Fortcoders Code Library

askd, yangster67, Nea1

April 5th 2024

Contents

Intro

Main template

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

#define FOR(x,n) for(int x=0;x<n;x++)
#define forn(i, n) for (int i = 0; i < int(n); i++)
#define all(v) v.begin(),v.end()
using ll = long long;
using ld = long double;
using pii = pair<int, int>;
const char nl = '\n';

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cout << fixed << setprecision(20);
  // mt19937
  // rng(chrono::steady_clock::now().time_since_epoch().count());
}
```

Fast IO

```cpp
namespace io {
constexpr int SIZE = 1 << 16;
char buf[SIZE], *head, *tail;
char get_char() {
  if (head == tail) tail = (head = buf) + fread(buf, 1, SIZE,
  ↪ stdin);
  return *head++;
}
ll read() {
  ll x = 0, f = 1;
  char c = get_char();
  for (; !isdigit(c); c = get_char()) (c == '-') && (f = -1);
  for (; isdigit(c); c = get_char()) x = x * 10 + c - '0';
  return x * f;
}
string read_s() {
  string str;
  char c = get_char();
  while (c == ' ' || c == '\n' || c == '\r') c = get_char();
  while (c != ' ' && c != '\n' && c != '\r') str += c, c =
  ↪ get_char();
  return str;
}
void print(int x) {
  if (x > 9) print(x / 10);
  putchar(x % 10 | '0');
}
void println(int x) { print(x), putchar('\n'); }
struct Read {
  Read& operator>>(ll& x) { return x = read(), *this; }
  Read& operator>>(long double& x) { return x =
  ↪ stold(read_s()), *this; }
} in;
} // namespace io
```

Data Structures

Segment Tree

Recursive

- Implicit segment tree, range query + point update

```cpp
struct Node {
  int lc, rc, p;
};

struct SegTree {
  vector<Node> t = {{}};
  SegTree(int n) { t.reserve(n * 40); }
  int modify(int p, int l, int r, int x, int v) {
```

```cpp
    int u = p;
    if (p == 0) {
      t.push_back(t[p]);
      u = (int)t.size() - 1;
    }
    if (r - l == 1) {
      t[u].p = t[p].p + v;
    } else {
      int m = (l + r) / 2;
      if (x < m) {
        t[u].lc = modify(t[p].lc, l, m, x, v);
      } else {
        t[u].rc = modify(t[p].rc, m, r, x, v);
      }
      t[u].p = t[t[u].lc].p + t[t[u].rc].p;
    }
    return u;
  }
  int query(int p, int l, int r, int x, int y) {
    if (x <= l && r <= y) return t[p].p;
    int m = (l + r) / 2, res = 0;
    if (x < m) res += query(t[p].lc, l, m, x, y);
    if (y > m) res += query(t[p].rc, m, r, x, y);
    return res;
  }
};
```

- Persistent implicit, range query + point update

```cpp
struct Node {
  int lc = 0, rc = 0, p = 0;
};

struct SegTree {
  vector<Node> t = {{}};  // init all
  SegTree() = default;
  SegTree(int n) { t.reserve(n * 20); }
  int modify(int p, int l, int r, int x, int v) {
    // p: original node, update a[x] -> v
    t.push_back(t[p]);
    int u = (int)t.size() - 1;
    if (r - l == 1) {
      t[u].p = v;
    } else {
      int m = (l + r) / 2;
      if (x < m) {
        t[u].lc = modify(t[p].lc, l, m, x, v);
        t[u].rc = t[p].rc;
      } else {
        t[u].lc = t[p].lc;
        t[u].rc = modify(t[p].rc, m, r, x, v);
      }
      t[u].p = t[t[u].lc].p + t[t[u].rc].p;
    }
    return u;
  }
  int query(int p, int l, int r, int x, int y) {
    // query sum a[x]...a[y-1] rooted at p
    // t[p] holds the info of [l, r)
    if (x <= l && r <= y) return t[p].p;
    int m = (l + r) / 2, res = 0;
    if (x < m) res += query(t[p].lc, l, m, x, y);
    if (y > m) res += query(t[p].rc, m, r, x, y);
    return res;
  }
};
```

Iterating

- Iterating, range query + point update

```cpp
struct Node {
  ll v = 0, init = 0;
};

Node pull(const Node &a, const Node &b) {
  if (!a.init) return b;
```

```
7      if (!b.init) return a;
8      Node c;
9      return c;
10   }
11
12   struct SegTree {
13     ll n;
14     vector<Node> t;
15     SegTree(ll _n) : n(_n), t(2 * n){};
16     void modify(ll p, const Node &v) {
17       t[p += n] = v;
18       for (p /= 2; p; p /= 2) t[p] = pull(t[p * 2], t[p * 2 +
    ↪ 1]);
19     }
20     Node query(ll l, ll r) {
21       Node left, right;
22       for (l += n, r += n; l < r; l /= 2, r /= 2) {
23         if (l & 1) left = pull(left, t[l++]);
24         if (r & 1) right = pull(t[--r], right);
25       }
26       return pull(left, right);
27     }
28   };
```

- Iterating, range query + range update

```
1    struct Node {
2      ll v = 0;
3    };
4    struct Tag {
5      ll v = 0;
6    };
7    Node pull(const Node& a, const Node& b) { return {max(a.v,
    ↪ b.v)}; }
8    Tag pull(const Tag& a, const Tag& b) { return {a.v + b.v}; }
9    Node apply_tag(const Node& a, const Tag& b) { return {a.v +
    ↪ b.v}; }
10
11   struct SegTree {
12     ll n, h;
13     vector<Node> t;
14     vector<Tag> lazy;
15     SegTree(ll _n) : n(_n), h((ll)log2(n)), t(2 * _n), lazy(2 *
    ↪ _n) {}
16     void apply(ll x, const Tag& tag) {
17       t[x] = apply_tag(t[x], tag);
18       lazy[x] = pull(lazy[x], tag);
19     }
20     void build(ll l) {
21       for (l = (l + n) / 2; l > 0; l /= 2) {
22         if (!lazy[l].v) t[l] = pull(t[l * 2], t[2 * l + 1]);
23       }
24     }
25     void push(ll l) {
26       l += n;
27       for (ll s = h; s > 0; s--) {
28         ll i = l >> s;
29         if (lazy[i].v) {
30           apply(2 * i, lazy[i]);
31           apply(2 * i + 1, lazy[i]);
32         }
33         lazy[i] = Tag();
34       }
35     }
36     void modify(ll l, ll r, const Tag& v) {
37       push(l), push(r - 1);
38       ll l0 = l, r0 = r;
39       for (l += n, r += n; l < r; l /= 2, r /= 2) {
40         if (l & 1) apply(l++, v);
41         if (r & 1) apply(--r, v);
42       }
43       build(l0), build(r0 - 1);
44     }
45     Node query(ll l, ll r) {
46       push(l), push(r - 1);
47       Node left, right;
48       for (l += n, r += n; l < r; l /= 2, r /= 2) {
49         if (l & 1) left = pull(left, t[l++]);
```

```
50         if (r & 1) right = pull(t[--r], right);
51       }
52       return pull(left, right);
53     }
54   };
```

- AtCoder Segment Tree (recursive structure but iterative)

```
1    template <class T> struct PointSegmentTree {
2      int size = 1;
3      vector<T> tree;
4      PointSegmentTree(int n) : PointSegmentTree(vector<T>(n)) {}
5      PointSegmentTree(vector<T>& arr) {
6        while(size < (int)arr.size())
7          size <<= 1;
8        tree = vector<T>(size << 1);
9        for(int i = size + arr.size() - 1; i >= 1; i--)
10         if(i >= size) tree[i] = arr[i - size];
11         else consume(i);
12     }
13     void set(int i, T val) {
14       tree[i += size] = val;
15       for(i >>= 1; i >= 1; i >>= 1)
16         consume(i);
17     }
18     T get(int i) { return tree[i + size]; }
19     T query(int l, int r) {
20       T resl, resr;
21       for(l += size, r += size + 1; l < r; l >>= 1, r >>= 1) {
22         if(l & 1) resl = resl * tree[l++];
23         if(r & 1) resr = tree[--r] * resr;
24       }
25       return resl * resr;
26     }
27     T query_all() { return tree[1]; }
28     void consume(int i) { tree[i] = tree[i << 1] * tree[i << 1 |
    ↪ 1]; }
29   };
30
31
32   struct SegInfo {
33     ll v;
34     SegInfo() : SegInfo(0) {}
35     SegInfo(ll val) : v(val) {}
36     SegInfo operator*(SegInfo b) {
37       return SegInfo(v + b.v);
38     }
39   };
```

cdq

```
1    function<void(int, int)> solve = [&](int l, int r) {
2      if (r == l + 1) return;
3      int mid = (l + r) / 2;
4      auto middle = b[mid];
5      solve(l, mid), solve(mid, r);
6      sort(b.begin() + l, b.begin() + r, [&](auto& x, auto& y) {
7        return array{x[1], x[2], x[0]} < array{y[1], y[2], y[0]};
8      });
9      for (int i = l; i < r; i++) {
10       if (b[i] < middle) {
11         seg.modify(b[i][2], b[i][3]);
12       } else {
13         b[i][4] += seg.query(0, b[i][2] + 1);
14       }
15     }
16     for (int i = l; i < r; i++) {
17       if (b[i] < middle) seg.modify(b[i][2], -b[i][3]);
18     }
19   };
20   solve(0, n);
```

Cartesian Tree

```
1    struct CartesianTree {
2      int n; vector<int> lson, rson;
```

```cpp
3     CartesianTree(vector<int>& a) : n(int(a.size())), lson(n,
  ↪   -1), rson(n, -1) {
4       vector<int> stk;
5       for (int i = 0; i < n; i++) {
6         while (stk.size() && a[stk.back()] > a[i]) {
7           lson[i] = stk.back(), stk.pop_back();
8         }
9         if (stk.size()) rson[stk.back()] = i;
10        stk.push_back(i);
11      }
12    }
13  };
```

Union Find

```cpp
1   struct DSU {
2       vector<int> e;
3
4       DSU(int N) {
5           e = vector<int>(N, -1);
6       }
7
8       // get representative component (uses path compression)
9       int get(int x) { return e[x] < 0 ? x : e[x] = get(e[x]); }
10
11      bool same_set(int a, int b) { return get(a) == get(b); }
12
13      int size(int x) { return -e[get(x)]; }
14
15      bool unite(int x, int y) {  // union by size, merge y into
  ↪   x
16          x = get(x), y = get(y);
17          if (x == y) return false;
18          if (e[x] > e[y]) swap(x, y);
19          e[x] += e[y]; e[y] = x;
20          return true;
21      }
22  };
```

- Persistent version

```cpp
1   struct Node {
2     int lc, rc, p;
3   };
4
5   struct SegTree {
6     vector<Node> t = {{0, 0, -1}};  // init all
7     SegTree() = default;
8     SegTree(int n) { t.reserve(n * 20); }
9     int modify(int p, int l, int r, int x, int v) {
10      // p: original node, update a[x] -> v
11      t.push_back(t[p]);
12      int u = (int)t.size() - 1;
13      if (r - l == 1) {
14        t[u].p = v;
15      } else {
16        int m = (l + r) / 2;
17        if (x < m) {
18          t[u].lc = modify(t[p].lc, l, m, x, v);
19          t[u].rc = t[p].rc;
20        } else {
21          t[u].lc = t[p].lc;
22          t[u].rc = modify(t[p].rc, m, r, x, v);
23        }
24        t[u].p = t[t[u].lc].p + t[t[u].rc].p;
25      }
26      return u;
27    }
28    int query(int p, int l, int r, int x, int y) {
29      // query sum a[x]...a[y-1] rooted at p
30      // t[p] holds the info of [l, r)
31      if (x <= l && r <= y) return t[p].p;
32      int m = (l + r) / 2, res = 0;
33      if (x < m) res += query(t[p].lc, l, m, x, y);
34      if (y > m) res += query(t[p].rc, m, r, x, y);
35      return res;
36    }
```

```cpp
37  };
38
39  struct DSU {
40    int n;
41    SegTree seg;
42    DSU(int _n) : n(_n), seg(n) {}
43    int get(int p, int x) { return seg.query(p, 0, n, x, x + 1);
  ↪   }
44    int set(int p, int x, int v) { return seg.modify(p, 0, n, x,
  ↪   v); }
45    int find(int p, int x) {
46      int parent = get(p, x);
47      if (parent < 0) return x;
48      return find(p, parent);
49    }
50    int is_same(int p, int x, int y) { return find(p, x) ==
  ↪   find(p, y); }
51    int merge(int p, int x, int y) {
52      int rx = find(p, x), ry = find(p, y);
53      if (rx == ry) return -1;
54      int rank_x = -get(p, rx), rank_y = -get(p, ry);
55      if (rank_x < rank_y) {
56        p = set(p, rx, ry);
57      } else if (rank_x > rank_y) {
58        p = set(p, ry, rx);
59      } else {
60        p = set(p, ry, rx);
61        p = set(p, rx, -rx - 1);
62      }
63      return p;
64    }
65  };
```

Fenwick Tree

```cpp
1   template <typename T> struct FenwickTree {
2     int size = 1, high_bit = 1;
3     vector<T> tree;
4     FenwickTree(int _size) : size(_size) {
5       tree.resize(size + 1);
6       while((high_bit << 1) <= size) high_bit <<= 1;
7     }
8     FenwickTree(vector<T>& arr) : FenwickTree(arr.size()) {
9       for(int i = 0; i < size; i++) update(i, arr[i]);
10    }
11    int lower_bound(T x) {
12      int res = 0; T cur = 0;
13      for(int bit = high_bit; bit > 0; bit >>= 1) {
14        if((res|bit) <= size && cur + tree[res|bit] < x) {
15          res |= bit; cur += tree[res];
16        }
17      }
18      return res;
19    }
20    T prefix_sum(int i) {
21      T ret = 0;
22      for(i++; i > 0; i -= (i & -i)) ret += tree[i];
23      return ret;
24    }
25    T range_sum(int l, int r) { return (l > r) ? 0 :
  ↪   prefix_sum(r) - prefix_sum(l - 1); }
26    void update(int i, T delta) { for(i++; i <= size; i += (i &
  ↪   -i)) tree[i] += delta; }
27  };
```

Fenwick2D Tree

```cpp
1   struct Fenwick2D {
2     ll n, m;
3     vector<vector<ll>> a;
4     Fenwick2D(ll _n, ll _m) : n(_n), m(_m), a(n, vector<ll>(m))
  ↪   {}
5     void add(ll x, ll y, ll v) {
6       for (int i = x + 1; i <= n; i += i & -i) {
7         for (int j = y + 1; j <= m; j += j & -j) {
8           (a[i - 1][j - 1] += v) %= MOD;
```

```
 9          }
10        }
11      }
12      void add(ll x1, ll x2, ll y1, ll y2, ll v) {
13        // [(x1, y1), (x2, y2))
14        add(x1, y1, v);
15        add(x1, y2, MOD - v), add(x2, y1, MOD - v);
16        add(x2, y2, v);
17      }
18      ll sum(ll x, ll y) {  // [(0, 0), (x, y))
19        ll ans = 0;
20        for (int i = x; i > 0; i -= i & -i) {
21          for (int j = y; j > 0; j -= j & -j) {
22            (ans += a[i - 1][j - 1]) %= MOD;
23          }
24        }
25        return ans;
26      }
27    };
```

PBDS

