time: O(2^k) per query. tested

```
ll countDivisibleByAny(ll x, vector<ll> &bad) {
    int k = (int)bad.size();
    ll res = 0;
    for (int m = 1; m < (1 << k); m++) {
        int amt = __builtin_popcount(m);
        ll L = 1;
        for (int i = 0; i < k; i++)
            if (m & (1 << i)) L = L / __gcd(L, bad[i]) *
bad[i];
        ll cnt = x / L;
        res += (amt % 2 == 1 ? cnt : -cnt);
    }
    return res;
}
```

Matrix Exponentiation

edbe9132

Matrix multiplication and exponentiation mod MOD, for linear recurrences / graph-walk-counting DP sped up via binary exponentiation.

Time: matmul O(n^3); matpow O(n^3 \log b). tested

```
typedef vector<vector<ll>> Matrix;

Matrix matmul(Matrix &a, Matrix &b, ll MOD) {
    int n = (int)a.size(), m = (int)b[0].size(), k =
(int)b.size();
    Matrix c(n, vector<ll>(m, 0));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            for (int l = 0; l < k; l++)
                c[i][j] = (c[i][j] + a[i][l] * b[l][j]) %
MOD;
    return c;
}

Matrix matpow(Matrix a, ll b, ll MOD) {
    int n = (int)a.size();
    Matrix res(n, vector<ll>(n, 0));
    for (int i = 0; i < n; i++) res[i][i] = 1;
    while (b) {
        if (b & 1) res = matmul(res, a, MOD);
        a = matmul(a, a, MOD);
        b >>= 1;
    }
    return res;
}
```

Mod Basics

d06d58b8

Modular exponentiation, single modular inverse (Fermat, MOD must be prime), and a linear-sieve inverse table for 1..MAXN-1 (MOD must be prime, MAXN < MOD).

Time: binpow/invMod O(\log b); invTable O(MAXN) total, O(1) per query after. tested

```
ll binpow(ll a, ll b, ll MOD) {
    a %= MOD;
    if (a < 0) a += MOD;
    ll res = 1;
    while (b) {
        if (b & 1) res = res * a % MOD;
        a = a * a % MOD;
        b >>= 1;
    }
    return res;
}

ll invMod(ll a, ll MOD) { return binpow(a, MOD - 2, MOD); }
```

```
vector<ll> invTable(int maxn, ll MOD) {
    vector<ll> inv(maxn);
    inv[1] = 1;
    for (int i = 2; i < maxn; i++) inv[i] = (MOD - MOD / i *
inv[MOD % i] % MOD) % MOD;
    return inv;
}
```

Ncr Factorials

5e3e87bd

nCr mod a prime via precomputed factorials + inverse factorials. Call build(maxn, MOD) once, then choose(n, k) is O(1). MOD must be prime and > maxn.

Time: O(maxn) build, O(1) per query. tested

```
struct Binomial {
    ll MOD;
    vector<ll> fact, inv;

    void build(int maxn, ll mod) {
        MOD = mod;
        fact.assign(maxn, 1);
        for (int i = 1; i < maxn; i++) fact[i] = fact[i - 1]
* i % MOD;
        inv.assign(maxn, 1);
        inv[maxn - 1] = invMod(fact[maxn - 1], MOD);
        for (int i = maxn - 2; i >= 0; i--) inv[i] = inv[i +
1] * (i + 1) % MOD;
    }

    ll choose(ll n, ll k) {
        if (k < 0 || k > n) return 0;
        return fact[n] * inv[k] % MOD * inv[n - k] % MOD;
    }
};
```

Ntt

a6e15b2d

Number Theoretic Transform (in-place, iterative) and exact polynomial multiplication mod `mod` (must be a prime of the form c*2^k+1 with a known primitive root). Default mod = (483<<21)+1, root = 62; other options: (119<<21)+1 (=998244353, root 3, most common), 5<<25+1, 7<<26+1. Source: kactl (CC0), see NOTICE.md.

