Nea1's Code Library

Nea1

ORZ

He is Nea1

Contents

ntro	2
Main template	2
Fast IO	2
Pragmas (lol)	. 2
Oata Structures	3
Segment Tree	
Recursive	
Iterating	
Union Find	
Fenwick Tree	7
PBDS	
Treap	
Implicit treap	
Persistent implicit treap	. 11
2D Sparsa Tabla	11

Intro

Main template

```
#include <bits/stdc++.h>
    using namespace std;
2
3
    #define FOR(x,n) for(int x=0;x< n;x++)
4
    #define form(i, n) for (int i = 0; i < int(n); i++)
    #define all(v) v.begin(), v.end()
6
    using ll = long long;
    using ld = long double;
    using pii = pair<int, int>;
9
    const char nl = '\n';
10
11
    int main() {
12
      cin.tie(nullptr)->sync_with_stdio(false);
13
      cout << fixed << setprecision(20);</pre>
14
      // mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
15
16
    Fast IO
    namespace io {
    constexpr int SIZE = 1 << 16;</pre>
2
    char buf[SIZE], *head, *tail;
3
    char get_char() {
      if (head == tail) tail = (head = buf) + fread(buf, 1, SIZE, stdin);
5
      return *head++;
6
7
    11 read() {
      11 x = 0, f = 1;
9
      char c = get_char();
10
      for (; !isdigit(c); c = get_char()) (c == '-') && (f = -1);
11
      for (; isdigit(c); c = get_char()) x = x * 10 + c - '0';
12
13
      return x * f;
    }
14
    string read_s() {
15
16
      string str;
      char c = get_char();
17
      while (c == ' ' || c == '\n' || c == '\r') c = get_char();
      while (c != ' ' && c != '\n' && c != '\r') str += c, c = get_char();
19
20
      return str;
21
    void print(int x) {
22
      if (x > 9) print(x / 10);
23
      putchar(x % 10 | '0');
24
25
    void println(int x) { print(x), putchar('\n'); }
26
27
      Read& operator>>(11& x) { return x = read(), *this; }
28
      Read& operator>>(long double& x) { return x = stold(read_s()), *this; }
29
30
    } // namespace io
31
    Pragmas (lol)
    #pragma GCC optimize(2)
1
    #pragma GCC optimize(3)
    #pragma GCC optimize("Ofast")
    #pragma GCC optimize("inline")
    #pragma GCC optimize("-fgcse")
    #pragma GCC optimize("-fgcse-lm")
    #pragma GCC optimize("-fipa-sra")
#pragma GCC optimize("-ftree-pre")
    #pragma GCC optimize("-ftree-vrp")
   #pragma GCC optimize("-fpeephole2")
   #pragma GCC optimize("-ffast-math")
11
    #pragma GCC optimize("-fsched-spec")
```

#pragma GCC optimize("unroll-loops")

```
#pragma GCC optimize("-falign-jumps")
14
    #pragma GCC optimize("-falign-loops")
15
    #pragma GCC optimize("-falign-labels")
16
    #pragma GCC optimize("-fdevirtualize")
17
    #pragma GCC optimize("-fcaller-saves")
18
    #pragma GCC optimize("-fcrossjumping")
19
    #pragma GCC optimize("-fthread-jumps")
20
    #pragma GCC optimize("-funroll-loops")
21
    #pragma GCC optimize("-fwhole-program")
22
    #pragma GCC optimize("-freorder-blocks")
23
    #pragma GCC optimize("-fschedule-insns")
24
    #pragma GCC optimize("inline-functions")
    #pragma GCC optimize("-ftree-tail-merge")
26
    #pragma GCC optimize("-fschedule-insns2")
27
    #pragma GCC optimize("-fstrict-aliasing")
28
    #pragma GCC optimize("-fstrict-overflow")
29
    #pragma GCC optimize("-falign-functions")
    #pragma GCC optimize("-fcse-skip-blocks")
31
    #pragma GCC optimize("-fcse-follow-jumps")
    #pragma GCC optimize("-fsched-interblock")
33
    #pragma GCC optimize("-fpartial-inlining")
34
    #pragma GCC optimize("no-stack-protector")
35
    #pragma GCC optimize("-freorder-functions")
36
    #pragma GCC optimize("-findirect-inlining")
    #pragma GCC optimize("-fhoist-adjacent-loads")
38
    #pragma GCC optimize("-frerun-cse-after-loop")
39
    #pragma GCC optimize("inline-small-functions")
40
    #pragma GCC optimize("-finline-small-functions")
41
    #pragma GCC optimize("-ftree-switch-conversion")
42
    #pragma GCC optimize("-foptimize-sibling-calls")
43
    #pragma GCC optimize("-fexpensive-optimizations")
44
    #pragma GCC optimize("-funsafe-loop-optimizations")
45
    #pragma GCC optimize("inline-functions-called-once")
46
    #pragma GCC optimize("-fdelete-null-pointer-checks")
47
    #pragma GCC target("sse,sse2,sse3,sse4.1,sse4.2,avx,avx2,popcnt,tune=native")
48
```

