

```

1 ===== bittrie.cpp =====
2 class BitTrie
3 {
4 public:
5     static constexpr int BITS = 31;          // for 32-bit ints use 31 down to 0
6     vector<array<int,2>> nxt;
7     vector<int> cnt;                          // count of numbers ending exactly at this node
8     vector<int> pref;                        // count of numbers passing through this node
9     int nodes = 0;
10
11     BitTrie(int maxNodes = 1e6+1)
12     {
13         nxt.reserve(maxNodes);
14         cnt.reserve(maxNodes);
15         pref.reserve(maxNodes);
16         newNode();
17     }
18
19     int newNode()
20     {
21         nxt.push_back({0,0});
22         cnt.push_back(0);
23         pref.push_back(0);
24         return nodes++;
25     }
26
27     // insert x (f=+1) or remove x (f=-1)
28     void insert(int x, int f = 1)
29     {
30         int u = 0;
31         for (int b = BITS; b >= 0; --b)
32         {
33             int bit = (x >> b) & 1;
34             if (!nxt[u][bit])
35                 nxt[u][bit] = newNode();
36             u = nxt[u][bit];
37             pref[u] += f;
38         }
39         cnt[u] += f;
40     }
41
42     void remove(int x) { insert(x, -1); }
43
44     // exact count of x
45     int count(int x) const
46     {
47         int u = 0;
48         for (int b = BITS; b >= 0; --b)
49         {
50             int bit = (x >> b) & 1;
51             int v = nxt[u][bit];
52             if (!v || pref[v] <= 0) return 0;
53             u = v;
54         }
55         return cnt[u];
56     }
57
58     // maximize x^y over all y in the trie
59     int maxXor(int x) const
60     {
61         int u = 0;
62         int ans = 0;
63         for (int b = BITS; b >= 0; --b)
64         {

```

```

65         int bit = (x >> b) & 1;
66         int want = bit ^ 1;
67         if (nxt[u][want] && pref[nxt[u][want]] > 0)
68         {
69             ans |= (1 << b);
70             u = nxt[u][want];
71         } else
72         {
73             u = nxt[u][bit];
74         }
75     }
76     return ans;
77 }
78
79 // count of elements      x
80 int leq(int x) const
81 {
82     int u = 0, res = 0;
83     for (int b = BITS; b >= 0; --b)
84     {
85         int bit = (x >> b) & 1;
86         if (bit == 1)
87         {
88             if(nxt[u][0])
89                 res += pref[nxt[u][0]];
90             u = nxt[u][1];
91         }
92         else
93         {
94             u = nxt[u][0];
95         }
96     }
97     if(u)
98         res += cnt[u];
99     return res;
100 }
101 };
102
103
104 ===== block_decomposition.cpp =====
105 class element_chan
106 {
107 public:
108 };
109 class block_chan
110 {
111 public:
112 };
113
114 template<typename E, typename T, const int B>
115 class block_decomposition_chan
116 {
117 public:
118     static int ceil_div(int x, int y)
119     {
120         return (x + y - 1)/y;
121     }
122     static int block_id(int i)
123     {
124         return i/B;
125     }
126     static int lb(int bid)
127     {
128         return bid * B;
129     }

```

```

130     static int rb(int bid)
131     {
132         return min(n, (bid + 1) * B - 1);
133     }
134
135 public:
136     int n;
137     vector<E> element;
138     vector<T> block;
139
140     block_decomposition_chan(int n, vector<E> a, vector<T> b) : n(n), element(a), block(b)
141     {
142     };
143
144     void process(int l, int r, auto block_brute, auto block_quick)
145     {
146         assert(1 <= l and l <= r and r <= n);
147         int bl = block_id(l), br = block_id(r);
148         if(bl == br)
149             block_brute(l, r);
150         else
151         {
152             block_brute(l, rb(bl));
153             for(int b = bl + 1; b < br; b++)
154                 block_quick(b);
155             block_brute(lb(br), r);
156         }
157     }
158 };
159
160
161 ===== bridges.cpp =====
162 vector<bool> visited;
163 vector<int> tin, low;
164 int timer;
165 timer = 0;
166 visited.assign(n, false);
167 tin.assign(n, -1);
168 low.assign(n, -1);
169
170 void dfs(int v, int p = -1) {
171     visited[v] = true;
172     tin[v] = low[v] = timer++;
173     bool parent_skipped = false;
174     for (int to : adj[v]) {
175         if (to == p && !parent_skipped) { // do this for multiple edges
176             parent_skipped = true;
177             continue;
178         }
179         if (visited[to]) {
180             low[v] = min(low[v], tin[to]);
181         } else {
182             dfs(to, v);
183             low[v] = min(low[v], low[to]);
184             if (low[to] > tin[v])
185                 IS_BRIDGE(v, to);
186         }
187     }
188 }
189
190
191 ===== centroid.cpp =====
192 int nodes = 0;
193 int subtree[N], parentcentroid[N];
194 set<int> g[N];