Time: O(n \log n). kactl-derived, tested

```
const ll ntt_mod = (483LL << 21) + 1, ntt_root = 62;

ll ntt_modpow(ll a, ll b, ll mod) {
    ll r = 1;
    a %= mod;
    while (b) {
        if (b & 1) r = r * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return r;
}

void ntt(vector<ll> &a) {
    int n = (int)a.size(), L = 31 - __builtin_clz(n);
    static vector<ll> rt(2, 1);
    for (static int k = 2, s = 2; k < n; k *= 2, s++) {
        rt.resize(n);
        ll z[] = {1, ntt_modpow(ntt_root, ntt_mod >> s,
ntt_mod)};
        for (int i = k; i < 2 * k; i++) rt[i] = rt[i / 2] *
z[i & 1] % ntt_mod;
    }
    vector<int> rev(n);
    for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] | (i &
1) << L) / 2;
    for (int i = 0; i < n; i++) if (i < rev[i]) swap(a[i],
a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k)
            for (int j = 0; j < k; j++) {
                ll z = rt[j + k] * a[i + j + k] % ntt_mod,
&ai = a[i + j];
                a[i + j + k] = ai - z + (z > ai ? ntt_mod :
0);
                ai += (ai + z >= ntt_mod ? z - ntt_mod : z);
            }
}

vector<ll> conv(const vector<ll> &a, const vector<ll> &b) {
    if (a.empty() || b.empty()) return {};
    int s = (int)a.size() + (int)b.size() - 1, B = 32 -
__builtin_clz(s), n = 1 << B;
    ll inv = ntt_modpow(n, ntt_mod - 2, ntt_mod);
    vector<ll> L(a), R(b), out(n);
    L.resize(n); R.resize(n);
    ntt(L); ntt(R);
    for (int i = 0; i < n; i++) out[-i & (n - 1)] = L[i] *
R[i] % ntt_mod * inv % ntt_mod;
    ntt(out);
    return {out.begin(), out.begin() + s};
}
```

Prime Factorization

eddd322c

Prime factorization via trial division. Returns factors with multiplicity, ascending. For gcd/lcm, just use std::gcd/std::lcm (<numeric>).

Time: O(\sqrt n). tested

```
vector<ll> primeFactors(ll n) {
    vector<ll> factors;
    for (ll i = 2; i * i <= n; i++) {
        while (n % i == 0) {
            factors.push_back(i);
            n /= i;
        }
    }
    if (n > 1) factors.push_back(n);
    return factors;
}
```

Segmented Sieve

76122da7

Primes in [l, r] for huge r (beyond what a direct sieve up to r would fit in memory), using base primes up to sqrt(r).

Time: O((r - l + 1) \log\log r + \sqrt r \log\log \sqrt r). tested

```
vector<ll> segmentedSieve(ll l, ll r) {
    l = max(l, 2LL);
    if (l > r) return {};
    auto [isPrimeSmall, basePrimes] = sieve((int)sqrtl((long
double)r) + 1);

    vector<bool> isComposite(r - l + 1, false);
    for (ll p : basePrimes) {
        ll start = max(p * p, (l + p - 1) / p * p);
        for (ll j = start; j <= r; j += p) isComposite[j -
l] = true;
    }

    vector<ll> primes;
    for (ll i = l; i <= r; i++) if (!isComposite[i - l])
primes.push_back(i);
    return primes;
}
```

Sieve

a924f4ae

Sieve of Eratosthenes. Returns isPrime[0..n] and the ascending list of primes <= n.

Time: O(n \log\log n). tested

```
pair<vector<bool>, vector<int>> sieve(int n) {
    vector<bool> isPrime(n + 1, true);
    if (n >= 0) isPrime[0] = false;
    if (n >= 1) isPrime[1] = false;
    for (int i = 2; (ll)i * i <= n; i++)
        if (isPrime[i])
            for (int j = i * i; j <= n; j += i) isPrime[j] =
false;
```

```
    vector<int> primes;
    for (int i = 2; i <= n; i++) if (isPrime[i])
primes.push_back(i);
    return {isPrime, primes};
}
```

Xor Basis 9555684b

XOR linear basis over GF(2), BIT bits wide. add() inserts a vector into the basis (returns false if it was already in the span); check() tests membership without modifying the basis. Distinct from xor-trie.h (which enumerates max-XOR against an explicit multiset, not a basis).