Data Structures

Segment Tree

Recursive

• Implicit segment tree, range query + point update

```
struct Node {
      int lc, rc, p;
2
    };
3
4
    struct SegTree {
5
      vector<Node> t = {{}};
6
      SegTree(int n) { t.reserve(n * 40); }
      int modify(int p, int l, int r, int x, int v) {
        int u = p;
9
        if (p == 0) {
10
          t.push_back(t[p]);
11
          u = (int)t.size() - 1;
12
13
        if (r - 1 == 1) {
14
           t[u].p = t[p].p + v;
15
        } else {
16
           int m = (1 + r) / 2;
17
18
           if (x < m) {
             t[u].lc = modify(t[p].lc, 1, m, x, v); // ub before c++17
19
20
           } else {
             t[u].rc = modify(t[p].rc, m, r, x, v);
21
22
          t[u].p = t[t[u].lc].p + t[t[u].rc].p;
23
24
25
        return u;
26
```

```
int query(int p, int 1, int r, int x, int y) {
27
28
         if (x <= 1 && r <= y) return t[p].p;</pre>
         int m = (1 + r) / 2, res = 0;
29
         if (x < m) res += query(t[p].lc, 1, m, x, y);</pre>
30
31
         if (y > m) res += query(t[p].rc, m, r, x, y);
        return res;
32
33
    };
34
        • Persistent implicit, range query + point update
1
2
      int lc = 0, rc = 0, p = 0;
    };
3
    struct SegTree {
5
      vector<Node> t = {{}}; // init all
6
      SegTree() = default;
      SegTree(int n) { t.reserve(n * 20); }
8
      int modify(int p, int l, int r, int x, int v) {
        // p: original node, update a[x] \rightarrow v
10
        t.push_back(t[p]);
11
         int u = (int)t.size() - 1;
12
        if (r - 1 == 1) {
13
          t[u].p = v;
        } else {
15
16
           int m = (1 + r) / 2;
           if (x < m) {
17
            t[u].lc = modify(t[p].lc, 1, m, x, v);
18
19
             t[u].rc = t[p].rc;
           } else {
20
21
             t[u].lc = t[p].lc;
             t[u].rc = modify(t[p].rc, m, r, x, v);
22
23
          t[u].p = t[t[u].lc].p + t[t[u].rc].p;
24
25
26
        return u;
      }
27
      int query(int p, int l, int r, int x, int y) {
28
        // query sum a[x]...a[y-1] rooted at p
29
30
         // t[p] holds the info of [l, r)
         if (x <= 1 && r <= y) return t[p].p;</pre>
31
         int m = (1 + r) / 2, res = 0;
32
         if (x < m) res += query(t[p].lc, l, m, x, y);
         if (y > m) res += query(t[p].rc, m, r, x, y);
34
35
         return res;
36
    };
37
```