```

```

195
196
197 void dfs(int u, int par){
198     nodes++;
199     subtree[u] = 1;
200     for(auto &it:g[u]){
201         if(it == par)
202             continue;
203         dfs(it, u);
204         subtree[u] += subtree[it];
205     }
206 }
207
208 int centroid(int k, int parent){
209     for(auto it:g[k]){
210         if(it==parent)
211             continue;
212         if(subtree[it]>(nodes>>1))
213             return centroid(it,k);
214     }
215     return k;
216 }
217
218 void decompose(int u, int par){
219     nodes = 0;
220     dfs(u, u);
221     int node = centroid(u, u);
222     //do something
223     parentcentroid[node] = par;
224     for(auto &it:g[node]){
225         g[it].erase(node);
226         decompose(it, node);
227     }
228 }
229
230
231 /*
232 Properties of Centroid Tree(VERY IMP): https://www.quora.com/q/threadsiiiithyderabad/
233 Centroid-Decomposition-of-a-Tree
234 */
235
236
237 ===== combinatorics2d.cpp =====
238 template<typename T>
239 class combinatorics2d
240 {
241     public:
242     int n;
243     vector<vector<T>>> C;
244     combinatorics2d(int n) : n(n), C(n+1, vector<T>(n+1, 0))
245     {
246         for (int i = 0; i <= n; i++)
247         {
248             C[i][0] = C[i][i] = T(1);
249             for (int j = 1; j < i; j++)
250             {
251                 C[i][j] = C[i-1][j-1] + C[i-1][j];
252             }
253         }
254     }
255
256     T ncr(int N, int R)
257     {
258         if (R < 0 || R > N) return T(0);

```

```

259         return C[N][R];
260     }
261 };
262
263
264 ===== combinatorics.cpp =====
265 template<typename T, const int P>
266 class combinatorics
267 {
268     //combinatorics<mint,2> c(200);
269     //2 is the number whose power will be precomputed
270 public:
271     int n;
272     vector<T> inv, fac, ifac, pw;
273     combinatorics (int n) : n(n), inv(n+1), fac(n+1), ifac(n+1), pw(n+1)
274     {
275         fac[0] = inv[0] = ifac[0] = pw[0] = T(1);
276
277         for(int i = 1; i <= n; i++)
278             inv[i] = T(1)/T(i), fac[i] = fac[i - 1] * T(i), ifac[i] = ifac[i - 1] * inv[i]
279             ], pw[i] = pw[i - 1] * T(P);
280
281     }
282
283     T ncr(int n, int r)
284     {
285         if(n < r or r < 0)
286             return 0;
287         return fac[n] * ifac[r] * ifac[n - r];
288     }
289 };
290
291 ===== crt.cpp =====
292 #include<bits/stdc++.h>
293 using namespace std;
294
295 // Extended Euclidean Algorithm
296 int extended_gcd(int a, int b, int &x, int &y)
297 {
298     if (b == 0) { x = 1; y = 0; return a; }
299     int x1, y1;
300     int g = extended_gcd(b, a % b, x1, y1);
301     x = y1;
302     y = x1 - (a / b) * y1;
303     return g;
304 }
305
306 // Modular inverse using extended Euclid
307 int modinv(int a, int m)
308 {
309     int x, y;
310     int g = extended_gcd(a, m, x, y);
311     if (g != 1) return -1; // Inverse doesn't exist
312     return (x % m + m) % m;
313 }
314
315 // Chinese Remainder Theorem solver
316 int crt(vector<int> &a, vector<int> &m)
317 {
318     int M = 1;
319     for (int mi : m) M *= mi;
320
321     int result = 0;
322     for (int i = 0; i < a.size(); ++i) {
323         int Mi = M / m[i];