Time: O(BIT) per operation. tested

```
template <int BIT>
struct XorBasis {
    array<bitset<BIT>, BIT> b{};

    bool add(bitset<BIT> x) {
        for (int i = BIT - 1; i >= 0; i--) {
            if (!x.test(i)) continue;
            if (b[i].any()) x ^= b[i];
            else { b[i] = x; return true; }
        }
        return false;
    }

    bool check(bitset<BIT> x) {
        for (int i = BIT - 1; i >= 0; i--) {
            if (!x.test(i)) continue;
            if (!b[i].any()) return false;
            x ^= b[i];
        }
        return true;
    }
};
```

Xor Basis Time 0c934f7f

XOR linear basis over GF(2), BIT bits wide, tagged with insertion time. On insert, the basis always keeps the newest vector at each pivot bit (displacing an older one further down), so queryMax/checkFrom can restrict to a suffix of insertions (time >= minTime) — the standard trick for "max/any XOR achievable using only elements inserted after time T" without rebuilding the basis per query. Distinct from xor-basis.h (no time dimension, ~40% shorter — use that one if you don't need suffix-of-insertions queries).

Time: O(BIT) per operation. tested

```
template <int BIT>
struct XorBasisTime {
    struct Entry { bitset<BIT> vec; int time = -1; };
    array<Entry, BIT> b{};

    void add(bitset<BIT> x, int t) {
        for (int i = BIT - 1; i >= 0; i--) {
            if (!x.test(i)) continue;
            if (b[i].time == -1) { b[i] = {x, t}; return; }
            if (t > b[i].time) { swap(x, b[i].vec); swap(t,
b[i].time); }
            x ^= b[i].vec;
        }
    }

    // max XOR achievable using only vectors with time >=
minTime
    bitset<BIT> queryMax(int minTime) {
        bitset<BIT> res;
        for (int i = BIT - 1; i >= 0; i--)
            if (b[i].time >= minTime && !res.test(i)) res ^=
b[i].vec;
        return res;
    }

    // is x representable using only vectors with time >=
minTime
    bool checkFrom(bitset<BIT> x, int minTime) {
        for (int i = BIT - 1; i >= 0; i--) {
            if (!x.test(i)) continue;
            if (b[i].time < minTime) return false;
            x ^= b[i].vec;
        }
        return x.none();
    }
};
```

Geometry

Closest Pair e891ddb2

Closest pair of points via divide and conquer. Mutates a copy (sorts it).

Time: O(n \log n). tested

```
ld closestPairRec(vector<Point> &pts, int lo, int hi) {  //
pts[lo..hi) sorted by x
    if (hi - lo <= 3) {
        ld best = numeric_limits<ld>::max();
        for (int i = lo; i < hi; i++)
            for (int j = i + 1; j < hi; j++) best =
min(best, (pts[i] - pts[j]).dist());
        return best;
    }

    int mid = (lo + hi) / 2;
    ld midX = pts[mid].x;
    ld best = min(closestPairRec(pts, lo, mid),
closestPairRec(pts, mid, hi));

    vector<Point> strip;
    for (int i = lo; i < hi; i++) if (abs(pts[i].x - midX) <
best) strip.push_back(pts[i]);
    sort(strip.begin(), strip.end(), [](Point a, Point b)
{ return a.y < b.y; });
    for (int i = 0; i < (int)strip.size(); i++)
        for (int j = i + 1; j < (int)strip.size() &&
strip[j].y - strip[i].y < best; j++)
            best = min(best, (strip[i] - strip[j]).dist());
    return best;
}

ld closestPair(vector<Point> pts) {
    sort(pts.begin(), pts.end(), [](Point a, Point b)
{ return a.x < b.x; });
    return closestPairRec(pts, 0, (int)pts.size());
}
```

Convex Hull d413dc5f

Convex hull via monotone chain. Returns hull vertices in counterclockwise order, no three consecutive collinear points (strictly convex turns only). Mutates a copy of the input (sorts it); pass by value or copy first if you need the original order.

Time: O(n \log n). tested

```
vector<Point> convexHull(vector<Point> pts) {
    int n = (int)pts.size();
    if (n <= 2) return pts;
    sort(pts.begin(), pts.end(), [](Point a, Point b)
{ return tie(a.x, a.y) < tie(b.x, b.y); });
    pts.erase(unique(pts.begin(), pts.end()), pts.end());
    n = (int)pts.size();
    if (n <= 2) return pts;

    vector<Point> hull(2 * n);
    int k = 0;
    for (int i = 0; i < n; i++) {
        while (k >= 2 && hull[k-2].cross(hull[k-1], pts[i])
<= 0) k--;
        hull[k++] = pts[i];
    }
    int lower = k + 1;
    for (int i = n - 2; i >= 0; i--) {
        while (k >= lower && hull[k-2].cross(hull[k-1],
pts[i]) <= 0) k--;
        hull[k++] = pts[i];
    }
    hull.resize(k - 1);
```

```
    return hull;
}
```

Point

683c0d0f

2D point/vector primitive. Foundation for all other geometry snippets — every geometry file in this notebook #includes this. cross(p) is the z-component of the 3D cross product (positive if p is counterclockwise from *this); cross(a,b) is the same but with *this as the pivot (positive if a->b turns left as seen from here).