Iterating

• Iterating, range query + point update

```
struct Node {
     11 v = 0, init = 0;
2
    };
3
    Node pull(const Node &a, const Node &b) {
5
      if (!a.init) return b;
      if (!b.init) return a;
      Node c;
      return c;
9
10
11
    struct SegTree {
12
      11 n;
      vector<Node> t:
14
      SegTree(ll_n) : n(_n), t(2 * n){};
15
      void modify(ll p, const Node &v) {
16
        t[p += n] = v;
17
        for (p \neq 2; p; p \neq 2) t[p] = pull(t[p * 2], t[p * 2 + 1]);
```

```
}
19
20
      Node query(ll 1, ll r) {
21
        Node left, right;
         for (1 += n, r += n; 1 < r; 1 /= 2, r /= 2) {
22
23
           if (1 & 1) left = pull(left, t[1++]);
           if (r & 1) right = pull(t[--r], right);
24
25
        return pull(left, right);
26
      }
27
28
    };
        • Iterating, range query + range update
    struct SegTree {
1
      11 n, h = 0;
      vector<Node> t;
3
      SegTree(ll _n) \; : \; n(_n), \; h((ll)log2(n)), \; t(n * 2) \; \{\}
4
      void apply(ll x, ll v) {
         if (v == 0) {
6
          t[x].one = 0;
8
        } else {
           t[x].one = t[x].total;
9
10
        t[x].lazy = v;
11
12
      }
      void build(11 1) {
13
14
        for (1 = (1 + n) / 2; 1 > 0; 1 /= 2) {
           if (t[1].lazy == -1) {
15
             t[1] = pull(t[1 * 2], t[1 * 2 + 1]);
16
17
           }
        }
18
19
      }
      void push(11 1) {
20
        1 += n;
21
        for (ll s = h; s > 0; s--) {
22
          ll i = 1 >> s;
23
24
           if (t[i].lazy != -1) {
             apply(2 * i, t[i].lazy);
25
             apply(2 * i + 1, t[i].lazy);
26
27
          t[i].lazy = -1;
28
29
        }
30
      void modify(ll 1, ll r, int v) {
31
         push(1), push(r - 1);
32
         11\ 10 = 1, r0 = r;
33
         for (1 += n, r += n; 1 < r; 1 /= 2, r /= 2) {
34
           if (1 & 1) apply(1++, v);
35
36
           if (r & 1) apply(--r, v);
37
        build(10), build(r0 - 1);
38
39
      Node query(11 1, 11 r) {
40
41
        push(1), push(r - 1);
         Node left, right;
42
43
         for (1 += n, r += n; 1 < r; 1 /= 2, r /= 2) {
          if (1 & 1) left = pull(left, t[1++]);
44
45
           if (r & 1) right = pull(t[--r], right);
         }
46
        return pull(left, right);
47
48
      }
    };
49
        • AtCoder Segment Tree (recursive structure but iterative)
    template <class T> struct PointSegmentTree {
2
      int size = 1;
      vector<T> tree;
3
      PointSegmentTree(int n) : PointSegmentTree(vector<T>(n)) {}
      PointSegmentTree(vector<T>& arr) {
5
        while(size < (int)arr.size())</pre>
6
           size <<= 1;
```

```
tree = vector<T>(size << 1);</pre>
8
9
        for(int i = size + arr.size() - 1; i >= 1; i--)
          if(i >= size) tree[i] = arr[i - size];
10
          else consume(i);
11
12
      }
      void set(int i, T val) {
13
        tree[i += size] = val;
14
        for(i >>= 1; i >= 1; i >>= 1)
15
          consume(i);
16
17
      T get(int i) { return tree[i + size]; }
18