```

```

323     int inv = modinv(Mi, m[i]);
324     result = (result + a[i] * Mi % M * inv % M) % M;
325 }
326
327 return (result % M + M) % M; // Ensure positive
328 }
329
330
331
332 ===== custom_hash.cpp =====
333 // #include<bits/extc++.h>
334 #include <ext/pb_ds/assoc_container.hpp>
335
336 struct splitmix64_hash {
337     static uint64_t splitmix64(uint64_t x) {
338         // http://xorshift.di.unimi.it/splitmix64.c
339         x += 0x9e3779b97f4a7c15;
340         x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
341         x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
342         return x ^ (x >> 31);
343     }
344
345     size_t operator()(uint64_t x) const {
346         static const uint64_t FIXED_RANDOM = std::chrono::steady_clock::now().
347             time_since_epoch().count();
348         return splitmix64(x + FIXED_RANDOM);
349     }
350 };
351
352 template <typename K, typename V, typename Hash = splitmix64_hash>
353 using HashMap = __gnu_pbds::gp_hash_table<K, V, Hash>;
354
355 template <typename K, typename Hash = splitmix64_hash>
356 using HashSet = HashMap<K, __gnu_pbds::null_type, Hash>;
357
358 ===== dsu.cpp =====
359 class dsu_chan
360 {
361 public:
362     int n;
363     vector<int> par, siz;
364     vector<int> nxt; // for fast skipping of deleted elements
365     vector<bool> dead; // whether element is deleted
366
367     dsu_chan(int n) : n(n), par(n), siz(n, 1), nxt(n + 1), dead(n, false)
368     {
369         iota(par.begin(), par.end(), 0);
370         iota(nxt.begin(), nxt.end(), 0);
371     }
372
373     int get(int x)
374     {
375         return (par[x] == x ? x : par[x] = get(par[x]));
376     }
377
378     void unite(int x, int y)
379     {
380         x = get(x), y = get(y);
381         if (x == y) return;
382         if (siz[x] > siz[y]) swap(x, y);
383         par[x] = y;
384         siz[y] += siz[x];
385     }
386

```



```

452     }
453     return sum;
454 }
455 T query(int x1, int y1, int x2, int y2)
456 {
457     return query(x2, y2)
458         - query(x1 - 1, y2)
459         - query(x2, y1 - 1)
460         + query(x1 - 1, y1 - 1);
461 }
462 // 1 based and [x1..x2]*[y1..y2]
463 };
464
465
466 ===== fenwick.cpp =====
467 template <typename T>
468 class fenwick_tree_chan
469 {
470 public:
471     vector<T> fenw;
472     int n;
473     int pw;
474
475     fenwick_tree_chan() : n(0) {}
476     fenwick_tree_chan(int n) : n(n)
477     {
478         fenw.resize(n);
479         pw = (n == 0 ? 0 : 1ULL << (63 - __builtin_clzll(unsigned(n))));
480     }
481
482     // a[x] += v;
483     void modify(int x, T v)
484     {
485         assert(0 <= x && x < n);
486         while (x < n)
487         {
488             fenw[x] += v;
489             x |= x + 1;
490         }
491     }
492
493     /// sum of prefix [0, .. x]
494     T query(int x)
495     {
496         ++ x;
497         assert(0 <= x && x <= n);
498         T v{};
499         while (x > 0)
500         {
501             v += fenw[x - 1];
502             x &= x - 1;
503         }
504         return v;
505     }
506
507     // Returns the length of the longest prefix (0 indexed) with sum <= c
508     int max_prefix(T c)
509     {
510         T v{};
511         int at = 0;
512         for (int len = pw; len > 0; len >>= 1)
513         {
514             if (at + len <= n)
515             {
516                 auto nv = v;