```
 1    #include <bits/stdc++.h>
 2    #include <ext/pb_ds/assoc_container.hpp>
 3    using namespace std;
 4    using namespace __gnu_pbds;
 5    template<typename T>
 6    using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
     ↪  tree_order_statistics_node_update>;
 7    template<typename T, typename X>
 8    using ordered_map = tree<T, X, less<T>, rb_tree_tag,
     ↪  tree_order_statistics_node_update>;
 9    template<typename T, typename X>
10    using fast_map = cc_hash_table<T, X>;
11    template<typename T, typename X>
12    using ht = gp_hash_table<T, X>;
13    mt19937_64
     ↪  rng(chrono::steady_clock::now().time_since_epoch().count());
14
15    struct splitmix64 {
16        size_t operator()(size_t x) const {
17            static const size_t fixed =
     ↪  chrono::steady_clock::now().time_since_epoch().count();
18            x += 0x9e3779b97f4a7c15 + fixed;
19            x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
20            x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
21            return x ^ (x >> 31);
22        }
23    };
```

Treap

- (No rotation version)

```
 1    struct Node {
 2      Node *l, *r;
 3      int s, sz;
 4      // int t = 0, a = 0, g = 0; // for lazy propagation
 5      ll w;
 6
 7      Node(int _s) : l(nullptr), r(nullptr), s(_s), sz(1),
     ↪  w(rng()) {}
 8      void apply(int vt, int vg) {
 9        // for lazy propagation
10        // s -= vt;
11        // t += vt, a += vg, g += vg;
12      }
13      void push() {
14        // for lazy propagation
15        // if (l != nullptr) l->apply(t, g);
16        // if (r != nullptr) r->apply(t, g);
17        // t = g = 0;
18      }
19      void pull() { sz = 1 + (l ? l->sz : 0) + (r ? r->sz : 0); }
20    };
21
```

```
22    std::pair<Node *, Node *> split(Node *t, int v) {
23      if (t == nullptr) return {nullptr, nullptr};
24      t->push();
25      if (t->s < v) {
26        auto [x, y] = split(t->r, v);
27        t->r = x;
28        t->pull();
29        return {t, y};
30      } else {
31        auto [x, y] = split(t->l, v);
32        t->l = y;
33        t->pull();
34        return {x, t};
35      }
36    }
37
38    Node *merge(Node *p, Node *q) {
39      if (p == nullptr) return q;
40      if (q == nullptr) return p;
41      if (p->w < q->w) swap(p, q);
42      auto [x, y] = split(q, p->s + rng() % 2);
43      p->push();
44      p->l = merge(p->l, x);
45      p->r = merge(p->r, y);
46      p->pull();
47      return p;
48    }
49
50    Node *insert(Node *t, int v) {
51      auto [x, y] = split(t, v);
52      return merge(merge(x, new Node(v)), y);
53    }
54
55    Node *erase(Node *t, int v) {
56      auto [x, y] = split(t, v);
57      auto [p, q] = split(y, v + 1);
58      return merge(merge(x, merge(p->l, p->r)), q);
59    }
60
61    int get_rank(Node *&t, int v) {
62      auto [x, y] = split(t, v);
63      int res = (x ? x->sz : 0) + 1;
64      t = merge(x, y);
65      return res;
66    }
67
68    Node *kth(Node *t, int k) {
69      k--;
70      while (true) {
71        int left_sz = t->l ? t->l->sz : 0;
72        if (k < left_sz) {
73          t = t->l;
74        } else if (k == left_sz) {
75          return t;
76        } else {
77          k -= left_sz + 1, t = t->r;
78        }
79      }
80    }
81
82    Node *get_prev(Node *&t, int v) {
83      auto [x, y] = split(t, v);
84      Node *res = kth(x, x->sz);
85      t = merge(x, y);
86      return res;
87    }
88
89    Node *get_next(Node *&t, int v) {
90      auto [x, y] = split(t, v + 1);
91      Node *res = kth(y, 1);
92      t = merge(x, y);
93      return res;
94    }
```

- USAGE

```
 1    int main() {
 2      cin.tie(nullptr)->sync_with_stdio(false);
```

```
3    int n;
4    cin >> n;
5    Node *t = nullptr;
6    for (int op, x; n--;) {
7      cin >> op >> x;
8      if (op == 1) {
9        t = insert(t, x);
10     } else if (op == 2) {
11       t = erase(t, x);
12     } else if (op == 3) {
13       cout << get_rank(t, x) << "\n";
14     } else if (op == 4) {
15       cout << kth(t, x)->s << "\n";
16     } else if (op == 5) {
17       cout << get_prev(t, x)->s << "\n";
18     } else {
19       cout << get_next(t, x)->s << "\n";
20     }
21   }
22 }
```

Implicit treap

- Split by size

```
1  struct Node {
2    Node *l, *r;
3    int s, sz;
4    // int lazy = 0;
5    ll w;
6
7    Node(int _s) : l(nullptr), r(nullptr), s(_s), sz(1),
   ↪  w(rnd()) {}
8    void apply() {
9      // for lazy propagation
10     // lazy ^= 1;
11   }
12   void push() {
13     // for lazy propagation
14     // if (lazy) {
15     //   swap(l, r);
16     //   if (l != nullptr) l->apply();
17     //   if (r != nullptr) r->apply();
18     //   lazy = 0;
19     // }
20   }
21   void pull() { sz = 1 + (l ? l->sz : 0) + (r ? r->sz : 0); }
22 };
23
24 std::pair<Node *, Node *> split(Node *t, int v) {
25   // first->sz == v
26   if (t == nullptr) return {nullptr, nullptr};
27   t->push();
28   int left_sz = t->l ? t->l->sz : 0;
29   if (left_sz < v) {
30     auto [x, y] = split(t->r, v - left_sz - 1);
31     t->r = x;
32     t->pull();
33     return {t, y};
34   } else {
35     auto [x, y] = split(t->l, v);
36     t->l = y;
37     t->pull();
38     return {x, t};
39   }
40 }
41
42 Node *merge(Node *p, Node *q) {
43   if (p == nullptr) return q;
44   if (q == nullptr) return p;
45   if (p->w < q->w) {
46     p->push();
47     p->r = merge(p->r, q);
48     p->pull();
49     return p;
50   } else {
51     q->push();
```

```
52     q->l = merge(p, q->l);
53     q->pull();
54     return q;
55   }
56 }
```

Persistent implicit treap

```
1  pair<Node *, Node *> split(Node *t, int v) {
2    // first->sz == v
3    if (t == nullptr) return {nullptr, nullptr};
4    t->push();
5    int left_sz = t->l ? t->l->sz : 0;
6    t = new Node(*t);
7    if (left_sz < v) {
8      auto [x, y] = split(t->r, v - left_sz - 1);
9      t->r = x;
10     t->pull();
11     return {t, y};
12   } else {
13     auto [x, y] = split(t->l, v);
14     t->l = y;
15     t->pull();
16     return {x, t};
17   }
18 }
19
20 Node *merge(Node *p, Node *q) {
21   if (p == nullptr) return new Node(*q);
22   if (q == nullptr) return new Node(*p);
23   if (p->w < q->w) {
24     p = new Node(*p);
25     p->push();
26     p->r = merge(p->r, q);
27     p->pull();
28     return p;
29   } else {
30     q = new Node(*q);
31     q->push();
32     q->l = merge(p, q->l);
33     q->pull();
34     return q;
35   }
36 }
```

2D Sparse Table

- Sorry that this sucks - askd

```
1  template <class T, class Compare = less<T>>
2  struct SparseTable2d {
3    int n = 0, m = 0;
4    T**** table;
5    int* log;
6    inline T choose(T x, T y) {
7      return Compare()(x, y) ? x : y;
8    }
9    SparseTable2d(vector<vector<T>>& grid) {
10     if(grid.empty() || grid[0].empty()) return;
11     n = grid.size(); m = grid[0].size();
12     log = new int[max(n, m) + 1];
13     log[1] = 0;
14     for(int i = 2; i <= max(n, m); i++)
15       log[i] = log[i - 1] + ((i ^ (i - 1)) > i);
16     table = new T***[n];
17     for(int i = n - 1; i >= 0; i--) {
18       table[i] = new T**[m];
19       for(int j = m - 1; j >= 0; j--) {
20         table[i][j] = new T*[log[n - i] + 1];
21         for(int k = 0; k <= log[n - i]; k++) {
22           table[i][j][k] = new T[log[m - j] + 1];
23           if(!k) table[i][j][k][0] = grid[i][j];
24           else table[i][j][k][0] = choose(table[i][j][k-1][0],
   ↪  table[i+(1<<(k-1))][j][k-1][0]);
25           for(int l = 1; l <= log[m - j]; l++)
```

```
26              table[i][j][k][l] = choose(table[i][j][k][l-1],
    ↪ table[i][j+(1<<(l-1))][k][l-1]);
27          }
28        }
29      }
30    }
31    T query(int r1, int r2, int c1, int c2) {
32      assert(r1 >= 0 && r2 < n && r1 <= r2);
33      assert(c1 >= 0 && c2 < m && c1 <= c2);
34      int rl = log[r2 - r1 + 1], cl = log[c2 - c1 + 1];
35      T ca1 = choose(table[r1][c1][rl][cl],
    ↪ table[r2-(1<<rl)+1][c1][rl][cl]);
36      T ca2 = choose(table[r1][c2-(1<<cl)+1][rl][cl],
    ↪ table[r2-(1<<rl)+1][c2-(1<<cl)+1][rl][cl]);
37      return choose(ca1, ca2);
38    }
39  };
```

- USAGE

```
1  vector<vector<int>> test =  {
2    {1, 2, 3, 4}, {2, 3, 4, 5}, {9, 9, 9, 9}, {-1, -1, -1, -1}
3  };
4
5  SparseTable2d<int> st(test);              // Range min query
6  SparseTable2d<int,greater<int>> st2(test);  // Range max query
```

K-D Tree

```
1  struct Point {
2    int x, y;
3  };
4  struct Rectangle {
5    int lx, rx, ly, ry;
6  };
7
8  bool is_in(const Point &p, const Rectangle &rg) {
9    return (p.x >= rg.lx) && (p.x <= rg.rx) && (p.y >= rg.ly) &&
    ↪ (p.y <= rg.ry);
10  }
11
12  struct KDTree {
13    vector<Point> points;
14    struct Node {
15      int lc, rc;
16      Point point;
17      Rectangle range;
18      int num;
19    };
20    vector<Node> nodes;
21    int root = -1;
22    KDTree(const vector<Point> &points_) {
23      points = points_;
24      Rectangle range = {-1e9, 1e9, -1e9, 1e9};
25      root = tree_construct(0, (int)points.size(), range, 0);
26    }
27    int tree_construct(int l, int r, Rectangle range, int depth)
    ↪ {
28      if (l == r) return -1;
29      if (l > r) throw;
30      int mid = (l + r) / 2;
31      auto comp = (depth % 2) ? [](Point &a, Point &b) { return
    ↪ a.x < b.x; }
32                             : [](Point &a, Point &b) { return
    ↪ a.y < b.y; };
33      nth_element(points.begin() + l, points.begin() + mid,
    ↪ points.begin() + r, comp);
34      Rectangle l_range(range), r_range(range);
35      if (depth % 2) {
36        l_range.rx = points[mid].x;
37        r_range.lx = points[mid].x;
38      } else {
39        l_range.ry = points[mid].y;
40        r_range.ly = points[mid].y;
41      }
42      Node node = {tree_construct(l, mid, l_range, depth + 1),
```

```
43              tree_construct(mid + 1, r, r_range, depth +
    ↪ 1), points[mid], range, r - l};
44      nodes.push_back(node);
45      return (int)nodes.size() - 1;
46    }
47
48    int inner_query(int id, const Rectangle &rec, int depth) {
49      if (id == -1) return 0;
50      Rectangle rg = nodes[id].range;
51      if (rg.lx >= rec.lx && rg.rx <= rec.rx && rg.ly >= rec.ly
    ↪ && rg.ry <= rec.ry) {
52        return nodes[id].num;
53      }
54      int ans = 0;
55      if (depth % 2) { // pruning
56        if (rec.lx <= nodes[id].point.x) ans +=
    ↪ inner_query(nodes[id].lc, rec, depth + 1);
57        if (rec.rx >= nodes[id].point.x) ans +=
    ↪ inner_query(nodes[id].rc, rec, depth + 1);
58      } else {
59        if (rec.ly <= nodes[id].point.y) ans +=
    ↪ inner_query(nodes[id].lc, rec, depth + 1);
60        if (rec.ry >= nodes[id].point.y) ans +=
    ↪ inner_query(nodes[id].rc, rec, depth + 1);
61      }
62      if (is_in(nodes[id].point, rec)) ans += 1;
63      return ans;
64    }
65    int query(const Rectangle &rec) { return inner_query(root,
    ↪ rec, 0); }
66  };
```

Link/Cut Tree

```
1  struct Node {
2    Node *ch[2], *p;
3    int id;
4    bool rev;
5    Node(int id) : ch{nullptr, nullptr}, p(nullptr), id(id),
    ↪ rev(false) {}
6    friend void reverse(Node *p) {
7      if (p != nullptr) {
8        swap(p->ch[0], p->ch[1]);
9        p->rev ^= 1;
10      }
11    }
12    void push() {
13      if (rev) {
14        reverse(ch[0]);
15        reverse(ch[1]);
16        rev = false;
17      }
18    }
19    void pull() {}
20    bool is_root() { return p == nullptr || p->ch[0] != this &&
    ↪ p->ch[1] != this; }
21    bool pos() { return p->ch[1] == this; }
22    void rotate() {
23      Node *q = p;
24      bool x = !pos();
25      q->ch[!x] = ch[x];
26      if (ch[x] != nullptr) ch[x]->p = q;
27      p = q->p;
28      if (!q->is_root()) q->p->ch[q->pos()] = this;
29      ch[x] = q;
30      q->p = this;
31      pull();
32      q->pull();
33    }
34    void splay() {
35      vector<Node *> s;
36      for (Node *i = this; !i->is_root(); i = i->p)
    ↪ s.push_back(i->p);
37      while (!s.empty()) s.back()->push(), s.pop_back();
38      push();
39      while (!is_root()) {
```

```
40    if (!p->is_root()) {
41      if (pos() == p->pos()) {
42        p->rotate();
43      } else {
44        rotate();
45      }
46    }
47    rotate();
48  }
49  pull();
50 }
51 void access() {
52   for (Node *i = this, *q = nullptr; i != nullptr; q = i, i
 ↪ = i->p) {
53     i->splay();
54     i->ch[1] = q;
55     i->pull();
56   }
57   splay();
58 }
59 void makeroot() {
60   access();
61   reverse(this);
62 }
63 };
64 void link(Node *x, Node *y) {
65   x->makeroot();
66   x->p = y;
67 }
68 void split(Node *x, Node *y) {
69   x->makeroot();
70   y->access();
71 }
72 void cut(Node *x, Node *y) {
73   split(x, y);
74   x->p = y->ch[0] = nullptr;
75   y->pull();
76 }
77 bool connected(Node *p, Node *q) {
78     p->access();
79     q->access();
80     return p->p != nullptr;
81 }
```

Li-Chao Tree

```
1 template <typename T, T LO, T HI, class C = less<T>> struct
 ↪ LiChaoTree {
2   struct Line {
3     T m, b;
4     int l = -1, r = -1;
5     Line(T m, T b) : m(m), b(b) {}
6     T operator()(T x) { return m*x + b; }
7   };
8   vector<Line> tree;
9   T query(int id, T l, T r, T x) {
10    auto& line = tree[id];
11    T mid = (l + r)/2, ans = line(x);
12    if(line.l != -1 && x <= mid)
13      ans = _choose(ans, query(line.l, l, mid, x));
14    else if(line.r != -1 && x > mid)
15      ans = _choose(ans, query(line.r, mid + 1, r, x));
16    return ans;
17  }
18  T query(T x) { return query(0, LO, HI, x); }
19  int add(int id, T l, T r, T m, T b) {
20    if(tree.empty() || id == -1) {
21      tree.push_back(Line(m, b));
22      return (int)tree.size() - 1;
23    }
24    auto& line = tree[id];
25    T mid = (l + r)/2;
26    if(C()(m*mid + b, line(mid))) {
27      swap(m, line.m);
28      swap(b, line.b);
29    }
```

CHT