19
      T query(int 1, int r) {
20
        T resl, resr;
        for(1 += size, r += size + 1; 1 < r; 1 >>= 1, r >>= 1) {
21
          if(1 & 1) resl = resl * tree[l++];
22
          if(r & 1) resr = tree[--r] * resr;
23
24
        return resl * resr;
25
26
      T query_all() { return tree[1]; }
27
      void consume(int i) { tree[i] = tree[i << 1] * tree[i << 1 | 1]; }</pre>
28
29
30
31
    struct SegInfo {
32
      11 v;
33
      SegInfo() : SegInfo(0) {}
34
      SegInfo(ll val) : v(val) {}
35
36
      SegInfo operator*(SegInfo b) {
        return SegInfo(v + b.v);
37
38
    };
39
    Union Find
    vector<int> p(n);
1
2
    iota(p.begin(), p.end(), 0);
    function<int(int)> find = [\(\alpha\)](int x) { return p[x] == x ? x : (p[x] = find(p[x])); };
3
    auto merge = [\&] (int x, int y) { p[find(x)] = find(y); };
        • Persistent version
    struct Node {
1
      int lc, rc, p;
2
3
    };
4
5
    struct SegTree {
      6
      SegTree() = default;
      SegTree(int n) { t.reserve(n * 20); }
      int modify(int p, int l, int r, int x, int v) {
9
        // p: original node, update a[x] \rightarrow v
10
        t.push_back(t[p]);
11
        int u = (int)t.size() - 1;
12
        if (r - 1 == 1) {
13
          t[u].p = v;
14
15
        } else {
          int m = (1 + r) / 2;
16
          if (x < m) {
17
18
            t[u].lc = modify(t[p].lc, 1, m, x, v);
            t[u].rc = t[p].rc;
19
          } else {
20
            t[u].lc = t[p].lc;
21
22
            t[u].rc = modify(t[p].rc, m, r, x, v);
23
          t[u].p = t[t[u].lc].p + t[t[u].rc].p;
24
        }
25
26
        return u;
      }
27
      int query(int p, int 1, int r, int x, int y) {
28
        // query sum a[x]...a[y-1] rooted at p
```

```
// t[p] holds the info of [l, r)
30
31
         if (x <= 1 && r <= y) return t[p].p;</pre>
         int m = (1 + r) / 2, res = 0;
32
         if (x < m) res += query(t[p].lc, l, m, x, y);</pre>
33
34
         if (y > m) res += query(t[p].rc, m, r, x, y);
        return res;
35
36
    };
37
38
    struct DSU {
39
      int n:
40
      SegTree seg;
41
42
       DSU(int _n) : n(_n), seg(n) {}
       int get(int p, int x) { return seg.query(p, 0, n, x, x + 1); }
43
44
       int set(int p, int x, int v) { return seg.modify(p, 0, n, x, v); }
       int find(int p, int x) {
45
46
         int parent = get(p, x);
         if (parent < 0) return x;</pre>
47
48
        return find(p, parent);
      }
49
       int is_same(int p, int x, int y) { return find(p, x) == find(p, y); }
50
51
       int merge(int p, int x, int y) {
        int rx = find(p, x), ry = find(p, y);
52
         if (rx == ry) return -1;
         int rank_x = -get(p, rx), rank_y = -get(p, ry);
54
         if (rank_x < rank_y) {</pre>
55
56
           p = set(p, rx, ry);
         } else if (rank_x > rank_y) {
57
58
          p = set(p, ry, rx);
        } else {
59
60
          p = set(p, ry, rx);
          p = set(p, rx, -rx - 1);
61
62
         return p;
63
      }
64
    };
```