```



```

517         nv += fenw[at + len - 1];
518         if (!(c < nv))
519         {
520             v = nv;
521             at += len;
522         }
523     }
524 }
525 assert(0 <= at && at <= n);
526 return at;
527 }
528 };
529
530 struct fenwick_lazy
531 {
532     fenwick_tree_chan<int> B1, B2;
533     int n;
534
535     fenwick_lazy(int n) : n(n), B1(n), B2(n) {}
536
537     void range_add(int l, int r, int v)
538     {
539         B1.modify(l, v);
540         if (r + 1 < n) B1.modify(r + 1, -v);
541         B2.modify(l, v * (l - 1));
542         if (r + 1 < n) B2.modify(r + 1, -v * r);
543     }
544
545     int prefix_sum(int x)
546     {
547         if (x < 0) return 0;
548         return B1.query(x) * x - B2.query(x);
549     }
550
551     int range_sum(int l, int r)
552     {
553         return prefix_sum(r) - prefix_sum(l - 1);
554     }
555 };
556
557
558
559 ===== heavy_light.cpp =====
560 template <class S,
561         auto op,
562         auto e,
563         class F,
564         auto mapping,
565         auto composition,
566         auto id,
567         const bool islazy,
568         const bool on_edge>
569 class heavy_light_chan
570 {
571     /*
572     info:
573         - 0 indexed
574         - range [pos[u], out[u]) represents subtree of u
575     vars:
576         - r = tree root
577         - heavy[u] = heavy child v (edge (u,v) is heavy edge)
578         - root[u] = It is the starting point of the heavy chain (for u-v it is u, if v
579           is heavy and u is not heavy).
580         - on_edge = true => values on edges. Internally, value of edge is stored at
581           lower node (node more away from root)

```

```

580     warning:
581         - handle segtree initialization correctly
582 */
583 using lazy_t = lazy_segtree<S, op, e, F, mapping, composition, id>;
584 using simple_t = simple_segtree<S, op, e, F, mapping>;
585 using seg_t = std::conditional_t<islazy, lazy_t, simple_t>;
586
587 public:
588     int n, r;
589     vector<int> par, heavy, dep, root, pos, out;
590     seg_t seg;
591     heavy_light_chan(int n, vector<vector<int>> adj, int r = 0) :
592     n(n), r(r), par(n, -1), heavy(n, -1), dep(n), root(n), pos(n), out(n),
593     seg(n)
594     {
595         assert(r < n);
596         auto dfs_sz = [&](int u, auto &&dfs) -> int
597         {
598             /* ensure that in adj list first child is the heavy child
599             int sz = 1, mx_sz = 0;
600             for(auto &v : adj[u])
601             {
602                 if(v != par[u])
603                 {
604                     par[v] = u, dep[v] = dep[u] + 1;
605                     int s = dfs(v, dfs);
606                     sz += s;
607                     if(s > mx_sz)
608                         heavy[u] = v, mx_sz = s, swap(adj[u][0], v);
609                 }
610             }
611             return sz;
612         };
613         int timer = 0;
614         auto dfs_hld = [&](int u, auto &&dfs) -> void
615         {
616             pos[u] = timer++;
617             for(auto v : adj[u])
618             {
619                 if(v != par[u])
620                 {
621                     root[v] = (heavy[u] == v ? root[u] : v);
622                     dfs(v, dfs);
623                 }
624             }
625             out[u] = timer ; // exclusive
626         };
627         par[r] = -1;
628         dep[r] = 0;
629         root[r] = r;
630         dfs_sz(r, dfs_sz);
631         dfs_hld(r, dfs_hld);
632     }
633     int lca(int u, int v)
634     {
635         for (; root[u] != root[v]; v = par[root[v]])
636         {
637             if (dep[root[u]] > dep[root[v]])
638             {
639                 swap(u, v);
640             }
641         }
642         return (dep[u] < dep[v] ? u : v);
643     }
644

```

```

645 // *process_path [u, v] in O(logn * logn)
646 template <typename O>
647 void process_path(int u, int v, O oper)
648 {
649     for (; root[u] != root[v]; v = par[root[v]])
650     {
651         if (dep[root[u]] > dep[root[v]])
652             swap(u, v);
653         oper(pos[root[v]], pos[v]);
654     }
655     if (dep[u] > dep[v])
656         swap(u, v);
657
658     if (!on_edge)
659         oper(pos[u], pos[v]);
660     else if (u != v)
661         oper(pos[u] + 1, pos[v]);
662 }
663
664 void set(int v, const S &value)
665 {
666     seg.set(pos[v], value);
667 }
668
669 S get(int v)
670 {
671     return seg.get(pos[v]);
672 }
673
674 void modify_path(int u, int v, const F &f)
675 {
676     process_path(u, v, [this, &f])(int l, int r)
677     {
678         seg.apply(l, r + 1, f); // convert inclusive [l,r] -> atcoder apply [l, r+1)
679     });
680 }
681
682 S query_path(int u, int v)
683 {
684     S res = e();
685     process_path(u, v, [this, &res])(int l, int r)
686     {
687         S part = seg.prod(l, r + 1);
688         res = op(res, part);
689     });
690     return res;
691 }
692
693 void modify_subtree(int u, const F &f)
694 {
695     if (!on_edge)
696     {
697         seg.apply(pos[u], out[u], f); // subtree = [pos[u], out[u])
698     }
699     else
700     {
701         // edges stored at children; exclude its parent values that it at u
702         if (pos[u] < out[u] - 1)
703             seg.apply(pos[u] + 1, out[u], f);
704     }
705 }
706 S query_subtree(int u)
707 {
708     if (on_edge)
709     {