```
30    if(C()(m, line.m) && l != r) tree[id].r = add(line.r, mid
 ↪ + 1, r, m, b);
31    else if(l != r) tree[id].l = add(line.l, l, mid, m, b);
32    return id;
33  }
34  void add(T m, T b) { add(0, LO, HI, m, b); }
35  T _choose(T x, T y) { return C()(x, y) ? x : y; }
36 };
```

```
1 struct line {
2   static bool Q; mutable ll k, m, p;
3   bool operator<(const line& o) const { return Q ? p < o.p : k
 ↪ < o.k; }
4 };
5 bool line::Q = false;
6 struct lines : multiset<line> {
7   //(for doubles, use inf = 1/.0, div(a,b) = a/b)
8   ll div(ll a, ll b) { return a / b - ((a ^ b) < 0 && a % b);
 ↪ }
9   bool isect(iterator x, iterator y) {
10    if (y == end()) return x->p = inf, false;
11    if (x->k == y->k) {
12      x->p = x->m > y->m ? inf : -inf;
13    } else {
14      x->p = div(y->m - x->m, x->k - y->k);
15    }
16    return x->p >= y->p;
17  }
18  void add(ll k, ll m) {
19    line::Q = false;
20    auto z = insert({k, m, 0}), y = z++, x = y;
21    while (isect(y, z)) z = erase(z);
22    if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
23    while ((y = x) != begin() && (--x)->p >= y->p) isect(x,
 ↪ erase(y));
24  }
25  ll query(ll x) {
26    line::Q = true; auto l = lower_bound({0, 0, x});
27    return l->k * x + l->m;
28  }
29 };
```

Bitset

```
1 struct Bitset {
2   using ull = unsigned long long;
3   static const int BLOCKSZ = CHAR_BIT * sizeof(ull);
4   int n;
5   vector<ull> a;
6   Bitset(int n) : n(n) { a.resize((n + BLOCKSZ - 1)/BLOCKSZ);
 ↪ }
7   void set(int p, bool v) {
8     ull b = (1ull << (p - BLOCKSZ * (p/BLOCKSZ)));
9     v ? a[p/BLOCKSZ] |= b : a[p/BLOCKSZ] &= ~b;
10  }
11  void flip(int p) {
12    ull b = (1ull << (p - BLOCKSZ * (p/BLOCKSZ)));
13    a[p/BLOCKSZ] ^= b;
14  }
15  string to_string() {
16    string res;
17    FOR(i,n) res += operator[](i) ? '1' : '0';
18    return res;
19  }
20  int count() {
21    int sz = (int)a.size(), ret = 0;
22    FOR(i,sz) ret += __builtin_popcountll(a[i]);
23    return ret;
24  }
25  int size() { return n; }
26  bool operator[](int p) { return a[p/BLOCKSZ] & (1ull << (p -
 ↪ BLOCKSZ * (p/BLOCKSZ))); }
27  bool operator==(const Bitset& other) {
28    if(n != other.n) return false;
```

```
29    FOR(i,(int)a.size()) if(a[i] != other.a[i]) return false;
30    return true;
31  }
32  bool operator!=(const Bitset& other) { return
   ↪ !operator==(other); }
33  Bitset& operator<<=(int x) {
34    int sz = (int)a.size(), sh = x/BLOCKSZ, xtra = x - sh *
   ↪ BLOCKSZ, rem = BLOCKSZ - xtra;
35    if(!xtra) FOR(i,sz-sh) a[i] = a[i + sh] >> xtra;
36    else {
37      FOR(i,sz-sh-1) a[i] = (a[i + sh] >> xtra) | (a[i + sh +
   ↪ 1] << rem);
38      if(sz - sh - 1 >= 0) a[sz - sh - 1] = a[sz - 1] >> xtra;
39    }
40    for(int i = max(0, sz - sh); i <= sz - 1; i++) a[i] = 0;
41    return *this;
42  }
43  Bitset& operator>>=(int x) {
44    int sz = (int)a.size(), sh = x/BLOCKSZ, xtra = x - sh *
   ↪ BLOCKSZ, rem = BLOCKSZ - xtra;
45    if(!xtra) for(int i = sz - 1; i >= sh; i--) a[i] = a[i -
   ↪ sh] << xtra;
46    else {
47      for(int i = sz - 1; i > sh; i--) a[i] = (a[i - sh] <<
   ↪ xtra) | (a[i - sh - 1] >> rem);
48      if(sh < sz) a[sh] = a[0] << xtra;
49    }
50    for(int i = min(sz-1,sh-1); i >= 0; i--) a[i] = 0;
51    a[sz - 1] <<= (sz * BLOCKSZ - n);
52    a[sz - 1] >>= (sz * BLOCKSZ - n);
53    return *this;
54  }
55  Bitset& operator&=(const Bitset& other) {
   ↪ FOR(i,(int)a.size()) a[i] &= other.a[i]; return *this; }
56  Bitset& operator|=(const Bitset& other) {
   ↪ FOR(i,(int)a.size()) a[i] |= other.a[i]; return *this; }
57  Bitset& operator^=(const Bitset& other) {
   ↪ FOR(i,(int)a.size()) a[i] ^= other.a[i]; return *this; }
58  Bitset operator~() {
59    int sz = (int)a.size();
60    Bitset ret(*this);
61    FOR(i,sz) ret.a[i] = ~ret.a[i];
62    ret.a[sz - 1] <<= (sz * BLOCKSZ - n);
63    ret.a[sz - 1] >>= (sz * BLOCKSZ - n);
64    return ret;
65  }
66  Bitset operator&(const Bitset& other) { return
   ↪ (Bitset(*this) &= other); }
67  Bitset operator|(const Bitset& other) { return
   ↪ (Bitset(*this) |= other); }
68  Bitset operator^(const Bitset& other) { return
   ↪ (Bitset(*this) ^= other); }
69  Bitset operator<<(int x) { return (Bitset(*this) <<= x); }
70  Bitset operator>>(int x) { return (Bitset(*this) >>= x); }
71 };
```

Geometry

Basic stuff

```
1  using ll = long long;
2  using ld = long double;
3
4  constexpr auto eps = 1e-8;
5  const auto PI = acos(-1);
6  int sgn(ld x) { return (abs(x) <= eps) ? 0 : (x < 0 ? -1 : 1);
   ↪ }
7
8  struct Point {
9    ld x = 0, y = 0;
10   Point() = default;
11   Point(ld _x, ld _y) : x(_x), y(_y) {}
12   bool operator<(const Point &p) const { return !sgn(p.x - x)
   ↪ ? sgn(y - p.y) < 0 : x < p.x; }
13   bool operator==(const Point &p) const { return !sgn(p.x - x)
   ↪ && !sgn(p.y - y); }
```

```
14   Point operator+(const Point &p) const { return {x + p.x, y +
   ↪ p.y}; }
15   Point operator-(const Point &p) const { return {x - p.x, y -
   ↪ p.y}; }
16   Point operator*(ld a) const { return {x * a, y * a}; }
17   Point operator/(ld a) const { return {x / a, y / a}; }
18   auto operator*(const Point &p) const { return x * p.x + y *
   ↪ p.y; }  // dot
19   auto operator^(const Point &p) const { return x * p.y - y *
   ↪ p.x; }  // cross
20   friend auto &operator>>(istream &i, Point &p) { return i >>
   ↪ p.x >> p.y; }
21   friend auto &operator<<(ostream &o, Point p) { return o <<
   ↪ p.x << ' ' << p.y; }
22 };
23
24 struct Line {
25   Point s = {0, 0}, e = {0, 0};
26   Line() = default;
27   Line(Point _s, Point _e) : s(_s), e(_e) {}
28   friend auto &operator>>(istream &i, Line &l) { return i >>
   ↪ l.s >> l.e; }  // ((x1, y1), (x2, y2)
29 };
30
31 struct Segment : Line {
32   using Line::Line;
33 };
34
35 struct Circle {
36   Point o = {0, 0};
37   ld r = 0;
38   Circle() = default;
39   Circle(Point _o, ld _r) : o(_o), r(_r) {}
40 };
```

```
1  auto dist2(const Point &a) { return a * a; }
2  auto dist2(const Point &a, const Point &b) { return dist2(a -
   ↪ b); }
3  auto dist(const Point &a) { return sqrt(dist2(a)); }
4  auto dist(const Point &a, const Point &b) { return
   ↪ sqrt(dist2(a - b)); }
5  auto dist(const Point &a, const Line &l) { return abs((a -
   ↪ l.s) ^ (l.e - l.s)) / dist(l.s, l.e); }
6  auto dist(const Point &p, const Segment &l) {
7    if (l.s == l.e) return dist(p, l.s);
8    auto d = dist2(l.s, l.e), t = min(d, max((ld)0, (p - l.s) *
   ↪ (l.e - l.s)));
9    return dist((p - l.s) * d, (l.e - l.s) * t) / d;
10 }
11 /* Needs is_intersect
12 auto dist(const Segment &l1, const Segment &l2) {
13   if (is_intersect(l1, l2)) return (ld)0;
14   return min({dist(l1.s, l2), dist(l1.e, l2), dist(l2.s, l1),
   ↪ dist(l2.e, l1)});
15 } */
16
17 Point perp(const Point &p) { return Point(-p.y, p.x); }
18
19 auto rad(const Point &p) { return atan2(p.y, p.x); }
```

Transformation

```
1  Point project(const Point &p, const Line &l) {
2    return l.s + ((l.e - l.s) * ((l.e - l.s) * (p - l.s))) /
   ↪ dist2(l.e - l.s);
3  }
4
5  Point reflect(const Point &p, const Line &l) {
6    return project(p, l) * 2 - p;
7  }
8
9  Point dilate(const Point &p, ld scale_x = 1, ld scale_y = 1) {
   ↪ return Point(p.x * scale_x, p.y * scale_y); }
10 Line dilate(const Line &l, ld scale_x = 1, ld scale_y = 1) {
   ↪ return Line(dilate(l.s, scale_x, scale_y), dilate(l.e,
   ↪ scale_x, scale_y)); }
```

```
11  Segment dilate(const Segment &l, ld scale_x = 1, ld scale_y =
    ↪  1) { return Segment(dilate(l.s, scale_x, scale_y),
    ↪  dilate(l.e, scale_x, scale_y)); }
12  vector<Point> dilate(const vector<Point> &p, ld scale_x = 1,
    ↪  ld scale_y = 1) {
13    int n = p.size();
14    vector<Point> res(n);
15    for (int i = 0; i < n; i++)
16      res[i] = dilate(p[i], scale_x, scale_y);
17    return res;
18  }
19
20  Point rotate(const Point &p, ld a) { return Point(p.x * cos(a)
    ↪  - p.y * sin(a), p.x * sin(a) + p.y * cos(a)); }
21  Line rotate(const Line &l, ld a) { return Line(rotate(l.s, a),
    ↪  rotate(l.e, a)); }
22  Segment rotate(const Segment &l, ld a) { return
    ↪  Segment(rotate(l.s, a), rotate(l.e, a)); }
23  Circle rotate(const Circle &c, ld a) { return
    ↪  Circle(rotate(c.o, a), c.r); }
24  vector<Point> rotate(const vector<Point> &p, ld a) {
25    int n = p.size();
26    vector<Point> res(n);
27    for (int i = 0; i < n; i++)
28      res[i] = rotate(p[i], a);
29    return res;
30  }
31
32  Point translate(const Point &p, ld dx = 0, ld dy = 0) { return
    ↪  Point(p.x + dx, p.y + dy); }
33  Line translate(const Line &l, ld dx = 0, ld dy = 0) { return
    ↪  Line(translate(l.s, dx, dy), translate(l.e, dx, dy)); }
34  Segment translate(const Segment &l, ld dx = 0, ld dy = 0) {
    ↪  return Segment(translate(l.s, dx, dy), translate(l.e, dx,
    ↪  dy)); }
35  Circle translate(const Circle &c, ld dx = 0, ld dy = 0) {
    ↪  return Circle(translate(c.o, dx, dy), c.r); }
36  vector<Point> translate(const vector<Point> &p, ld dx = 0, ld
    ↪  dy = 0) {
37    int n = p.size();
38    vector<Point> res(n);
39    for (int i = 0; i < n; i++)
40      res[i] = translate(p[i], dx, dy);
41    return res;
42  }
```

Relation

```
1   enum class Relation { SEPARATE, EX_TOUCH, OVERLAP, IN_TOUCH,
    ↪  INSIDE };
2   Relation get_relation(const Circle &a, const Circle &b) {
3     auto c1c2 = dist(a.o, b.o);
4     auto r1r2 = a.r + b.r, diff = abs(a.r - b.r);
5     if (sgn(c1c2 - r1r2) > 0) return Relation::SEPARATE;
6     if (sgn(c1c2 - r1r2) == 0) return Relation::EX_TOUCH;
7     if (sgn(c1c2 - diff) > 0) return Relation::OVERLAP;
8     if (sgn(c1c2 - diff) == 0) return Relation::IN_TOUCH;
9     return Relation::INSIDE;
10  }
11
12  auto get_cos_from_triangle(ld a, ld b, ld c) { return (a * a +
    ↪  b * b - c * c) / (2.0 * a * b); }
13
14  bool on_line(const Line &l, const Point &p) { return !sgn((l.s
    ↪  - p) ^ (l.e - p)); }
15
16  bool on_segment(const Segment &l, const Point &p) {
17    return !sgn((l.s - p) ^ (l.e - p)) && sgn((l.s - p) * (l.e -
    ↪  p)) <= 0;
18  }
19
20  bool on_segment2(const Segment &l, const Point &p) { // assume
    ↪  p on Line l
21    if (l.s == p || l.e == p) return true;
22    if (min(l.s, l.e) < p && p < max(l.s, l.e)) return true;
23    return false;
24  }
```