Fenwick Tree

askd version

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

• Nea1 version

```
1 template <typename T>
2
    struct Fenwick {
      const int n;
3
      vector<T> a;
4
      Fenwick(int n) : n(n), a(n) {}
      void add(int x, T v) {
6
        for (int i = x + 1; i <= n; i += i & -i) {
7
          a[i - 1] += v;
8
        }
9
10
      }
      T sum(int x) {
11
12
        T ans = 0;
        for (int i = x; i > 0; i -= i \& -i) {
13
          ans += a[i - 1];
14
        7
15
        return ans;
16
17
      T rangeSum(int 1, int r) { return sum(r) - sum(1); }
18
    };
    PBDS
    #include <bits/stdc++.h>
    #include <ext/pb ds/assoc container.hpp>
    using namespace std;
    using namespace __gnu_pbds;
    template<typename T>
    using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
    template<typename T, typename X>
    using ordered_map = tree<T, X, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
9
    template<typename T, typename X>
    using fast_map = cc_hash_table<T, X>;
    template<typename T, typename X>
11
    using ht = gp_hash_table<T, X>;
13
    mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
14
15
    struct splitmix64 {
        size_t operator()(size_t x) const {
16
17
             static const size_t fixed = chrono::steady_clock::now().time_since_epoch().count();
             x += 0x9e3779b97f4a7c15 + fixed;
18
            x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;

x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
19
20
            return x \hat{ } (x >> 31);
21
22
    };
23
    Treap
        • (No rotation version)
    struct Node {
      Node *1, *r;
2
3
      int s, sz;
      // int t = 0, a = 0, g = 0; // for lazy propagation
4
5
      Node(int _s) : 1(nullptr), r(nullptr), s(_s), sz(1), w(rng()) {}
7
      void apply(int vt, int vg) {
8
9
        // for lazy propagation
        // s -= vt;
10
        // t += vt, a += vg, g += vg;
11
12
13
      void push() {
        // for lazy propagation
14
        // if (l != nullptr) l->apply(t, g);
15
        // if (r != nullptr) r->apply(t, g);
16
        // t = g = 0;
17
      void pull() { sz = 1 + (1 ? 1->sz : 0) + (r ? r->sz : 0); }
19
```

```
21
22
    std::pair<Node *, Node *> split(Node *t, int v) {
      if (t == nullptr) return {nullptr, nullptr};
23
24
       t->push();
       if (t->s < v) {
25
        auto [x, y] = split(t->r, v);
26
27
        t->r = x;
        t->pull();
28
        return {t, y};
29
      } else {
30
        auto [x, y] = split(t->1, v);
31
32
         t->1 = y;
33
        t->pull();
        return {x, t};
34
35
36
37
    Node *merge(Node *p, Node *q) {
38
       if (p == nullptr) return q;
       if (q == nullptr) return p;
40
       if (p->w < q->w) swap(p, q);
41
       auto [x, y] = split(q, p->s + rng() % 2);
42
      p->push();
43
      p->1 = merge(p->1, x);
      p->r = merge(p->r, y);
45
46
      p->pull();
47
      return p;
48
49
    Node *insert(Node *t, int v) {
50
      auto [x, y] = split(t, v);
51
      return merge(merge(x, new Node(v)), y);
52
53
    Node *erase(Node *t, int v) {
55
       auto [x, y] = split(t, v);
56
       auto [p, q] = split(y, v + 1);
57
      return merge(merge(x, merge(p->1, p->r)), q);
58
59
60
61
    int get_rank(Node *&t, int v) {
      auto [x, y] = split(t, v);
62
       int res = (x ? x->sz : 0) + 1;
63
64
      t = merge(x, y);
65
      return res;
    }
66
67
    Node *kth(Node *t, int k) {
69
70
      while (true) {
        int left_sz = t->1 ? t->1->sz : 0;
71
        if (k < left_sz) {</pre>
72
73
          t = t->1;
        } else if (k == left_sz) {
74
75
          return t;
        } else {
76
          k = left_sz + 1, t = t->r;
77
        }
78
      }
79
    }
80
81
    Node *get_prev(Node *&t, int v) {
82
83
      auto [x, y] = split(t, v);
      Node *res = kth(x, x->sz);
84
85
       t = merge(x, y);
      return res;
86
87
88
    Node *get_next(Node *&t, int v) {
89
      auto [x, y] = split(t, v + 1);
90
      Node *res = kth(y, 1);
91
```

```
t = merge(x, y);
92
93
      return res;
   }
94
        • USAGE
    int main() {
1
      cin.tie(nullptr)->sync_with_stdio(false);
2
      int n;
3
      cin >> n;
      Node *t = nullptr;
5
      for (int op, x; n--;) {
6
        cin >> op >> x;
        if (op == 1) {
8
          t = insert(t, x);
        } else if (op == 2) {
10
          t = erase(t, x);
11
        } else if (op == 3) {
12
          cout << get_rank(t, x) << "\n";</pre>
13
        } else if (op == 4) {
          cout << kth(t, x)->s << "\n";
15
        } else if (op == 5) {
16
          cout << get_prev(t, x)->s << "\n";
17
        } else {
18
19
          cout << get_next(t, x)->s << "\n";</pre>
        }
20
21
      }
    }
22
```