```

```

710         if (pos[u] < out[u] - 1)
711         {
712             return seg.prod(pos[u] + 1, out[u]);
713         }
714         else
715         {
716             return e();
717         }
718     }
719     return seg.prod(pos[u], out[u]);
720 }
721
722 };
723
724
725
726 ===== lazy_segtree.cpp =====
727 template <class S,
728           auto op,
729           auto e,
730           class F,
731           auto mapping,
732           auto composition,
733           auto id>
734 struct lazy_segtree
735 {
736 public:
737     unsigned int bit_ceil(unsigned int n)
738     {
739         unsigned int x = 1;
740         while (x < (unsigned int)(n))
741             x *= 2;
742         return x;
743     }
744     int countr_zero(unsigned int n)
745     {
746         return __builtin_ctz(n);
747     }
748     lazy_segtree() : lazy_segtree(0) {}
749     explicit lazy_segtree(int n) : lazy_segtree(std::vector<S>(n, e())) {}
750     explicit lazy_segtree(const std::vector<S> &v) : _n((int)(v.size()))
751     {
752         size = (int)bit_ceil((unsigned int)(_n));
753         log = countr_zero((unsigned int)size);
754         d = std::vector<S>(2 * size, e());
755         lz = std::vector<F>(size, id());
756         for (int i = 0; i < _n; i++)
757             d[size + i] = v[i];
758         for (int i = size - 1; i >= 1; i--)
759         {
760             update(i);
761         }
762     }
763
764     void set(int p, S x)
765     {
766         assert(0 <= p && p < _n);
767         p += size;
768         for (int i = log; i >= 1; i--)
769             push(p >> i);
770         d[p] = x;
771         for (int i = 1; i <= log; i++)
772             update(p >> i);
773     }
774

```

```

775 S get(int p)
776 {
777     assert(0 <= p && p < _n);
778     p += size;
779     for (int i = log; i >= 1; i--)
780         push(p >> i);
781     return d[p];
782 }
783
784 S prod(int l, int r)
785 {
786     assert(0 <= l && l <= r && r <= _n);
787     if (l == r)
788         return e();
789
790     l += size;
791     r += size;
792
793     for (int i = log; i >= 1; i--)
794     {
795         if (((l >> i) << i) != l)
796             push(l >> i);
797         if (((r >> i) << i) != r)
798             push((r - 1) >> i);
799     }
800
801     S sml = e(), smr = e();
802     while (l < r)
803     {
804         if (l & 1)
805             sml = op(sml, d[l++]);
806         if (r & 1)
807             smr = op(d[--r], smr);
808         l >>= 1;
809         r >>= 1;
810     }
811
812     return op(sml, smr);
813 }
814
815 S all_prod() { return d[1]; }
816
817 void apply(int p, F f)
818 {
819     assert(0 <= p && p < _n);
820     p += size;
821     for (int i = log; i >= 1; i--)
822         push(p >> i);
823     d[p] = mapping(f, d[p]);
824     for (int i = 1; i <= log; i++)
825         update(p >> i);
826 }
827 void apply(int l, int r, F f)
828 {
829     assert(0 <= l && l <= r && r <= _n);
830     if (l == r)
831         return;
832
833     l += size;
834     r += size;
835
836     for (int i = log; i >= 1; i--)
837     {
838         if (((l >> i) << i) != l)
839             push(l >> i);