Time: O(1) per operation. tested

```
struct Point {
    ld x, y;
    Point(ld x = 0, ld y = 0) : x(x), y(y) {}

    bool operator==(Point p) const { return tie(x, y) ==
tie(p.x, p.y); }
    Point operator+(Point p) const { return Point(x + p.x, y
+ p.y); }
    Point operator-(Point p) const { return Point(x - p.x, y
- p.y); }
    Point operator*(ld d) const { return Point(x * d, y *
d); }
    Point operator/(ld d) const { return Point(x / d, y /
d); }

    ld dist2() const { return x * x + y * y; }
    ld dist() const { return sqrtl(dist2()); }
    ld dot(Point p) const { return x * p.x + y * p.y; }
    ld angle() const { return atan2(y, x); }
    ld cross(Point p) const { return x * p.y - y * p.x; }
    ld cross(Point a, Point b) const { return (a -
*this).cross(b - *this); }

    Point unit() const { return *this / dist(); }
    Point rotate(ld a) const { return Point(x * cos(a) - y *
sin(a), x * sin(a) + y * cos(a)); }

    friend ostream &operator<<(ostream &os, Point p)
{ return os << p.x << " " << p.y; }
};
```

Point In Polygon

aa16bff9

Point-in-polygon test for a simple polygon (convex or not). Returns 1 if strictly inside, 0 if strictly outside, -1 if on the boundary.

Time: O(n). tested

```
int pointInPolygon(vector<Point> &poly, Point p) {
    int n = (int)poly.size();
    bool inside = false;
    for (int i = 0, j = n - 1; i < n; j = i++) {
        Point a = poly[i], b = poly[j];
        // on-segment check: collinear and within the
bounding box
        if (abs(p.cross(a, b)) < 1e-9 && min(a.x,b.x) - 1e-9
<= p.x && p.x <= max(a.x,b.x) + 1e-9
                                    && min(a.y,b.y) -
1e-9 <= p.y && p.y <= max(a.y,b.y) + 1e-9) return -1;
        if ((a.y > p.y) != (b.y > p.y)) {
            ld xCross = a.x + (p.y - a.y) * (b.x - a.x) /
(b.y - a.y);
            if (p.x < xCross) inside = !inside;
        }
    }
    return inside ? 1 : 0;
}
```

Polygon Area

384fe2b4

Polygon area via the shoelace formula. Works for any simple polygon (convex or not), vertices in either winding order.

Time: O(n). tested

```
ld polygonArea(vector<Point> &poly) {
    ld res = 0;
    int n = (int)poly.size();
    for (int i = 0; i < n; i++) res += poly[i].cross(poly[(i
+ 1) % n]);
    return abs(res) / 2.0;
}
```

Segment Intersection

980f3679

Test whether segments (a,b) and (c,d) intersect (including touching at an endpoint or overlapping collinearly).

Time: O(1). tested

```
int sgn(ld x) { return (x > 1e-9) - (x < -1e-9); }

bool onSegment(Point a, Point b, Point p) {  // p collinear
with a,b already assumed by caller context
    return min(a.x,b.x) - 1e-9 <= p.x && p.x <= max(a.x,b.x)
+ 1e-9
        && min(a.y,b.y) - 1e-9 <= p.y && p.y <= max(a.y,b.y)
+ 1e-9;
}

bool segmentsIntersect(Point a, Point b, Point c, Point d) {
    int d1 = sgn(a.cross(b, c)), d2 = sgn(a.cross(b, d));
    int d3 = sgn(c.cross(d, a)), d4 = sgn(c.cross(d, b));
    if (d1 != d2 && d3 != d4) return true;
    if (d1 == 0 && onSegment(a, b, c)) return true;
    if (d2 == 0 && onSegment(a, b, d)) return true;
    if (d3 == 0 && onSegment(c, d, a)) return true;
    if (d4 == 0 && onSegment(c, d, b)) return true;
    return false;
}
```

Flows

Bipartite Matching

758317fb

Maximum bipartite matching via augmenting paths (Kuhn's). Left side has nLeft nodes (0-indexed), adj[u] lists right-side neighbors; matchR[v] = matched left node or -1. Slower than Hopcroft-Karp in the worst case but much shorter to type.