```
25
26  bool is_parallel(const Line &a, const Line &b) { return
    ↪  !sgn((a.s - a.e) ^ (b.s - b.e)); }
27  bool is_orthogonal(const Line &a, const Line &b) { return
    ↪  !sgn((a.s - a.e) * (b.s - b.e)); }
28
29  int is_intersect(const Segment &a, const Segment &b) {
30    auto d1 = sgn((a.e - a.s) ^ (b.s - a.s)), d2 = sgn((a.e -
    ↪  a.s) ^ (b.e - a.s));
31    auto d3 = sgn((b.e - b.s) ^ (a.s - b.s)), d4 = sgn((b.e -
    ↪  b.s) ^ (a.e - b.s));
32    if (d1 * d2 < 0 && d3 * d4 < 0) return 2;  // intersect at
    ↪  non-end point
33    return (d1 == 0 && sgn((b.s - a.s) * (b.s - a.e)) <= 0) ||
34        (d2 == 0 && sgn((b.e - a.s) * (b.e - a.e)) <= 0) ||
35        (d3 == 0 && sgn((a.s - b.s) * (a.s - b.e)) <= 0) ||
36        (d4 == 0 && sgn((a.e - b.s) * (a.e - b.e)) <= 0);
37  }
38
39  int is_intersect(const Line &a, const Segment &b) {
40    auto d1 = sgn((a.e - a.s) ^ (b.s - a.s)), d2 = sgn((a.e -
    ↪  a.s) ^ (b.e - a.s));
41    if (d1 * d2 < 0) return 2;  // intersect at non-end point
42    return d1 == 0 || d2 == 0;
43  }
44
45  Point intersect(const Line &a, const Line &b) {
46    auto u = a.e - a.s, v = b.e - b.s;
47    auto t = ((b.s - a.s) ^ v) / (u ^ v);
48    return a.s + u * t;
49  }
50
51  int is_intersect(const Circle &c, const Line &l) {
52    auto d = dist(c.o, l);
53    return sgn(d - c.r) < 0 ? 2 : !sgn(d - c.r);
54  }
55
56  vector<Point> intersect(const Circle &a, const Circle &b) {
57    auto relation = get_relation(a, b);
58    if (relation == Relation::INSIDE || relation ==
    ↪  Relation::SEPARATE) return {};
59    auto vec = b.o - a.o;
60    auto d2 = dist2(vec);
61    auto p = (d2 + a.r * a.r - b.r * b.r) / ((long double)2 *
    ↪  d2), h2 = a.r * a.r - p * p * d2;
62    auto mid = a.o + vec * p, per = perp(vec) * sqrt(max((long
    ↪  double)0, h2) / d2);
63    if (relation == Relation::OVERLAP)
64      return {mid + per, mid - per};
65    else
66      return {mid};
67  }
68
69  vector<Point> intersect(const Circle &c, const Line &l) {
70    if (!is_intersect(c, l)) return {};
71    auto v = l.e - l.s, t = v / dist(v);
72    Point a = l.s + t * ((c.o - l.s) * t);
73    auto d = sqrt(max((ld)0, c.r * c.r - dist2(c.o, a)));
74    if (!sgn(d)) return {a};
75    return {a - t * d, a + t * d};
76  }
77
78  int in_poly(const vector<Point> &p, const Point &a) {
79    int cnt = 0, n = (int)p.size();
80    for (int i = 0; i < n; i++) {
81      auto q = p[(i + 1) % n];
82      if (on_segment(Segment(p[i], q), a)) return 1;  // on the
    ↪  edge of the polygon
83      cnt ^= ((a.y < p[i].y) - (a.y < q.y)) * ((p[i] - a) ^ (q -
    ↪  a)) > 0;
84    }
85    return cnt ? 2 : 0;
86  }
87
88  int is_intersect(const vector<Point> &p, const Line &a) {
89    // 1: touching, >=2: intersect count
90    int cnt = 0, edge_cnt = 0, n = (int)p.size();
```

```cpp
91      for (int i = 0; i < n; i++) {
92        auto q = p[(i + 1) % n];
93        if (on_line(a, p[i]) && on_line(a, q)) return -1;  //
    ↪ infinity
94        auto t = is_intersect(a, Segment(p[i], q));
95        (t == 1) && edge_cnt++, (t == 2) && cnt++;
96      }
97      return cnt + edge_cnt / 2;
98    }
99
100   vector<Point> tangent(const Circle &c, const Point &p) {
101     auto d = dist(c.o, p), l = c.r * c.r / d, h = sqrt(c.r * c.r
    ↪ - l * l);
102     auto v = (p - c.o) / d;
103     return {c.o + v * l + perp(v) * h, c.o + v * l - perp(v) *
    ↪ h};
104   }
105
106   Circle get_circumscribed(const Point &a, const Point &b, const
    ↪ Point &c) {
107     Line u((a + b) / 2, ((a + b) / 2) + perp(b - a));
108     Line v((b + c) / 2, ((b + c) / 2) + perp(c - b));
109     auto o = intersect(u, v);
110     return Circle(o, dist(o, a));
111   }
112
113   Circle get_inscribed(const Point &a, const Point &b, const
    ↪ Point &c) {
114     auto l1 = dist(b - c), l2 = dist(c - a), l3 = dist(a - b);
115     Point o = (a * l1 + b * l2 + c * l3) / (l1 + l2 + l3);
116     return Circle(o, dist(o, Line(a, b)));
117   }
118
119   pair<ld, ld> get_centroid(const vector<Point> &p) {
120     int n = (int)p.size();
121     ld x = 0, y = 0, sum = 0;
122     auto a = p[0], b = p[1];
123     for (int i = 2; i < n; i++) {
124       auto c = p[i];
125       auto s = area({a, b, c});
126       sum += s;
127       x += s * (a.x + b.x + c.x);
128       y += s * (a.y + b.y + c.y);
129       swap(b, c);
130     }
131     return {x / (3 * sum), y / (3 * sum)};
132   }
```

Area

```cpp
1   auto area(const vector<Point> &p) {
2     int n = (int)p.size();
3     long double area = 0;
4     for (int i = 0; i < n; i++) area += p[i] ^ p[(i + 1) % n];
5     return area / 2.0;
6   }
7
8   auto area(const Point &a, const Point &b, const Point &c) {
9     return ((long double)((b - a) ^ (c - a))) / 2.0;
10  }
11
12  auto area2(const Point &a, const Point &b, const Point &c) {
    ↪ return (b - a) ^ (c - a); }
13
14  auto area_intersect(const Circle &c, const vector<Point> &ps)
    ↪ {
15    int n = (int)ps.size();
16    auto arg = [&](const Point &p, const Point &q) { return
    ↪ atan2(p ^ q, p * q); };
17    auto tri = [&](const Point &p, const Point &q) {
18      auto r2 = c.r * c.r / (long double)2;
19      auto d = q - p;
20      auto a = d * p / dist2(d), b = (dist2(p) - c.r * c.r) /
    ↪ dist2(d);
21      long double det = a * a - b;
22      if (sgn(det) <= 0) return arg(p, q) * r2;
```

```cpp
23      auto s = max((long double)0, -a - sqrt(det)), t =
    ↪ min((long double)1, -a + sqrt(det));
24      if (sgn(t) < 0 || sgn(1 - s) <= 0) return arg(p, q) * r2;
25      auto u = p + d * s, v = p + d * t;
26      return arg(p, u) * r2 + (u ^ v) / 2 + arg(v, q) * r2;
27    };
28    long double sum = 0;
29    for (int i = 0; i < n; i++) sum += tri(ps[i] - c.o, ps[(i +
    ↪ 1) % n] - c.o);
30    return sum;
31  }
32
33  auto adaptive_simpson(ld _l, ld _r, function<ld(ld)> f) {
34    auto simpson = [&](ld l, ld r) { return (r - l) * (f(l) + 4
    ↪ * f((l + r) / 2) + f(r)) / 6; };
35    function<ld(ld, ld, ld)> asr = [&](ld l, ld r, ld s) {
36      auto mid = (l + r) / 2;
37      auto left = simpson(l, mid), right = simpson(mid, r);
38      if (!sgn(left + right - s)) return left + right;
39      return asr(l, mid, left) + asr(mid, r, right);
40    };
41    return asr(_l, _r, simpson(_l, _r));
42  }
43
44  vector<Point> half_plane_intersect(vector<Line> &L) {
45    int n = (int)L.size(), l = 0, r = 0;  // [left, right]
46    sort(L.begin(), L.end(),
47         [](const Line &a, const Line &b) { return rad(a.s -
    ↪ a.e) < rad(b.s - b.e); });
48    vector<Point> p(n), res;
49    vector<Line> q(n);
50    q[0] = L[0];
51    for (int i = 1; i < n; i++) {
52      while (l < r && sgn((L[i].e - L[i].s) ^ (p[r - 1] -
    ↪ L[i].s)) <= 0) r--;
53      while (l < r && sgn((L[i].e - L[i].s) ^ (p[l] - L[i].s))
    ↪ <= 0) l++;
54      q[++r] = L[i];
55      if (sgn((q[r].e - q[r].s) ^ (q[r - 1].e - q[r - 1].s)) ==
    ↪ 0) {
56        r--;
57        if (sgn((q[r].e - q[r].s) ^ (L[i].s - q[r].s)) > 0) q[r]
    ↪ = L[i];
58      }
59      if (l < r) p[r - 1] = intersect(q[r - 1], q[r]);
60    }
61    while (l < r && sgn((q[l].e - q[l].s) ^ (p[r - 1] - q[l].s))
    ↪ <= 0) r--;
62    if (r - l <= 1) return {};
63    p[r] = intersect(q[r], q[l]);
64    return vector<Point>(p.begin() + l, p.begin() + r + 1);
65  }
```

Convex

```cpp
1   vector<Point> get_convex(vector<Point> &points, bool
    ↪ allow_collinear = false) {
2     // strict, no repeat, two pass
3     sort(points.begin(), points.end());
4     points.erase(unique(points.begin(), points.end()),
    ↪ points.end());
5     vector<Point> L, U;
6     for (auto &t : points) {
7       for (ll sz = L.size(); sz > 1 && (sgn((t - L[sz - 2]) ^
    ↪ (L[sz - 1] - L[sz - 2])) >= 0);
8            L.pop_back(), sz = L.size()) {
9       }
10      L.push_back(t);
11    }
12    for (auto &t : points) {
13      for (ll sz = U.size(); sz > 1 && (sgn((t - U[sz - 2]) ^
    ↪ (U[sz - 1] - U[sz - 2])) <= 0);
14           U.pop_back(), sz = U.size()) {
15      }
16      U.push_back(t);
17    }
```

11

```
18      // contain repeats if all collinear, use a set to remove
   ↪    repeats
19      if (allow_collinear) {
20        for (int i = (int)U.size() - 2; i >= 1; i--)
   ↪      L.push_back(U[i]);
21      } else {
22        set<Point> st(L.begin(), L.end());
23        for (int i = (int)U.size() - 2; i >= 1; i--) {
24          if (st.count(U[i]) == 0) L.push_back(U[i]),
   ↪        st.insert(U[i]);
25        }
26      }
27      return L;
28    }
29
30    vector<Point> get_convex2(vector<Point> &points, bool
   ↪    allow_collinear = false) {  // strict, no repeat, one pass
31      nth_element(points.begin(), points.begin(), points.end());
32      sort(points.begin() + 1, points.end(), [&](const Point &a,
   ↪    const Point &b) {
33        int rad_diff = sgn((a - points[0]) ^ (b - points[0]));
34        return !rad_diff ? (dist2(a - points[0]) < dist2(b -
   ↪    points[0])) : (rad_diff > 0);
35      });
36      if (allow_collinear) {
37        int i = (int)points.size() - 1;
38        while (i >= 0 && !sgn((points[i] - points[0]) ^ (points[i]
   ↪    - points.back()))) i--;
39        reverse(points.begin() + i + 1, points.end());
40      }
41      vector<Point> hull;
42      for (auto &t : points) {
43        for (ll sz = hull.size();
44            sz > 1 && (sgn((t - hull[sz - 2]) ^ (hull[sz - 1] -
   ↪    hull[sz - 2])) >= allow_collinear);
45            hull.pop_back(), sz = hull.size()) {
46        }
47        hull.push_back(t);
48      }
49      return hull;
50    }
51
52    vector<Point> get_convex_safe(vector<Point> points, bool
   ↪    allow_collinear = false) {
53      return get_convex(points, allow_collinear);
54    }
55
56    vector<Point> get_convex2_safe(vector<Point> points, bool
   ↪    allow_collinear = false) {
57      return get_convex2(points, allow_collinear);
58    }
59
60    bool is_convex(const vector<Point> &p, bool allow_collinear =
   ↪    false) {
61      int n = p.size();
62      int lo = 1, hi = -1;
63      for (int i = 0; i < n; i++) {
64        int cur = sgn((p[(i + 2) % n] - p[(i + 1) % n]) ^ (p[(i +
   ↪    1) % n] - p[i]));
65        lo = min(lo, cur); hi = max(hi, cur);
66      }
67      return allow_collinear ? (hi - lo) < 2 : (lo == hi && lo);
68    }
69
70    auto rotating_calipers(const vector<Point> &hull) {
71      // use get_convex2
72      int n = (int)hull.size();  // return the square of longest
   ↪    dist
73      assert(n > 1);
74      if (n <= 2) return dist2(hull[0], hull[1]);
75      ld res = 0;
76      for (int i = 0, j = 2; i < n; i++) {
77        auto d = hull[i], e = hull[(i + 1) % n];
78        while (area2(d, e, hull[j]) < area2(d, e, hull[(j + 1) %
   ↪    n])) j = (j + 1) % n;
79        res = max(res, max(dist2(d, hull[j]), dist2(e, hull[j])));
80      }
```

```
81      return res;
82    }
83
84    // Find polygon cut to the left of l
85    vector<Point> convex_cut(const vector<Point> &p, const Line
   ↪    &l) {
86      int n = p.size();
87      vector<Point> cut;
88      for (int i = 0; i < n; i++) {
89        auto a = p[i], b = p[(i + 1) % n];
90        if (sgn((l.e - l.s) ^ (a - l.s)) >= 0)
91          cut.push_back(a);
92        if (sgn((l.e - l.s) ^ (a - l.s)) * sgn((l.e - l.s) ^ (b -
   ↪    l.s)) == -1)
93          cut.push_back(intersect(Line(a, b), l));
94      }
95      return cut;
96    }
97
98    // Sort by angle in range [0, 2pi)
99    template <class RandomIt>
100   void polar_sort(RandomIt first, RandomIt last, Point origin =
   ↪    Point(0, 0)) {
101     auto get_quad = [&](const Point& p) {
102       Point diff = p - origin;
103       if (diff.x > 0 && diff.y >= 0) return 1;
104       if (diff.x <= 0 && diff.y > 0) return 2;
105       if (diff.x < 0 && diff.y <= 0) return 3;
106       return 4;
107     };
108     auto polar_cmp = [&](const Point& p1, const Point& p2) {
109       int q1 = get_quad(p1), q2 = get_quad(p2);
110       if (q1 != q2) return q1 < q2;
111       return ((p1 - origin) ^ (p2 - origin)) > 0;
112     };
113     sort(first, last, polar_cmp);
114   }
```

Basic 3D

```
1     using ll = long long;
2     using ld = long double;
3
4     constexpr auto eps = 1e-8;
5     const auto PI = acos(-1);
6     int sgn(ld x) { return (abs(x) <= eps) ? 0 : (x < 0 ? -1 : 1);
   ↪    }
7
8     struct Point3D {
9       ld x = 0, y = 0, z = 0;
10      Point3D() = default;
11      Point3D(ld _x, ld _y, ld _z) : x(_x), y(_y), z(_z) {}
12      bool operator<(const Point3D &p) const { return !sgn(p.x -
   ↪    x) ? (!sgn(p.y - y) ? sgn(p.z - z) < 0 : y < p.y) : x <
   ↪    p.x; }
13      bool operator==(const Point3D &p) const { return !sgn(p.x -
   ↪    x) && !sgn(p.y - y) && !sgn(p.z - z); }
14      Point3D operator+(const Point3D &p) const { return {x + p.x,
   ↪    y + p.y, z + p.z}; }
15      Point3D operator-(const Point3D &p) const { return {x - p.x,
   ↪    y - p.y, z - p.z}; }
16      Point3D operator*(ld a) const { return {x * a, y * a, z *
   ↪    a}; }
17      Point3D operator/(ld a) const { return {x / a, y / a, z /
   ↪    a}; }
18      auto operator*(const Point3D &p) const { return x * p.x + y
   ↪    * p.y + z * p.z; }  // dot
19      Point3D operator^(const Point3D &p) const { return {y * p.z
   ↪    - z * p.y, z * p.x - x * p.z, x * p.y - y * p.x}; }  //
   ↪    cross
20      friend auto &operator>>(istream &i, Point3D &p) { return i
   ↪    >> p.x >> p.y >> p.z; }
21    };
22
23    struct Line3D {
24      Point3D s = {0, 0, 0}, e = {0, 0, 0};
25      Line3D() = default;
```