```

```

840         if (((r >> i) << i) != r)
841             push((r - 1) >> i);
842     }
843
844     {
845         int l2 = l, r2 = r;
846         while (l < r)
847         {
848             if (l & 1)
849                 all_apply(l++, f);
850             if (r & 1)
851                 all_apply(--r, f);
852             l >>= 1;
853             r >>= 1;
854         }
855         l = l2;
856         r = r2;
857     }
858
859     for (int i = 1; i <= log; i++)
860     {
861         if (((l >> i) << i) != l)
862             update(l >> i);
863         if (((r >> i) << i) != r)
864             update((r - 1) >> i);
865     }
866 }
867
868 template <bool (*g)(S)>
869 int max_right(int l)
870 {
871     return max_right(l, [](S x)
872                        { return g(x); });
873 }
874 template <class G>
875 int max_right(int l, G g)
876 {
877     assert(0 <= l && l <= _n);
878     assert(g(e()));
879     if (l == _n)
880         return _n;
881     l += size;
882     for (int i = log; i >= 1; i--)
883         push(l >> i);
884     S sm = e();
885     do
886     {
887         while (l % 2 == 0)
888             l >>= 1;
889         if (!g(op(sm, d[l])))
890         {
891             while (l < size)
892             {
893                 push(l);
894                 l = (2 * l);
895                 if (g(op(sm, d[l])))
896                 {
897                     sm = op(sm, d[l]);
898                     l++;
899                 }
900             }
901             return l - size;
902         }
903         sm = op(sm, d[l]);
904         l++;

```

```

905     } while ((l & -l) != l);
906     return _n;
907 }
908
909 template <bool (*g)(S)>
910 int min_left(int r)
911 {
912     return min_left(r, [] (S x)
913                     { return g(x); });
914 }
915 template <class G>
916 int min_left(int r, G g)
917 {
918     assert(0 <= r && r <= _n);
919     assert(g(e()));
920     if (r == 0)
921         return 0;
922     r += size;
923     for (int i = log; i >= 1; i--)
924         push((r - 1) >> i);
925     S sm = e();
926     do
927     {
928         r--;
929         while (r > 1 && (r % 2))
930             r >>= 1;
931         if (!g(op(d[r], sm)))
932         {
933             while (r < size)
934             {
935                 push(r);
936                 r = (2 * r + 1);
937                 if (g(op(d[r], sm)))
938                 {
939                     sm = op(d[r], sm);
940                     r--;
941                 }
942             }
943             return r + 1 - size;
944         }
945         sm = op(d[r], sm);
946     } while ((r & -r) != r);
947     return 0;
948 }

```

private:

```

951 int _n, size, log;
952 std::vector<S> d;
953 std::vector<F> lz;
954
955 void update(int k) { d[k] = op(d[2 * k], d[2 * k + 1]); }
956 void all_apply(int k, F f)
957 {
958     d[k] = mapping(f, d[k]);
959     if (k < size)
960         lz[k] = composition(f, lz[k]);
961 }
962 void push(int k)
963 {
964     all_apply(2 * k, lz[k]);
965     all_apply(2 * k + 1, lz[k]);
966     lz[k] = id();
967 }

```

```

968 };
969 /*

```

```

970 !index is 0 based [l,r)
971
972 S - segment tree node
973 op(left, right) - monoid merge of two S values
974 e() - identity element for op
975 F - lazy tag type
976 mapping - apply a tag f to a node value s
977 composition - combine two tags (f applied after g)
978 id - neutral tag (no-op)
979
980 void set(int p, S x) - ar[p] = x
981 S get(int p) - return ar[p]
982
983 S prod(int l, int r) - Combine values in [l, r) via op.
984 S all_prod() == prod for whole array
985
986 void apply(int p, F f) - Apply tag f only to index p.
987 void apply(int l, int r, F f) - Apply tag f to every index in [l, r).
988
989 int max_right(int l, G g) - max r s.t g(l...r-1) is true and g(l...r) is false; true
    for e()
990 int min_left(int r, G g) - min l s.t g(l...r-1) is true and g(l-1....r-1) is false;
    true for e()
991
992
993
994 struct S { int sum, size; };
995 S op(S left, S right) { return {left.sum + right.sum, left.size + right.size}; }
996 S e() { return {0, 0}; }
997 struct F { int x; bool is_set; }; // lazy tag type
998 S mapping(F f, S s) // apply tag f to segment s
999 {
1000     if (!f.is_set) return s;
1001     return {f.x * s.size, s.size};
1002 }
1003
1004 F composition(F f, F g) { // Compose two tags: new f after old g
1005     if (f.is_set) return f;
1006     return g;
1007 }
1008 F id() { return {0, false}; }
1009
1010 * constructor - expects vector<S>
1011
1012 */
1013
1014
1015 ===== lca.cpp =====
1016 class binary_lifter_chan
1017 {
1018     /*
1019     tc: O(n logn) preprocessing, O(logn) query
1020     ml: O(n logn)
1021
1022     info:
1023         1 indexed
1024         n -> number of nodes
1025         r -> root of the tree
1026         dep[u] -> depth of node u
1027         tin[u], tout[u] -> intime, outtime of node u
1028         up[u][i] -> stores 2^ith ancestor of u
1029     */
1030 public:
1031     int n, L, timer;
1032     vector<int> dep, tin, tout;