Implicit treap

• Split by size

```
struct Node {
       Node *1, *r;
2
       int s, sz;
3
       // int lazy = 0;
 4
       11 w;
5
       Node(int _s) : l(nullptr), r(nullptr), s(_s), sz(1), w(rnd()) {}
8
       void apply() {
        // for lazy propagation
9
        // lazy ^= 1;
10
11
       }
       void push() {
12
        // for lazy propagation
// if (lazy) {
13
14
         // swap(l, r);
15
         // if (l != nullptr) l->apply();
16
         // if (r != nullptr) r->apply();
// lazy = 0;
17
18
         // }
19
20
       void pull() { sz = 1 + (1 ? 1-> sz : 0) + (r ? r-> sz : 0); }
^{21}
22
23
     std::pair<Node *, Node *> split(Node *t, int v) {
24
       // first \rightarrow sz == v
25
26
       if (t == nullptr) return {nullptr, nullptr};
       t->push();
27
       int left_sz = t->1 ? t->1->sz : 0;
28
       if (left_sz < v) {</pre>
29
         auto [x, y] = split(t->r, v - left_sz - 1);
         t->r = x;
31
32
         t->pull();
33
        return {t, y};
       } else {
34
         auto [x, y] = split(t->1, v);
35
         t->1 = y;
36
37
         t->pull();
         return {x, t};
38
```

```
}
39
40
    }
41
    Node *merge(Node *p, Node *q) {
42
43
       if (p == nullptr) return q;
       if (q == nullptr) return p;
44
45
       if (p->w < q->w) {
        p->push();
46
        p->r = merge(p->r, q);
47
48
        p->pull();
        return p;
49
50
       } else {
         q->push();
51
         q->1 = merge(p, q->1);
52
53
         q->pull();
        return q;
54
55
    }
56
```

Persistent implicit treap

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

2D Sparse Table

• Sorry that this sucks - askd

```
template <class T, class Compare = less<T>>
truct SparseTable2d {
   int n = 0, m = 0;
   T**** table;
   int* log;
   inline T choose(T x, T y) {
    return Compare()(x, y) ? x : y;
}
```

```
SparseTable2d(vector<vector<T>>& grid) {
9
10
        if(grid.empty() || grid[0].empty()) return;
        n = grid.size(); m = grid[0].size();
11
        log = new int[max(n, m) + 1];
12
13
        log[1] = 0;
        for(int i = 2; i <= max(n, m); i++)</pre>
14
          log[i] = log[i - 1] + ((i ^ (i - 1)) > i);
15
        table = new T***[n];
16
        for(int i = n - 1; i >= 0; i--) {
17
          table[i] = new T**[m];
18
          for(int j = m - 1; j >= 0; j--) {
19
20
            table[i][j] = new T*[log[n - i] + 1];
            for(int k = 0; k \le log[n - i]; k++) {
21
             table[i][j][k] = new T[log[m - j] + 1];
22
             if(!k) table[i][j][k][0] = grid[i][j];
23
             else table[i][j][k][0] = choose(table[i][j][k-1][0], table[i+(1<<(k-1))][j][k-1][0]);
24
25
             for(int 1 = 1; 1 <= log[m - j]; 1++)
               table[i][j][k][1] = choose(table[i][j][k][1-1], \ table[i][j+(1<<(1-1))][k][1-1]);
26
         }
28
       }
29
      }
30
      T query(int r1, int r2, int c1, int c2) {
31
32
        assert(r1 >= 0 && r2 < n && r1 <= r2);
        assert(c1 >= 0 \&\& c2 < m \&\& c1 <= c2);
33
        int rl = log[r2 - r1 + 1], cl = log[c2 - c1 + 1];
34
        T ca1 = choose(table[r1][c1][r1][c1], table[r2-(1<<r1)+1][c1][r1][c1]);
35
        36
37
        return choose(ca1, ca2);
      }
38
    };
39
       • USAGE
    vector<vector<int>> test = {
1
     \{1, 2, 3, 4\}, \{2, 3, 4, 5\}, \{9, 9, 9, 9\}, \{-1, -1, -1, -1\}
2
3
    };
    SparseTable2d<int> st(test);
                                               // Range min query
    SparseTable2d<int,greater<int>>> st2(test); // Range max query
```