```
26      Line3D(Point3D _s, Point3D _e) : s(_s), e(_e) {}
27    };
28
29    struct Segment3D : Line3D {
30      using Line3D::Line3D;
31    };
32
33    auto dist2(const Point3D &a) { return a * a; }
34    auto dist2(const Point3D &a, const Point3D &b) { return
   ↪ dist2(a - b); }
35    auto dist(const Point3D &a) { return sqrt(dist2(a)); }
36    auto dist(const Point3D &a, const Point3D &b) { return
   ↪ sqrt(dist2(a - b)); }
37    auto dist(const Point3D &a, const Line3D &l) { return dist((a
   ↪ - l.s) ^ (l.e - l.s)) / dist(l.s, l.e); }
38    auto dist(const Point3D &p, const Segment3D &l) {
39      if (l.s == l.e) return dist(p, l.s);
40      auto d = dist2(l.s, l.e), t = min(d, max((ld)0, (p - l.s) *
   ↪ (l.e - l.s)));
41      return dist((p - l.s) * d, (l.e - l.s) * t) / d;
42    }
```

Miscellaneous

```
1    tuple<int,int,ld> closest_pair(vector<Point> &p) {
2      using Pt = pair<Point,int>;
3      int n = p.size();
4      assert(n > 1);
5      vector<Pt> pts(n), buf;
6      for (int i = 0; i < n; i++) pts[i] = {p[i], i};
7      sort(pts.begin(), pts.end());
8      buf.reserve(n);
9      auto cmp_y = [](const Pt& p1, const Pt& p2) { return
   ↪ p1.first.y < p2.first.y; };
10     function<tuple<int,int,ld>(int, int)> recurse = [&](int l,
   ↪ int r) -> tuple<int,int,ld> {
11       int i = pts[l].second, j = pts[l + 1].second;
12       ld d = dist(pts[l].first, pts[l + 1].first);
13       if (r - l < 5) {
14         for (int a = l; a < r; a++) for (int b = a + 1; b < r;
   ↪ b++) {
15           ld cur = dist(pts[a].first, pts[b].first);
16           if (cur < d) { i = pts[a].second; j = pts[b].second; d
   ↪ = cur; }
17         }
18         sort(pts.begin() + l, pts.begin() + r, cmp_y);
19       }
20       else {
21         int mid = (l + r)/2;
22         ld x = pts[mid].first.x;
23         auto [li, lj, ldist] = recurse(l, mid);
24         auto [ri, rj, rdist] = recurse(mid, r);
25         if (ldist < rdist) { i = li; j = lj; d = ldist; }
26         else { i = ri; j = rj; d = rdist; }
27         inplace_merge(pts.begin() + l, pts.begin() + mid,
   ↪ pts.begin() + r, cmp_y);
28         buf.clear();
29         for (int a = l; a < r; a++) {
30           if (abs(x - pts[a].first.x) >= d) continue;
31           for (int b = buf.size() - 1; b >= 0; b--) {
32             if (pts[a].first.y - buf[b].first.y >= d) break;
33             ld cur = dist(pts[a].first, buf[b].first);
34             if (cur < d) { i = pts[a].second; j = buf[b].second;
   ↪ d = cur; }
35           }
36           buf.push_back(pts[a]);
37         }
38       }
39       return {i, j, d};
40     };
41     return recurse(0, n);
42   }
43
44   Line abc_to_line(ld a, ld b, ld c) {
45     assert(!sgn(a) || !sgn(b));
46     if(a == 0) return Line(Point(0, -c/b), Point(1, -c/b));
47     if(b == 0) return Line(Point(-c/a, 0), Point(-c/a, 1));
```

```
48     Point s(0, -c/b), e(1, (-c - a)/b), diff = e - s;
49     return Line(s, s + diff/dist(diff));
50   }
51
52   tuple<ld,ld,ld> line_to_abc(const Line& l) {
53     Point diff = l.e - l.s;
54     return {-diff.y, diff.x, -(diff ^ l.s)};
55   }
```

Graph Theory

Max Flow

```
1    struct Edge {
2      int from, to, cap, remain;
3    };
4
5    struct Dinic {
6      int n;
7      vector<Edge> e;
8      vector<vector<int>> g;
9      vector<int> d, cur;
10     Dinic(int _n) : n(_n), g(n), d(n), cur(n) {}
11     void add_edge(int u, int v, int c) {
12       g[u].push_back((int)e.size());
13       e.push_back({u, v, c, c});
14       g[v].push_back((int)e.size());
15       e.push_back({v, u, 0, 0});
16     }
17     ll max_flow(int s, int t) {
18       int inf = 1e9;
19       auto bfs = [&]() {
20         fill(d.begin(), d.end(), inf), fill(cur.begin(),
   ↪ cur.end(), 0);
21         d[s] = 0;
22         vector<int> q{s}, nq;
23         for (int step = 1; q.size(); swap(q, nq), nq.clear(),
   ↪ step++) {
24           for (auto& node : q) {
25             for (auto& edge : g[node]) {
26               int ne = e[edge].to;
27               if (!e[edge].remain || d[ne] <= step) continue;
28               d[ne] = step, nq.push_back(ne);
29               if (ne == t) return true;
30             }
31           }
32         }
33         return false;
34       };
35       function<int(int, int)> find = [&](int node, int limit) {
36         if (node == t || !limit) return limit;
37         int flow = 0;
38         for (int i = cur[node]; i < g[node].size(); i++) {
39           cur[node] = i;
40           int edge = g[node][i], oe = edge ^ 1, ne = e[edge].to;
41           if (!e[edge].remain || d[ne] != d[node] + 1) continue;
42           if (int temp = find(ne, min(limit - flow,
   ↪ e[edge].remain))) {
43             e[edge].remain -= temp, e[oe].remain += temp, flow
   ↪ += temp;
44           } else {
45             d[ne] = -1;
46           }
47           if (flow == limit) break;
48         }
49         return flow;
50       };
51       ll res = 0;
52       while (bfs())
53         while (int flow = find(s, inf)) res += flow;
54       return res;
55     }
56   };
```

  • USAGE

```cpp
int main() {
  int n, m, s, t;
  cin >> n >> m >> s >> t;
  Dinic dinic(n);
  for (int i = 0, u, v, c; i < m; i++) {
    cin >> u >> v >> c;
    dinic.add_edge(u - 1, v - 1, c);
  }
  cout << dinic.max_flow(s - 1, t - 1) << '\n';
}
```

PushRelabel Max-Flow (faster)

```cpp
//
↪  https://github.com/kth-competitive-programming/kactl/blob/main/content/graph/PushRelabel.h
#define rep(i, a, b) for (int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

struct PushRelabel {
  struct Edge {
    int dest, back;
    ll f, c;
  };
  vector<vector<Edge>> g;
  vector<ll> ec;
  vector<Edge*> cur;
  vector<vi> hs;
  vi H;
  PushRelabel(int n) : g(n), ec(n), cur(n), hs(2 * n), H(n) {}

  void addEdge(int s, int t, ll cap, ll rcap = 0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s]) - 1, 0, rcap});
  }

  void addFlow(Edge& e, ll f) {
    Edge& back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f;
    e.c -= f;
    ec[e.dest] += f;
    back.f -= f;
    back.c += f;
    ec[back.dest] -= f;
  }
  ll calc(int s, int t) {
    int v = sz(g);
    H[s] = v;
    ec[t] = 1;
    vi co(2 * v);
    co[0] = v - 1;
    rep(i, 0, v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

    for (int hi = 0;;) {
      while (hs[hi].empty())
        if (!hi--) return -ec[s];
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ec[u] > 0)  // discharge u
        if (cur[u] == g[u].data() + sz(g[u])) {
          H[u] = 1e9;
          for (Edge& e : g[u])
            if (e.c && H[u] > H[e.dest] + 1) H[u] = H[e.dest]
↪  + 1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            rep(i, 0, v) if (hi < H[i] && H[i] < v)--
↪  co[H[i]], H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest] + 1)
          addFlow(*cur[u], min(ec[u], cur[u]->c));
```

```cpp
      else
        ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H[a] >= sz(g); }
};
```

Min-Cost Max-Flow

```cpp
class MCMF {
public:
  static constexpr int INF = 1e9;
  const int n;
  vector<tuple<int, int, int>> e;
  vector<vector<int>> g;
  vector<int> h, dis, pre;
  bool dijkstra(int s, int t) {
    dis.assign(n, INF);
    pre.assign(n, -1);
    priority_queue<pair<int, int>, vector<pair<int, int>>,
↪  greater<>> que;
    dis[s] = 0;
    que.emplace(0, s);
    while (!que.empty()) {
      auto [d, u] = que.top();
      que.pop();
      if (dis[u] != d) continue;
      for (int i : g[u]) {
        auto [v, f, c] = e[i];
        if (c > 0 && dis[v] > d + h[u] - h[v] + f) {
          dis[v] = d + h[u] - h[v] + f;
          pre[v] = i;
          que.emplace(dis[v], v);
        }
      }
    }
    return dis[t] != INF;
  }
  MCMF(int n) : n(n), g(n) {}
  void add_edge(int u, int v, int fee, int c) {
    g[u].push_back(e.size());
    e.emplace_back(v, fee, c);
    g[v].push_back(e.size());
    e.emplace_back(u, -fee, 0);
  }
  pair<ll, ll> max_flow(const int s, const int t) {
    int flow = 0, cost = 0;
    h.assign(n, 0);
    while (dijkstra(s, t)) {
      for (int i = 0; i < n; ++i) h[i] += dis[i];
      for (int i = t; i != s; i = get<0>(e[pre[i] ^ 1])) {
        --get<2>(e[pre[i]]);
        ++get<2>(e[pre[i] ^ 1]);
      }
      ++flow;
      cost += h[t];
    }
    return {flow, cost};
  }
};
```

Max Cost Feasible Flow

```cpp
struct Edge {
  int from, to, cap, remain, cost;
};

struct MCMF {
  int n;
  vector<Edge> e;
  vector<vector<int>> g;
  vector<ll> d, pre;
  MCMF(int _n) : n(_n), g(n), d(n), pre(n) {}
  void add_edge(int u, int v, int c, int w) {
    g[u].push_back((int)e.size());
    e.push_back({u, v, c, c, w});
```

```
14        g[v].push_back((int)e.size());
15        e.push_back({v, u, 0, 0, -w});
16      }
17      pair<ll, ll> max_flow(int s, int t) {
18        ll inf = 1e18;
19        auto spfa = [&]() {
20          fill(d.begin(), d.end(), -inf); // important!
21          vector<int> f(n), seen(n);
22          d[s] = 0, f[s] = 1e9;
23          vector<int> q{s}, nq;
24          for (; q.size(); swap(q, nq), nq.clear()) {
25            for (auto& node : q) {
26              seen[node] = false;
27              for (auto& edge : g[node]) {
28                int ne = e[edge].to, cost = e[edge].cost;
29                if (!e[edge].remain || d[ne] >= d[node] + cost)
↪  continue;
30                d[ne] = d[node] + cost, pre[ne] = edge;
31                f[ne] = min(e[edge].remain, f[node]);
32                if (!seen[ne]) seen[ne] = true, nq.push_back(ne);
33              }
34            }
35          }
36          return f[t];
37        };
38        ll flow = 0, cost = 0;
39        while (int temp = spfa()) {
40          if (d[t] < 0) break; // important!
41          flow += temp, cost += temp * d[t];
42          for (ll i = t; i != s; i = e[pre[i]].from) {
43            e[pre[i]].remain -= temp, e[pre[i] ^ 1].remain +=
↪  temp;
44          }
45        }
46        return {flow, cost};
47      }
48    };
```

Heavy-Light Decomposition

```
1    struct HeavyLight {
2      int root = 0, n = 0;
3      std::vector<int> parent, deep, hson, top, sz, dfn;
4      HeavyLight(std::vector<std::vector<int>> &g, int _root)
5          : root(_root), n(int(g.size())), parent(n), deep(n),
↪  hson(n, -1), top(n), sz(n), dfn(n, -1) {
6        int cur = 0;
7        std::function<int(int, int, int)> dfs = [&](int node, int
↪  fa, int dep) {
8          deep[node] = dep, sz[node] = 1, parent[node] = fa;
9          for (auto &ne : g[node]) {
10            if (ne == fa) continue;
11            sz[node] += dfs(ne, node, dep + 1);
12            if (hson[node] == -1 || sz[ne] > sz[hson[node]])
↪  hson[node] = ne;
13          }
14          return sz[node];
15        };
16        std::function<void(int, int)> dfs2 = [&](int node, int t)
↪  {
17          top[node] = t, dfn[node] = cur++;
18          if (hson[node] == -1) return;
19          dfs2(hson[node], t);
20          for (auto &ne : g[node]) {
21            if (ne == parent[node] || ne == hson[node]) continue;
22            dfs2(ne, ne);
23          }
24        };
25        dfs(root, -1, 0), dfs2(root, root);
26      }
27
28      int lca(int x, int y) const {
29        while (top[x] != top[y]) {
30          if (deep[top[x]] < deep[top[y]]) swap(x, y);
31          x = parent[top[x]];
32        }
33        return deep[x] < deep[y] ? x : y;
```

```
34      }
35
36      std::vector<std::array<int, 2>> get_dfn_path(int x, int y)
↪  const {
37        std::array<std::vector<std::array<int, 2>>, 2> path;
38        bool front = true;
39        while (top[x] != top[y]) {
40          if (deep[top[x]] > deep[top[y]]) swap(x, y), front =
↪  !front;
41          path[front].push_back({dfn[top[y]], dfn[y] + 1});
42          y = parent[top[y]];
43        }
44        if (deep[x] > deep[y]) swap(x, y), front = !front;
45
46        path[front].push_back({dfn[x], dfn[y] + 1});
47        std::reverse(path[1].begin(), path[1].end());
48        for (const auto &[left, right] : path[1])
↪  path[0].push_back({right, left});
49        return path[0];
50      }
51
52      Node query_seg(int u, int v, const SegTree &seg) const {
53        auto node = Node();
54        for (const auto &[left, right] : get_dfn_path(u, v)) {
55          if (left > right) {
56            node = pull(node, rev(seg.query(right, left)));
57          } else {
58            node = pull(node, seg.query(left, right));
59          }
60        }
61        return node;
62      }
63    };
```

- USAGE:

```
1    vector<ll> light(n);
2    SegTree heavy(n), form_parent(n);
3    // cin >> x >> y, x--, y--;
4    int z = lca(x, y);
5    while (x != z) {
6      if (dfn[top[x]] <= dfn[top[z]]) {
7        // [dfn[z], dfn[x]), from heavy
8        heavy.modify(dfn[z], dfn[x], 1);
9        break;
10      }
11      // x -> top[x];
12      heavy.modify(dfn[top[x]], dfn[x], 1);
13      light[parent[top[x]]] += a[top[x]];
14      x = parent[top[x]];
15    }
16    while (y != z) {
17      if (dfn[top[y]] <= dfn[top[z]]) {
18        // (dfn[z], dfn[y]], from heavy
19        form_parent.modify(dfn[z] + 1, dfn[y] + 1, 1);
20        break;
21      }
22      // y -> top[y];
23      form_parent.modify(dfn[top[y]], dfn[y] + 1, 1);
24      y = parent[top[y]];
25    }
```

General Unweight Graph Matching

- Complexity: $O(n^3)$ (?)