Time: O(V \cdot E). tested

```
struct BipartiteMatching {
    int nLeft, nRight;
    vector<vector<int>> adj;
    vector<int> matchL, matchR;
    vector<bool> vis;

    BipartiteMatching(int nLeft, int nRight) : nLeft(nLeft),
nRight(nRight), adj(nLeft), matchL(nLeft, -1),
matchR(nRight, -1) {}

    void addEdge(int u, int v) { adj[u].push_back(v); }

    bool tryKuhn(int u) {
        for (int v : adj[u]) {
            if (vis[v]) continue;
            vis[v] = true;
            if (matchR[v] == -1 || tryKuhn(matchR[v])) {
                matchL[u] = v; matchR[v] = u;
                return true;
            }
        }
        return false;
    }

    int maxMatching() {
        int res = 0;
        for (int u = 0; u < nLeft; u++) {
            vis.assign(nRight, false);
            if (tryKuhn(u)) res++;
        }
        return res;
    }
};
```

Dinic

d5c183c9

Dinic's max flow with capacity scaling: process augmenting paths in decreasing order of a capacity threshold (31 phases, thresholds 2^30 down to 2^0) so large-capacity paths get pushed first. The last phase (threshold 2^0) is always an unrestricted normal Dinic's pass, so correctness holds regardless of how large capacities are — scaling only affects how fast you get there. Prefer this over edmonds-karp.h. Source: kactl (CC0), see NOTICE.md.

Time: O(VE \log U), U = \max cap; O(E \sqrt V) for unit-capacity/ bipartite-matching graphs. kactl-derived, tested

```
template <typename T>
struct Dinic {
    struct Edge { int to, rev; T cap, cap0; T flow()
{ return max(cap0 - cap, T(0)); } };
    vector<vector<Edge>> adj;
    vector<int> lvl, ptr;
    int n;

    Dinic(int n) : adj(n), lvl(n), ptr(n), n(n) {}

    void addEdge(int a, int b, T cap, T rcap = 0) {
        adj[a].push_back({b, (int)adj[b].size(), cap, cap});
        adj[b].push_back({a, (int)adj[a].size() - 1, rcap,
rcap});
    }

    T dfs(int v, int t, T f) {
        if (v == t || f == 0) 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 (T p = dfs(e.to, t, min(f, e.cap))) {
                    e.cap -= p;
                    adj[e.to][e.rev].cap += p;
                    return p;
                }
            }
        }
        return 0;
    }

    T maxflow(int s, int t) {
        T flow = 0;
        vector<int> q(n);
        q[0] = s;
        for (int shift = 30; shift >= 0; shift--) {
            do {
                fill(lvl.begin(), lvl.end(), 0);
                fill(ptr.begin(), ptr.end(), 0);
                int qi = 0, qe = 0;
                lvl[s] = 1;
                q[qe++] = s;
                while (qi < qe && !lvl[t]) {
                    int v = q[qi++];
                    for (auto &e : adj[v])
                        if (!lvl[e.to] && (e.cap >> shift))
{ lvl[e.to] = lvl[v] + 1; q[qe++] = e.to; }
                }
                while (T p = dfs(s, t,
numeric_limits<T>::max())) flow += p;
            } while (lvl[t]);
        }
        return flow;
    }
};
```

Edmonds Karp

69f9586e

Edmonds-Karp max flow (BFS augmenting paths), same edge-list interface as dinic.h. Simpler to type but strictly worse complexity — prefer dinic.h unless V and E are both small.

Time: O(VE^2). tested

```
template <typename T>
struct EdmondsKarp {
    struct Edge { int to; T cap; };
    vector<Edge> edges;
    vector<vector<int>> adj;
    int n;

    EdmondsKarp(int n) : adj(n), n(n) {}

    void addEdge(int u, int v, T cap, T rcap = 0) {
        adj[u].push_back((int)edges.size());
edges.push_back({v, cap});
        adj[v].push_back((int)edges.size());
edges.push_back({u, rcap});
    }

    T maxflow(int s, int t) {
        T flow = 0;
        while (true) {
            vector<int> parEdge(n, -1);
            vector<bool> vis(n, false);
            queue<int> q;
            vis[s] = true; q.push(s);
            while (!q.empty() && !vis[t]) {
                int u = q.front(); q.pop();
                for (int id : adj[u]) {
                    auto &e = edges[id];
                    if (e.cap > 0 && !vis[e.to]) { vis[e.to]
= true; parEdge[e.to] = id; q.push(e.to); }
                }
            }
            if (!vis[t]) break;

            T bottleneck = numeric_limits<T>::max();
            for (int v = t; v != s; v = edges[parEdge[v] ^
1].to) bottleneck = min(bottleneck, edges[parEdge[v]].cap);
            for (int v = t; v != s; v = edges[parEdge[v] ^
1].to) {
                edges[parEdge[v]].cap -= bottleneck;
                edges[parEdge[v] ^ 1].cap += bottleneck;
            }
            flow += bottleneck;
        }
        return flow;
    }
};
```