```
1    struct BlossomMatch {
2      int n;
3      vector<vector<int>> e;
4      BlossomMatch(int _n) : n(_n), e(_n) {}
5      void add_edge(int u, int v) { e[u].push_back(v),
↪  e[v].push_back(u); }
6      vector<int> find_matching() {
7        vector<int> match(n, -1), vis(n), link(n), f(n), dep(n);
8        function<int(int)> find = [&](int x) { return f[x] == x ?
↪  x : (f[x] = find(f[x])); };
9        auto lca = [&](int u, int v) {
```

```
10        u = find(u), v = find(v);
11        while (u != v) {
12          if (dep[u] < dep[v]) swap(u, v);
13          u = find(link[match[u]]);
14        }
15        return u;
16      };
17      queue<int> que;
18      auto blossom = [&](int u, int v, int p) {
19        while (find(u) != p) {
20          link[u] = v, v = match[u];
21          if (vis[v] == 0) vis[v] = 1, que.push(v);
22          f[u] = f[v] = p, u = link[v];
23        }
24      };
25      // find an augmenting path starting from u and augment (if
↪    exist)
26      auto augment = [&](int node) {
27        while (!que.empty()) que.pop();
28        iota(f.begin(), f.end(), 0);
29        // vis = 0 corresponds to inner vertices, vis = 1
↪    corresponds to outer vertices
30        fill(vis.begin(), vis.end(), -1);
31        que.push(node);
32        vis[node] = 1, dep[node] = 0;
33        while (!que.empty()) {
34          int u = que.front();
35          que.pop();
36          for (auto v : e[u]) {
37            if (vis[v] == -1) {
38              vis[v] = 0, link[v] = u, dep[v] = dep[u] + 1;
39              // found an augmenting path
40              if (match[v] == -1) {
41                for (int x = v, y = u, temp; y != -1; x = temp,
↪    y = x == -1 ? -1 : link[x]) {
42                  temp = match[y], match[x] = y, match[y] = x;
43                }
44                return;
45              }
46              vis[match[v]] = 1, dep[match[v]] = dep[u] + 2;
47              que.push(match[v]);
48            } else if (vis[v] == 1 && find(v) != find(u)) {
49              // found a blossom
50              int p = lca(u, v);
51              blossom(u, v, p), blossom(v, u, p);
52            }
53          }
54        }
55      };
56      // find a maximal matching greedily (decrease constant)
57      auto greedy = [&]() {
58        for (int u = 0; u < n; ++u) {
59          if (match[u] != -1) continue;
60          for (auto v : e[u]) {
61            if (match[v] == -1) {
62              match[u] = v, match[v] = u;
63              break;
64            }
65          }
66        }
67      };
68      greedy();
69      for (int u = 0; u < n; ++u)
70        if (match[u] == -1) augment(u);
71      return match;
72    }
73  };
```

Maximum Bipartite Matching

- Needs dinic, complexity $\approx O(n + m\sqrt{n})$

```
1  struct BipartiteMatch {
2    int l, r;
3    Dinic dinic = Dinic(0);
4    BipartiteMatch(int _l, int _r) : l(_l), r(_r) {
5      dinic = Dinic(l + r + 2);
6      for (int i = 1; i <= l; i++) dinic.add_edge(0, i, 1);
7      for (int i = 1; i <= r; i++) dinic.add_edge(l + i, l + r +
↪    1, 1);
8    }
9    void add_edge(int u, int v) { dinic.add_edge(u + 1, l + v +
↪    1, 1); }
10   ll max_matching() { return dinic.max_flow(0, l + r + 1); }
11 };
```

2-SAT and Strongly Connected Components

```
1  void scc(vector<vector<int>>& g, int* idx) {
2    int n = g.size(), ct = 0;
3    int out[n];
4    vector<int> ginv[n];
5    memset(out, -1, sizeof out);
6    memset(idx, -1, n * sizeof(int));
7    function<void(int)> dfs = [&](int cur) {
8      out[cur] = INT_MAX;
9      for(int v : g[cur]) {
10       ginv[v].push_back(cur);
11       if(out[v] == -1) dfs(v);
12     }
13     ct++; out[cur] = ct;
14   };
15   vector<int> order;
16   for(int i = 0; i < n; i++) {
17     order.push_back(i);
18     if(out[i] == -1) dfs(i);
19   }
20   sort(order.begin(), order.end(), [&](int& u, int& v) {
21     return out[u] > out[v];
22   });
23   ct = 0;
24   stack<int> s;
25   auto dfs2 = [&](int start) {
26     s.push(start);
27     while(!s.empty()) {
28       int cur = s.top();
29       s.pop();
30       idx[cur] = ct;
31       for(int v : ginv[cur])
32         if(idx[v] == -1) s.push(v);
33     }
34   };
35   for(int v : order) {
36     if(idx[v] == -1) {
37       dfs2(v);
38       ct++;
39     }
40   }
41 }
42
43 // 0 => impossible, 1 => possible
44 pair<int,vector<int>> sat2(int n, vector<pair<int,int>>&
↪    clauses) {
45   vector<int> ans(n);
46   vector<vector<int>> g(2*n + 1);
47   for(auto [x, y] : clauses) {
48     x = x < 0 ? -x + n : x;
49     y = y < 0 ? -y + n : y;
50     int nx = x <= n ? x + n : x - n;
51     int ny = y <= n ? y + n : y - n;
52     g[nx].push_back(y);
53     g[ny].push_back(x);
54   }
55   int idx[2*n + 1];
56   scc(g, idx);
57   for(int i = 1; i <= n; i++) {
58     if(idx[i] == idx[i + n]) return {0, {}};
59     ans[i - 1] = idx[i + n] < idx[i];
60   }
61   return {1, ans};
62 }
```

Enumerating Triangles

- Complexity: $O(n + m\sqrt{m})$

```cpp
void enumerate_triangles(vector<pair<int,int>>& edges,
↪   function<void(int,int,int)> f) {
  int n = 0;
  for(auto [u, v] : edges) n = max({n, u + 1, v + 1});
  vector<int> deg(n);
  vector<int> g[n];
  for(auto [u, v] : edges) {
    deg[u]++;
    deg[v]++;
  }
  for(auto [u, v] : edges) {
    if(u == v) continue;
    if(deg[u] > deg[v] || (deg[u] == deg[v] && u > v))
      swap(u, v);
    g[u].push_back(v);
  }
  vector<int> flag(n);
  for(int i = 0; i < n; i++) {
    for(int v : g[i]) flag[v] = 1;
    for(int v : g[i]) for(int u : g[v]) {
      if(flag[u]) f(i, v, u);
    }
    for(int v : g[i]) flag[v] = 0;
  }
}
```

Tarjan

- shrink all circles into points (2-edge-connected-component)

```cpp
int cnt = 0, now = 0;
vector<ll> dfn(n, -1), low(n), belong(n, -1), stk;
function<void(ll, ll)> tarjan = [&](ll node, ll fa) {
  dfn[node] = low[node] = now++, stk.push_back(node);
  for (auto& ne : g[node]) {
    if (ne == fa) continue;
    if (dfn[ne] == -1) {
      tarjan(ne, node);
      low[node] = min(low[node], low[ne]);
    } else if (belong[ne] == -1) {
      low[node] = min(low[node], dfn[ne]);
    }
  }
  if (dfn[node] == low[node]) {
    while (true) {
      auto v = stk.back();
      belong[v] = cnt;
      stk.pop_back();
      if (v == node) break;
    }
    ++cnt;
  }
};
```

- 2-vertex-connected-component / Block forest

```cpp
int cnt = 0, now = 0;
vector<vector<ll>> e1(n);
vector<ll> dfn(n, -1), low(n), stk;
function<void(ll)> tarjan = [&](ll node) {
  dfn[node] = low[node] = now++, stk.push_back(node);
  for (auto& ne : g[node]) {
    if (dfn[ne] == -1) {
      tarjan(ne);
      low[node] = min(low[node], low[ne]);
      if (low[ne] == dfn[node]) {
        e1.push_back({});
        while (true) {
          auto x = stk.back();
          stk.pop_back();
          e1[n + cnt].push_back(x);
          // e1[x].push_back(n + cnt); // undirected
          if (x == ne) break;
        }
        e1[node].push_back(n + cnt);
        // e1[n + cnt].push_back(node); // undirected
        cnt++;
      }
    } else {
      low[node] = min(low[node], dfn[ne]);
    }
  }
};
```

Kruskal reconstruct tree

```cpp
int _n, m;
cin >> _n >> m; // _n: # of node, m: # of edge
int n = 2 * _n - 1; // root: n-1
vector<array<int, 3>> edges(m);
for (auto& [w, u, v] : edges) {
  cin >> u >> v >> w, u--, v--;
}
sort(edges.begin(), edges.end());
vector<int> p(n);
iota(p.begin(), p.end(), 0);
function<int(int)> find = [&](int x) { return p[x] == x ? x :
↪   (p[x] = find(p[x])); };
auto merge = [&](int x, int y) { p[find(x)] = find(y); };
vector<vector<int>> g(n);
vector<int> val(m);
val.reserve(n);
for (auto [w, u, v] : edges) {
  u = find(u), v = find(v);
  if (u == v) continue;
  val.push_back(w);
  int node = (int)val.size() - 1;
  g[node].push_back(u), g[node].push_back(v);
  merge(u, node), merge(v, node);
}
```

Math

Inverse

```cpp
ll inv(ll a, ll m) { return a == 1 ? 1 : ((m - m / a) * inv(m
↪   % a, m)) % m); }
// or
power(a, MOD - 2)
```

- USAGE: get factorial

```cpp
vector<Z> f(MAX_N, 1), rf(MAX_N, 1);
for (int i = 2; i < MAX_N; i++) f[i] = f[i - 1] * i % MOD;
rf[MAX_N - 1] = power(f[MAX_N - 1], MOD - 2);
for (int i = MAX_N - 2; i > 1; i--) rf[i] = rf[i + 1] * (i +
↪   1) % MOD;
auto binom = [&](ll n, ll r) -> Z {
  if (n < 0 || r < 0 || n < r) return 0;
  return f[n] * rf[n - r] * rf[r];
};
```

Mod Class

```cpp
constexpr ll norm(ll x) { return (x % MOD + MOD) % MOD; }
template <typename T>
constexpr T power(T a, ll b, T res = 1) {
  for (; b; b /= 2, (a *= a) %= MOD)
    if (b & 1) (res *= a) %= MOD;
  return res;
}
struct Z {
  ll x;
  constexpr Z(ll _x = 0) : x(norm(_x)) {}
  // auto operator<=>(const Z &) const = default; // cpp20
↪   only
  Z operator-() const { return Z(norm(MOD - x)); }
```

```
13    Z inv() const { return power(*this, MOD - 2); }
14    Z &operator*=(const Z &rhs) { return x = x * rhs.x % MOD,
   ↪  *this; }
15    Z &operator+=(const Z &rhs) { return x = norm(x + rhs.x),
   ↪  *this; }
16    Z &operator-=(const Z &rhs) { return x = norm(x - rhs.x),
   ↪  *this; }
17    Z &operator/=(const Z &rhs) { return *this *= rhs.inv(); }
18    Z &operator%=(const ll &rhs) { return x %= rhs, *this; }
19    friend Z operator*(Z lhs, const Z &rhs) { return lhs *= rhs;
   ↪  }
20    friend Z operator+(Z lhs, const Z &rhs) { return lhs += rhs;
   ↪  }
21    friend Z operator-(Z lhs, const Z &rhs) { return lhs -= rhs;
   ↪  }
22    friend Z operator/(Z lhs, const Z &rhs) { return lhs /= rhs;
   ↪  }
23    friend Z operator%(Z lhs, const ll &rhs) { return lhs %=
   ↪  rhs; }
24    friend auto &operator>>(istream &i, Z &z) { return i >> z.x;
   ↪  }
25    friend auto &operator<<(ostream &o, const Z &z) { return o
   ↪  << z.x; }
26  };
```

- large mod (for NTT to do FFT in ll range without modulo)

```
1  constexpr i128 MOD = 9223372036737335297;
```

- fastest mod class! be careful with overflow, only use when the time limit is tight

```
1  constexpr int norm(int x) {
2    if (x < 0) x += MOD;
3    if (x >= MOD) x -= MOD;
4    return x;
5  }
```

Combinatorics

```
1   const int NMAX = 3000010;
2   ll factorialcompute[NMAX];
3   ll invfactorialcompute[NMAX];
4   ll binpow(ll a, ll pow, ll mod) {
5     if (pow <= 0)
6       return 1;
7     ll p = binpow(a, pow / 2, mod) % mod;
8     p = (p * p) % mod;
9
10    return (pow % 2 == 0) ? p : (a * p) % mod;
11  }
12  ll inverse(ll a, ll mod) {
13    if (a == 1) return 1;
14    return binpow(a, mod-2, mod);
15  }
16  ll combination(int a, int b, ll mod) {
17    if ( a < b ) return 0;
18    ll cur = factorialcompute[a];
19    cur *= invfactorialcompute[b];
20    cur %= mod;
21    cur *= invfactorialcompute[a - b];
22    cur %= mod;
23    return cur;
24  }
25  void precomputeFactorial() {
26    factorialcompute[0] = 1;
27    invfactorialcompute[0] = 1;
28    for(int i = 1; i < NMAX; i++) {
29      factorialcompute[i] = factorialcompute[i-1] * i;
30      factorialcompute[i] %= MOD;
31    }
32    invfactorialcompute[NMAX-1] =
   ↪  inverse(factorialcompute[NMAX-1], MOD);
33    for(int i = NMAX-2; i > -1; i--) {
34      invfactorialcompute[i] = invfactorialcompute[i+1] *
   ↪  (i+1);
```

```
35        invfactorialcompute[i] %= MOD;
36    }
37  }
```

exgcd

```
1  array<ll, 3> exgcd(ll a, ll b) {
2    if(!b) return {a, 1, 0};
3
4    auto [g, x, y] = exgcd(b, a%b);
5    return {g, y, x - a/b*y};
6  }
```

Factor/primes

```
1   vector<int> primes(0);
2   void gen_primes(int a) {
3     vector<bool> is_prime(a+1, true);
4     is_prime[0] = is_prime[1] = false;
5     for(int i = 2; i * i <= a; i++) {
6       if(is_prime[i]) {
7         for(int j = i * i; j <= a; j += i) is_prime[j] =
   ↪  false;
8       }
9     }
10    for(int i = 0; i <= a; i++) {
11      if(is_prime[i]) primes.push_back(i);
12    }
13  }
14  vector<ll> gen_factors_prime(ll a){
15    vector<ll> factors;
16    factors.push_back(1);
17    if(a == 1) return factors;
18    for(int z : primes) {
19      if(z * z > a) {
20        z = a;
21      }
22      int cnt = 0;
23      while(a % z == 0) {
24        cnt++;
25        a /= z;
26      }
27      ll num = z;
28      int size = factors.size();
29      for(int i = 1; i <= cnt; i++) {
30        for(int j = 0; j < size; j++) {
31          factors.push_back(num * factors[j]);
32        }
33        num *= z;
34      }
35      if (a == 1) break;
36    }
37    return factors;
38  }
39  vector<int> get_primes(int num) {
40    vector<int> curPrime;
41    if(num == 1) return curPrime;
42    for(int z : primes) {
43      if(z * z > num) {
44        curPrime.push_back(num);
45        break;
46      }
47      if(num % z == 0) {
48        curPrime.push_back(z);
49        while(num % z == 0) num /= z;
50      }
51      if(num == 1) break;
52    }
53    return curPrime;
54  }
```

Cancer mod class

- Explanation: for some prime modulo p, maintains numbers of form p^x * y, where y is a nonzero remainder

mod p

- Be careful with calling Cancer(x, y), it doesn't fix the input if y > p

```cpp
struct Cancer {
  ll x; ll y;
  Cancer() : Cancer(0, 1) {}
  Cancer(ll _y) {
    x = 0, y = _y;
    while(y % MOD == 0) {
      y /= MOD;
      x++;
    }
  }
  Cancer(ll _x, ll _y) : x(_x), y(_y) {}
  Cancer inv() { return Cancer(-x, power(y, MOD - 2)); }
  Cancer operator*(const Cancer &c) { return Cancer(x + c.x,
  ↪ (y * c.y) % MOD); }
  Cancer operator*(ll m) {
    ll p = 0;
    while(m % MOD == 0) {
      m /= MOD;
      p++;
    }
    return Cancer(x + p, (m * y) % MOD);
  }
  friend auto &operator<<(ostream &o, Cancer c) { return o <<
  ↪ c.x << ' ' << c.y; }
};
```

NTT, FFT, FWT

- ntt

```cpp
void ntt(vector<Z>& a, int f) {
  int n = int(a.size());
  vector<Z> w(n);
  vector<int> rev(n);
  for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] / 2) | ((i
  ↪ & 1) * (n / 2));
  for (int i = 0; i < n; i++) {
    if (i < rev[i]) swap(a[i], a[rev[i]]);
  }
  Z wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
  w[0] = 1;
  for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn;
  for (int mid = 1; mid < n; mid *= 2) {
    for (int i = 0; i < n; i += 2 * mid) {
      for (int j = 0; j < mid; j++) {
        Z x = a[i + j], y = a[i + j + mid] * w[n / (2 * mid) *
  ↪ j];
        a[i + j] = x + y, a[i + j + mid] = x - y;
      }
    }
  }
  if (f) {
    Z iv = power(Z(n), MOD - 2);
    for (auto& x : a) x *= iv;
  }
}
```

- USAGE: Polynomial multiplication

```cpp
vector<Z> mul(vector<Z> a, vector<Z> b) {
  int n = 1, m = (int)a.size() + (int)b.size() - 1;
  while (n < m) n *= 2;
  a.resize(n), b.resize(n);
  ntt(a, 0), ntt(b, 0);
  for (int i = 0; i < n; i++) a[i] *= b[i];
  ntt(a, 1);
  a.resize(m);
  return a;
}
```

- FFT (should prefer NTT, only use this when input is not integer)

```cpp
const double PI = acos(-1);
auto mul = [&](const vector<double>& aa, const vector<double>&
↪ bb) {
  int n = (int)aa.size(), m = (int)bb.size(), bit = 1;
  while ((1 << bit) < n + m - 1) bit++;
  int len = 1 << bit;
  vector<complex<double>> a(len), b(len);
  vector<int> rev(len);
  for (int i = 0; i < n; i++) a[i].real(aa[i]);
  for (int i = 0; i < m; i++) b[i].real(bb[i]);
  for (int i = 0; i < len; i++) rev[i] = (rev[i >> 1] >> 1) |
↪ ((i & 1) << (bit - 1));
  auto fft = [&](vector<complex<double>>& p, int inv) {
    for (int i = 0; i < len; i++)
      if (i < rev[i]) swap(p[i], p[rev[i]]);
    for (int mid = 1; mid < len; mid *= 2) {
      auto w1 = complex<double>(cos(PI / mid), (inv ? -1 : 1)
↪ * sin(PI / mid));
      for (int i = 0; i < len; i += mid * 2) {
        auto wk = complex<double>(1, 0);
        for (int j = 0; j < mid; j++, wk = wk * w1) {
          auto x = p[i + j], y = wk * p[i + j + mid];
          p[i + j] = x + y, p[i + j + mid] = x - y;
        }
      }
    }
    if (inv == 1) {
      for (int i = 0; i < len; i++) p[i].real(p[i].real() /
↪ len);
    }
  };
  fft(a, 0), fft(b, 0);
  for (int i = 0; i < len; i++) a[i] = a[i] * b[i];
  fft(a, 1);
  a.resize(n + m - 1);
  vector<double> res(n + m - 1);
  for (int i = 0; i < n + m - 1; i++) res[i] = a[i].real();
  return res;
};
```

Polynomial Class

```cpp
using ll = long long;
constexpr ll MOD = 998244353;

ll norm(ll x) { return (x % MOD + MOD) % MOD; }
template <class T>
T power(T a, ll b, T res = 1) {
  for (; b; b /= 2, (a *= a) %= MOD)
    if (b & 1) (res *= a) %= MOD;
  return res;
}

struct Z {
  ll x;
  Z(ll _x = 0) : x(norm(_x)) {}
  // auto operator<=>(const Z &) const = default;
  Z operator-() const { return Z(norm(MOD - x)); }
  Z inv() const { return power(*this, MOD - 2); }
  Z &operator*=(const Z &rhs) { return x = x * rhs.x % MOD,
  ↪ *this; }
  Z &operator+=(const Z &rhs) { return x = norm(x + rhs.x),
  ↪ *this; }
  Z &operator-=(const Z &rhs) { return x = norm(x - rhs.x),
  ↪ *this; }
  Z &operator/=(const Z &rhs) { return *this *= rhs.inv(); }
  Z &operator%=(const ll &rhs) { return x %= rhs, *this; }
  friend Z operator*(Z lhs, const Z &rhs) { return lhs *= rhs;
  ↪ }
  friend Z operator+(Z lhs, const Z &rhs) { return lhs += rhs;
  ↪ }
  friend Z operator-(Z lhs, const Z &rhs) { return lhs -= rhs;
  ↪ }
  friend Z operator/(Z lhs, const Z &rhs) { return lhs /= rhs;
  ↪ }
  friend Z operator%(Z lhs, const ll &rhs) { return lhs %=
  ↪ rhs; }
```

```
28    friend auto &operator>>(istream &i, Z &z) { return i >> z.x;
   ↪  }
29    friend auto &operator<<(ostream &o, const Z &z) { return o
   ↪  << z.x; }
30  };

32  void ntt(vector<Z> &a, int f) {
33    int n = (int)a.size();
34    vector<Z> w(n);
35    vector<int> rev(n);
36    for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] / 2) | ((i
   ↪  & 1) * (n / 2));
37    for (int i = 0; i < n; i++)
38      if (i < rev[i]) swap(a[i], a[rev[i]]);
39    Z wn = power(ll(f ? (MOD + 1) / 3 : 3), (MOD - 1) / n);
40    w[0] = 1;
41    for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn;
42    for (int mid = 1; mid < n; mid *= 2) {
43      for (int i = 0; i < n; i += 2 * mid) {
44        for (int j = 0; j < mid; j++) {
45          Z x = a[i + j], y = a[i + j + mid] * w[n / (2 * mid) *
   ↪  j];
46          a[i + j] = x + y, a[i + j + mid] = x - y;
47        }
48      }
49    }
50    if (f) {
51      Z iv = power(Z(n), MOD - 2);
52      for (int i = 0; i < n; i++) a[i] *= iv;
53    }
54  }

56  struct Poly {
57    vector<Z> a;
58    Poly() {}
59    Poly(const vector<Z> &_a) : a(_a) {}
60    int size() const { return (int)a.size(); }
61    void resize(int n) { a.resize(n); }
62    Z operator[](int idx) const {
63      if (idx < 0 || idx >= size()) return 0;
64      return a[idx];
65    }
66    Z &operator[](int idx) { return a[idx]; }
67    Poly mulxk(int k) const {
68      auto b = a;
69      b.insert(b.begin(), k, 0);
70      return Poly(b);
71    }
72    Poly modxk(int k) const { return Poly(vector<Z>(a.begin(),
   ↪  a.begin() + min(k, size()))); }
73    Poly divxk(int k) const {
74      if (size() <= k) return Poly();
75      return Poly(vector<Z>(a.begin() + k, a.end()));
76    }
77    friend Poly operator+(const Poly &a, const Poly &b) {
78      vector<Z> res(max(a.size(), b.size()));
79      for (int i = 0; i < (int)res.size(); i++) res[i] = a[i] +
   ↪  b[i];
80      return Poly(res);
81    }
82    friend Poly operator-(const Poly &a, const Poly &b) {
83      vector<Z> res(max(a.size(), b.size()));
84      for (int i = 0; i < (int)res.size(); i++) res[i] = a[i] -
   ↪  b[i];
85      return Poly(res);
86    }
87    friend Poly operator*(Poly a, Poly b) {
88      if (a.size() == 0 || b.size() == 0) return Poly();
89      int n = 1, m = (int)a.size() + (int)b.size() - 1;
90      while (n < m) n *= 2;
91      a.resize(n), b.resize(n);
92      ntt(a.a, 0), ntt(b.a, 0);
93      for (int i = 0; i < n; i++) a[i] *= b[i];
94      ntt(a.a, 1);
95      a.resize(m);
96      return a;
97    }
98    friend Poly operator*(Z a, Poly b) {
99      for (int i = 0; i < (int)b.size(); i++) b[i] *= a;
100     return b;
101   }
102   friend Poly operator*(Poly a, Z b) {
103     for (int i = 0; i < (int)a.size(); i++) a[i] *= b;
104     return a;
105   }
106   Poly &operator+=(Poly b) { return (*this) = (*this) + b; }
107   Poly &operator-=(Poly b) { return (*this) = (*this) - b; }
108   Poly &operator*=(Poly b) { return (*this) = (*this) * b; }
109   Poly deriv() const {
110     if (a.empty()) return Poly();
111     vector<Z> res(size() - 1);
112     for (int i = 0; i < size() - 1; ++i) res[i] = (i + 1) *
   ↪  a[i + 1];
113     return Poly(res);
114   }
115   Poly integr() const {
116     vector<Z> res(size() + 1);
117     for (int i = 0; i < size(); ++i) res[i + 1] = a[i] / (i +
   ↪  1);
118     return Poly(res);
119   }
120   Poly inv(int m) const {
121     Poly x({a[0].inv()});
122     int k = 1;
123     while (k < m) {
124       k *= 2;
125       x = (x * (Poly({2}) - modxk(k) * x)).modxk(k);
126     }
127     return x.modxk(m);
128   }
129   Poly log(int m) const { return (deriv() *
   ↪  inv(m)).integr().modxk(m); }
130   Poly exp(int m) const {
131     Poly x({1});
132     int k = 1;
133     while (k < m) {
134       k *= 2;
135       x = (x * (Poly({1}) - x.log(k) + modxk(k))).modxk(k);
136     }
137     return x.modxk(m);
138   }
139   Poly pow(int k, int m) const {
140     int i = 0;
141     while (i < size() && a[i].x == 0) i++;
142     if (i == size() || 1LL * i * k >= m) {
143       return Poly(vector<Z>(m));
144     }
145     Z v = a[i];
146     auto f = divxk(i) * v.inv();
147     return (f.log(m - i * k) * k).exp(m - i * k).mulxk(i * k)
   ↪  * power(v, k);
148   }
149   Poly sqrt(int m) const {
150     Poly x({1});
151     int k = 1;
152     while (k < m) {
153       k *= 2;
154       x = (x + (modxk(k) * x.inv(k)).modxk(k)) * ((MOD + 1) /
   ↪  2);
155     }
156     return x.modxk(m);
157   }
158   Poly mulT(Poly b) const {
159     if (b.size() == 0) return Poly();
160     int n = b.size();
161     reverse(b.a.begin(), b.a.end());
162     return ((*this) * b).divxk(n - 1);
163   }
164   Poly divmod(Poly b) const {
165     auto n = size(), m = b.size();
166     auto t = *this;
167     reverse(t.a.begin(), t.a.end());
168     reverse(b.a.begin(), b.a.end());
169     Poly res = (t * b.inv(n)).modxk(n - m + 1);
```

```
170      reverse(res.a.begin(), res.a.end());
171      return res;
172    }
173    vector<Z> eval(vector<Z> x) const {
174      if (size() == 0) return vector<Z>(x.size(), 0);
175      const int n = max(int(x.size()), size());
176      vector<Poly> q(4 * n);
177      vector<Z> ans(x.size());
178      x.resize(n);
179      function<void(int, int, int)> build = [&](int p, int l,
    ↪  int r) {
180        if (r - l == 1) {
181          q[p] = Poly({1, -x[l]});
182        } else {
183          int m = (l + r) / 2;
184          build(2 * p, l, m), build(2 * p + 1, m, r);
185          q[p] = q[2 * p] * q[2 * p + 1];
186        }
187      };
188      build(1, 0, n);
189      auto work = [&](auto self, int p, int l, int r, const Poly
    ↪  &num) -> void {
190        if (r - l == 1) {
191          if (l < int(ans.size())) ans[l] = num[0];
192        } else {
193          int m = (l + r) / 2;
194          self(self, 2 * p, l, m, num.mulT(q[2 * p + 1]).modxk(m
    ↪  - l));
195          self(self, 2 * p + 1, m, r, num.mulT(q[2 * p]).modxk(r
    ↪  - m));
196        }
197      };
198      work(work, 1, 0, n, mulT(q[1].inv(n)));
199      return ans;
200    }
201  };
```

Sieve

- linear sieve

```
1  vector<int> min_primes(MAX_N), primes;
2  primes.reserve(1e5);
3  for (int i = 2; i < MAX_N; i++) {
4    if (!min_primes[i]) min_primes[i] = i, primes.push_back(i);
5    for (auto& p : primes) {
6      if (p * i >= MAX_N) break;
7      min_primes[p * i] = p;
8      if (i % p == 0) break;
9    }
10  }
```

- mobius function

```
1  vector<int> min_p(MAX_N), mu(MAX_N), primes;
2  mu[1] = 1, primes.reserve(1e5);
3  for (int i = 2; I < MAX_N; i++) {
4    if (min_p[i] == 0) {
5      min_p[i] = i;
6      primes.push_back(i);
7      mu[i] = -1;
8    }
9    for (auto p : primes) {
10      if (i * p >= MAX_N) break;
11      min_p[i * p] = p;
12      if (i % p == 0) {
13        mu[i * p] = 0;
14        break;
15      }
16      mu[i * p] = -mu[i];
17    }
18  }
```

- Euler's totient function

```
1  vector<int> min_p(MAX_N), phi(MAX_N), primes;
2  phi[1] = 1, primes.reserve(1e5);
3  for (int i = 2; i < MAX_N; i++) {
```

```
4    if (min_p[i] == 0) {
5      min_p[i] = i;
6      primes.push_back(i);
7      phi[i] = i - 1;
8    }
9    for (auto p : primes) {
10      if (i * p >= MAX_N) break;
11      min_p[i * p] = p;
12      if (i % p == 0) {
13        phi[i * p] = phi[i] * p;
14        break;
15      }
16      phi[i * p] = phi[i] * phi[p];
17    }
18  }
```

Gaussian Elimination

```
1  bool is_0(Z v) { return v.x == 0; }
2  Z abs(Z v) { return v; }
3  bool is_0(double v) { return abs(v) < 1e-9; }
4
5  // 1 => unique solution, 0 => no solution, -1 => multiple
   ↪  solutions
6  template <typename T>
7  int gaussian_elimination(vector<vector<T>> &a, int limit) {
8    if (a.empty() || a[0].empty()) return -1;
9    int h = (int)a.size(), w = (int)a[0].size(), r = 0;
10    for (int c = 0; c < limit; c++) {
11      int id = -1;
12      for (int i = r; i < h; i++) {
13        if (!is_0(a[i][c]) && (id == -1 || abs(a[id][c]) <
    ↪  abs(a[i][c]))) {
14          id = i;
15        }
16      }
17      if (id == -1) continue;
18      if (id > r) {
19        swap(a[r], a[id]);
20        for (int j = c; j < w; j++) a[id][j] = -a[id][j];
21      }
22      vector<int> nonzero;
23      for (int j = c; j < w; j++) {
24        if (!is_0(a[r][j])) nonzero.push_back(j);
25      }
26      T inv_a = 1 / a[r][c];
27      for (int i = r + 1; i < h; i++) {
28        if (is_0(a[i][c])) continue;
29        T coeff = -a[i][c] * inv_a;
30        for (int j : nonzero) a[i][j] += coeff * a[r][j];
31      }
32      ++r;
33    }
34    for (int row = h - 1; row >= 0; row--) {
35      for (int c = 0; c < limit; c++) {
36        if (!is_0(a[row][c])) {
37          T inv_a = 1 / a[row][c];
38          for (int i = row - 1; i >= 0; i--) {
39            if (is_0(a[i][c])) continue;
40            T coeff = -a[i][c] * inv_a;
41            for (int j = c; j < w; j++) a[i][j] += coeff *
    ↪  a[row][j];
42          }
43          break;
44        }
45      }
46    } // not-free variables: only it on its line
47    for(int i = r; i < h; i++) if(!is_0(a[i][limit])) return 0;
48    return (r == limit) ? 1 : -1;
49  }
50
51  template <typename T>
52  pair<int,vector<T>> solve_linear(vector<vector<T>> a, const
   ↪  vector<T> &b, int w) {
53    int h = (int)a.size();
54    for (int i = 0; i < h; i++) a[i].push_back(b[i]);
55    int sol = gaussian_elimination(a, w);
```