Mcmf

b9362c4f

Min-cost max-flow via Dijkstra with Johnson's potentials (a PBDS pairing heap gives O(1) decrease-key). If costs can be negative, call setpi(s) once before maxflow (negative cost cycles are not supported). Add every edge via addEdge() before the first maxflow() call — edges are stored by value per-node and augmenting paths hold raw pointers into them, which addEdge()'s push_back could invalidate mid-flow.

Time: O(FE \log V), F = total flow; O(VE) for setpi. kactl-derived, tested

```
#undef int  // pb_ds headers use 'int' internally; #define
int long long corrupts them
#include <ext/pb_ds/priority_queue.hpp>
#define int long long

template <typename T>
struct MCMF {
    struct Edge { int from, to, rev; T cap, cost, flow; };
    int n;
    vector<vector<Edge>> adj;
    vector<bool> seen;
    vector<T> dist, pi;
    vector<Edge *> par;
    T INF = numeric_limits<T>::max() / 4;

    MCMF(int n) : n(n), adj(n), seen(n), dist(n), pi(n,
T()), par(n) {}

    void addEdge(int from, int to, T cap, T cost) {
        if (from == to) return;
        adj[from].push_back({from, to, (int)adj[to].size(),
cap, cost, T()});
        adj[to].push_back({to, from, (int)adj[from].size() -
1, T(), -cost, T()});
    }
```

```
    void path(int s) {
        fill(seen.begin(), seen.end(), false);
        fill(dist.begin(), dist.end(), INF);
        dist[s] = T();
        __gnu_pbds::priority_queue<pair<T, int>> q;
        vector<typename decltype(q)::point_iterator> its(n,
q.end());
        q.push({T(), s});
        while (!q.empty()) {
            s = q.top().second; q.pop();
            seen[s] = true;
            T di = dist[s] + pi[s];
            for (Edge &e : adj[s]) {
                if (seen[e.to]) continue;
                T val = di - pi[e.to] + e.cost;
                if (e.cap - e.flow > 0 && val < dist[e.to])
{
                    dist[e.to] = val;
                    par[e.to] = &e;
                    if (its[e.to] == q.end()) its[e.to] =
q.push({-dist[e.to], e.to});
                    else q.modify(its[e.to], {-dist[e.to],
e.to});
                }
            }
        }
        for (int i = 0; i < n; i++) pi[i] = min(pi[i] +
dist[i], INF);
    }

    pair<T, T> maxflow(int s, int t) {
        T totalFlow = T(), totalCost = T();
        for (path(s); seen[t]; path(s)) {
            T f = INF;
            for (Edge *e = par[t]; e; e = par[e->from]) f =
min(f, e->cap - e->flow);
            totalFlow += f;
            for (Edge *e = par[t]; e; e = par[e->from]) {
                e->flow += f;
                adj[e->to][e->rev].flow -= f;
            }
        }
        for (int i = 0; i < n; i++) for (Edge &e : adj[i])
totalCost += e.cost * e.flow;
        return {totalFlow, totalCost / 2};
    }

    // call before maxflow() only if some costs are negative
    void setpi(int s) {
        fill(pi.begin(), pi.end(), INF);
        pi[s] = T();
        int rounds = n, changed = 1;
        while (changed-- && rounds--) {
            for (int i = 0; i < n; i++) {
                if (pi[i] == INF) continue;
                for (Edge &e : adj[i]) {
                    if (e.cap == 0) continue;
                    if (pi[i] + e.cost < pi[e.to])
{ pi[e.to] = pi[i] + e.cost; changed = 1; }
                }
            }
        }
        assert(rounds >= 0);  // negative cost cycle
    }
};
```