```
56    if(!sol) return {0, vector<T>()};
57    vector<T> x(w, 0);
58    for (int i = 0; i < h; i++) {
59      for (int j = 0; j < w; j++) {
60        if (!is_0(a[i][j])) {
61          x[j] = a[i][w] / a[i][j];
62          break;
63        }
64      }
65    }
66    return {sol, x};
67 }
```

is_prime

- (Miller–Rabin primality test)

```
1  i128 power(i128 a, i128 b, i128 MOD = 1, i128 res = 1) {
2    for (; b; b /= 2, (a *= a) %= MOD)
3      if (b & 1) (res *= a) %= MOD;
4    return res;
5  }
6
7  bool is_prime(ll n) {
8    if (n < 2) return false;
9    static constexpr int A[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
10   int s = __builtin_ctzll(n - 1);
11   ll d = (n - 1) >> s;
12   for (auto a : A) {
13     if (a == n) return true;
14     ll x = (ll)power(a, d, n);
15     if (x == 1 || x == n - 1) continue;
16     bool ok = false;
17     for (int i = 0; i < s - 1; ++i) {
18       x = ll((i128)x * x % n);  // potential overflow!
19       if (x == n - 1) {
20         ok = true;
21         break;
22       }
23     }
24     if (!ok) return false;
25   }
26   return true;
27 }
```

```
1  ll pollard_rho(ll x) {
2    ll s = 0, t = 0, c = rng() % (x - 1) + 1;
3    ll stp = 0, goal = 1, val = 1;
4    for (goal = 1;; goal *= 2, s = t, val = 1) {
5      for (stp = 1; stp <= goal; ++stp) {
6        t = ll(((i128)t * t + c) % x);
7        val = ll((i128)val * abs(t - s) % x);
8        if ((stp % 127) == 0) {
9          ll d = gcd(val, x);
10         if (d > 1) return d;
11       }
12     }
13     ll d = gcd(val, x);
14     if (d > 1) return d;
15   }
16 }
17
18 ll get_max_factor(ll _x) {
19   ll max_factor = 0;
20   function<void(ll)> fac = [&](ll x) {
21     if (x <= max_factor || x < 2) return;
22     if (is_prime(x)) {
23       max_factor = max_factor > x ? max_factor : x;
24       return;
25     }
26     ll p = x;
27     while (p >= x) p = pollard_rho(x);
28     while ((x % p) == 0) x /= p;
29     fac(x), fac(p);
30   };
31   fac(_x);
32   return max_factor;
33 }
```

Radix Sort

```
1  struct identity {
2    template<typename T>
3    T operator()(const T &x) const {
4      return x;
5    }
6  };
7  // A stable sort that sorts in passes of `bits_per_pass` bits
   //   at a time.
8  template<typename T, typename T_extract_key = identity>
9  void radix_sort(vector<T> &data, int bits_per_pass = 10, const
   //   T_extract_key &extract_key = identity()) {
10   if (int64_t(data.size()) * (64 -
   //   __builtin_clzll(data.size())) < 2 * (1 << bits_per_pass))
   //   {
11     stable_sort(data.begin(), data.end(), [&](const T &a,
   //   const T &b) {
12       return extract_key(a) < extract_key(b);
13     });
14     return;
15   }
16
17   using T_key = decltype(extract_key(data.front()));
18   T_key minimum = numeric_limits<T_key>::max();
19   for (T &x : data)
20     minimum = min(minimum, extract_key(x));
21
22   int max_bits = 0;
23   for (T &x : data) {
24     T_key key = extract_key(x);
25     max_bits = max(max_bits, key == minimum ? 0 : 64 -
   //   __builtin_clzll(key - minimum));
26   }
27   int passes = max((max_bits + bits_per_pass / 2) /
   //   bits_per_pass, 1);
28   if (64 - __builtin_clzll(data.size()) <= 1.5 * passes) {
29     stable_sort(data.begin(), data.end(), [&](const T &a,
   //   const T &b) {
30       return extract_key(a) < extract_key(b);
31     });
32     return;
33   }
34   vector<T> buffer(data.size());
35   vector<int> counts;
36   int bits_so_far = 0;
37
38   for (int p = 0; p < passes; p++) {
39     int bits = (max_bits + p) / passes;
40     counts.assign(1 << bits, 0);
41     for (T &x : data) {
42       T_key key = T_key(extract_key(x) - minimum);
43       counts[(key >> bits_so_far) & ((1 << bits) -
   //   1)]++;
44     }
45     int count_sum = 0;
46     for (int &count : counts) {
47       int current = count;
48       count = count_sum;
49       count_sum += current;
50     }
51     for (T &x : data) {
52       T_key key = T_key(extract_key(x) - minimum);
53       int key_section = int((key >> bits_so_far) & ((1
   //   << bits) - 1));
54       buffer[counts[key_section]++] = x;
55     }
56     swap(data, buffer);
57     bits_so_far += bits;
58   }
59 }
```

- USAGE

```
radix_sort(edges, 10, [&](const edge &e) -> int { return
↪   abs(e.weight - x); });
```

lucas

```
1  ll lucas(ll n, ll m, ll p) {
2    if (m == 0) return 1;
3    return (binom(n % p, m % p, p) * lucas(n / p, m / p, p)) %
↪   p;
4  }
```

parity of n choose m

```
1  (n & m) == m <=> odd
```

sosdp

subset sum

```
1  auto f = a;
2  for (int i = 0; i < SZ; i++) {
3    for (int mask = 0; mask < (1 << SZ); mask++) {
4      if (mask & (1 << i)) f[mask] += f[mask ^ (1 << i)];
5    }
6  }
```

prf

```
1  ll _h(ll x) { return x * x * x * 1241483 + 19278349; }
2  ll prf(ll x) { return _h(x & ((1 << 31) - 1)) + _h(x >> 31); }
```

String

AC Automaton

```
1   struct AC_automaton {
2     int sz = 26;
3     vector<vector<int>> e = {vector<int>(sz)};  // vector is
↪    faster than unordered_map
4     vector<int> fail = {0}, end = {0};
5     vector<int> fast = {0};  // closest end
6
7     int insert(string& s) {
8       int p = 0;
9       for (auto c : s) {
10        c -= 'a';
11        if (!e[p][c]) {
12          e.emplace_back(sz);
13          fail.emplace_back();
14          end.emplace_back();
15          fast.emplace_back();
16          e[p][c] = (int)e.size() - 1;
17        }
18        p = e[p][c];
19      }
20      end[p] += 1;
21      return p;
22    }
23
24    void build() {
25      queue<int> q;
26      for (int i = 0; i < sz; i++)
27        if (e[0][i]) q.push(e[0][i]);
28      while (!q.empty()) {
29        int p = q.front();
30        q.pop();
31        fast[p] = end[p] ? p : fast[fail[p]];
32        for (int i = 0; i < sz; i++) {
33          if (e[p][i]) {
34            fail[e[p][i]] = e[fail[p]][i];
35            q.push(e[p][i]);
36          } else {
37            e[p][i] = e[fail[p]][i];
```

```
38          }
39        }
40      }
41    }
42  };
```

KMP

- nex[i]: length of longest common prefix & suffix for pat[0..i]

```
1   vector<int> get_next(vector<int> &pat) {
2     int m = (int)pat.size();
3     vector<int> nex(m);
4     for (int i = 1, j = 0; i < m; i++) {
5       while (j && pat[j] != pat[i]) j = nex[j - 1];
6       if (pat[j] == pat[i]) j++;
7       nex[i] = j;
8     }
9     return nex;
10  }
```

- kmp match for txt and pat

```
1   auto nex = get_next(pat);
2   for (int i = 0, j = 0; i < n; i++) {
3     while (j && pat[j] != txt[i]) j = nex[j - 1];
4     if (pat[j] == txt[i]) j++;
5     if (j == m) {
6       // do what you want with the match
7       // start index is `i - m + 1`
8       j = nex[j - 1];
9     }
10  }
```

Z function

- z[i]: length of longest common prefix of s and s[i:]

```
1   vector<int> z_function(string s) {
2     int n = (int)s.size();
3     vector<int> z(n);
4     for (int i = 1, l = 0, r = 0; i < n; ++i) {
5       if (i <= r) z[i] = min(r - i + 1, z[i - l]);
6       while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
7       if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
8     }
9     return z;
10  }
```

General Suffix Automaton

```
1   constexpr int SZ = 26;
2
3   struct GSAM {
4     vector<vector<int>> e = {vector<int>(SZ)};  // the labeled
↪    edges from node i
5     vector<int> parent = {-1};                  // the parent of
↪    i
6     vector<int> length = {0};                   // the length of
↪    the longest string
7
8     GSAM(int n) { e.reserve(2 * n), parent.reserve(2 * n),
↪    length.reserve(2 * n); };
9     int extend(int c, int p) {  // character, last
10      bool f = true;           // if already exist
11      int r = 0;               // potential new node
12      if (!e[p][c]) {          // only extend when not exist
13        f = false;
14        e.push_back(vector<int>(SZ));
15        parent.push_back(0);
16        length.push_back(length[p] + 1);
17        r = (int)e.size() - 1;
18        for (; ~p && !e[p][c]; p = parent[p]) e[p][c] = r;  //
↪    update parents
19      }
```

```
20      if (f || ~p) {
21        int q = e[p][c];
22        if (length[q] == length[p] + 1) {
23          if (f) return q;
24          parent[r] = q;
25        } else {
26          e.push_back(e[q]);
27          parent.push_back(parent[q]);
28          length.push_back(length[p] + 1);
29          int qq = parent[q] = (int)e.size() - 1;
30          for (; ~p && e[p][c] == q; p = parent[p]) e[p][c] =
   ↪   qq;
31          if (f) return qq;
32          parent[r] = qq;
33        }
34      }
35      return r;
36    }
37  };
```

- Topo sort on GSAM

```
1  ll sz = gsam.e.size();
2  vector<int> c(sz + 1);
3  vector<int> order(sz);
4  for (int i = 1; i < sz; i++) c[gsam.length[i]]++;
5  for (int i = 1; i < sz; i++) c[i] += c[i - 1];
6  for (int i = 1; i < sz; i++) order[c[gsam.length[i]]--] = i;
7  reverse(order.begin(), order.end()); // reverse so that large
   ↪   len to small
```

- can be used as an ordinary SAM
- USAGE (the number of distinct substring)

```
1  int main() {
2    int n, last = 0;
3    string s;
4    cin >> n;
5    auto a = GSAM();
6    for (int i = 0; i < n; i++) {
7      cin >> s;
8      last = 0;  // reset last
9      for (auto&& c : s) last = a.extend(c, last);
10   }
11   ll ans = 0;
12   for (int i = 1; i < a.e.size(); i++) {
13     ans += a.length[i] - a.length[a.parent[i]];
14   }
15   cout << ans << endl;
16   return 0;
17 }
```

Manacher

```
1  string longest_palindrome(string& s) {
2    // init "abc" -> "^$a#b#c$"
3    vector<char> t{'^', '#'};
4    for (char c : s) t.push_back(c), t.push_back('#');
5    t.push_back('$');
6    // manacher
7    int n = t.size(), r = 0, c = 0;
8    vector<int> p(n, 0);
9    for (int i = 1; i < n - 1; i++) {
10     if (i < r + c) p[i] = min(p[2 * c - i], r + c - i);
11     while (t[i + p[i] + 1] == t[i - p[i] - 1]) p[i]++;
12     if (i + p[i] > r + c) r = p[i], c = i;
13   }
14   // s[i] -> p[2 * i + 2] (even), p[2 * i + 2] (odd)
15   // output answer
16   int index = 0;
17   for (int i = 0; i < n; i++)
18     if (p[index] < p[i]) index = i;
19   return s.substr((index - p[index]) / 2, p[index]);
20 }
```

Lyndon

- def: suf(s) > s

```
1  void duval(const string &s) {
2    int n = (int)s.size();
3    for (int i = 0; i < n;) {
4      int j = i, k = i + 1;
5      for (; j < n && s[j] <= s[k]; j++, k++)
6        if (s[j] < s[k]) j = i - 1;
7
8      while (i <= j) {
9        // cout << s.substr(i, k - j) << '\n';
10       i += k - j;
11     }
12   }
13 }
```

minimal representation

```
1  int k = 0, i = 0, j = 1;
2  while (k < n && i < n && j < n) {
3    if (s[(i + k) % n] == s[(j + k) % n]) {
4      k++;
5    } else {
6      s[(i + k) % n] > s[(j + k) % n] ? i = i + k + 1 : j = j +
   ↪   k + 1;
7      if (i == j) i++;
8      k = 0;
9    }
10 }
11 i = min(i, j); // from 0
```

suffix array

```
1  vi classTable[21];
2  vector<int> suffix_array(string const& s) {
3      forn(i, 21) classTable[i].clear();
4
5      int n = s.size();
6      const int alphabet = 256;
7      vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
8      for (int i = 0; i < n; i++)
9          cnt[s[i]]++;
10     for (int i = 1; i < alphabet; i++)
11         cnt[i] += cnt[i-1];
12     for (int i = 0; i < n; i++)
13         p[--cnt[s[i]]] = i;
14     c[p[0]] = 0;
15     int classes = 1;
16     for (int i = 1; i < n; i++) {
17         if (s[p[i]] != s[p[i-1]])
18             classes++;
19         c[p[i]] = classes - 1;
20     }
21     classTable[0] = c;
22     vector<int> pn(n), cn(n);
23     for (int h = 0; (1 << h) < n; ++h) {
24         for (int i = 0; i < n; i++) {
25             pn[i] = p[i] - (1 << h);
26             if (pn[i] < 0)
27                 pn[i] += n;
28         }
29         fill(cnt.begin(), cnt.begin() + classes, 0);
30         for (int i = 0; i < n; i++)
31             cnt[c[pn[i]]]++;
32         for (int i = 1; i < classes; i++)
33             cnt[i] += cnt[i-1];
34         for (int i = n-1; i >= 0; i--)
35             p[--cnt[c[pn[i]]]] = pn[i];
36         cn[p[0]] = 0;
37         classes = 1;
38         for (int i = 1; i < n; i++) {
39             pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h))
   ↪   % n]};
40             pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1
   ↪   << h)) % n]};
```

```
41          if (cur != prev)
42              ++classes;
43          cn[p[i]] = classes - 1;
44      }
45      c.swap(cn);
46      classTable[h+1] = c;
47    }
48    return p;
49  }
50
51  int lcp(int a, int b) {
52    int ans = 0;
53    for(int i = 19; i >= 0; i--) {
54        if(classTable[i].size() == 0) continue;
55        if(classTable[i][a] == classTable[i][b]) {
56            a += (1 << i);
57            b += (1 << i);
58            ans += (1 << i);
59        }
60    }
61    return ans;
62  }
```