```



```

1033     vector<vector<int>>> up;
1034
1035     binary_lifter_chan(int n, int r, const vector<vector<int>>> &adj) :
1036     n(n), L(ceil(log2(n)) + 1), timer(0), dep(n), tin(n), tout(n), up(n, vector<int> (L, r
1037     ))
1038     {
1039         timer = 0;
1040         dep[r] = 0;
1041         dfs(r, r, adj);
1042     }
1043
1044     void dfs(int u, int p, const vector<vector<int>>> &adj)
1045     {
1046         tin[u] = ++ timer;
1047         up[u][0] = p;
1048
1049         for(int i = 1; i < L; ++i)
1050             up[u][i] = up[up[u][i - 1]][i - 1];
1051
1052         for(auto v : adj[u])
1053             if (v != p)
1054                 dep[v] = dep[u] + 1, dfs(v, u, adj);
1055
1056         tout[u] = ++ timer;
1057     }
1058
1059     int get_kth(int v, int k)
1060     {
1061         if(k != 0)
1062             for(int i = L - 1; i >= 0 and v > 0; i --)
1063                 if((1 << i) <= k)
1064                     k -= (1 << i), v = up[v][i];
1065         return v;
1066     }
1067
1068     bool is_anc(int anc, int v)
1069     {
1070         return tin[anc] <= tin[v] and tout[v] <= tout[anc];
1071     }
1072
1073     int lca(int u, int v)
1074     {
1075         if (is_anc(u, v))
1076             return u;
1077         if (is_anc(v, u))
1078             return v;
1079         for (int i = L - 1; i >= 0; --i)
1080             if (!is_anc(up[u][i], v))
1081                 u = up[u][i];
1082         return up[u][0];
1083     };
1084
1085
1086     ===== linear_sieve.cpp =====
1087     #include<bits/stdc++.h>
1088     using namespace std;
1089
1090     const int N = 10000000;
1091     vector<int> lp(N+1); //lowest prime factor
1092     vector<int> pr; // prime list
1093
1094     void sieve(){
1095         for(int i = 2; i <= N; ++i){
1096             if (lp[i] == 0) {

```

```

1097         lp[i] = i;
1098         pr.push_back(i);
1099     }
1100     for (int j = 0; i * pr[j] <= N && j < (int)pr.size(); ++j) {
1101         lp[i * pr[j]] = pr[j];
1102         if (pr[j] == lp[i]) {
1103             break;
1104         }
1105     }
1106 }
1107 }
1108
1109 map<int, int> factorize(int k)
1110 {
1111     map<int, int> mp;
1112     while(k > 1)
1113     {
1114         mp[lp[k]]++;
1115         k /= lp[k];
1116     }
1117     return mp;
1118 }
1119
1120
1121
1122 ===== matrix_op.cpp =====
1123 template <typename T, size_t N, size_t M, size_t K>
1124 array<array<T, K>, N> operator*(const array<array<T, M>, N>& a, const array<array<T, K>, M
1125 >& b) {
1126     array<array<T, K>, N> c;
1127     for (size_t i = 0; i < N; i++) {
1128         for (size_t j = 0; j < K; j++) {
1129             c[i][j] = 0;
1130             for (size_t k = 0; k < M; k++) {
1131                 c[i][j] += a[i][k] * b[k][j];
1132             }
1133         }
1134     }
1135     return c;
1136 }
1137
1138 template <typename T>
1139 vector<vector<T>> operator*(const vector<vector<T>>& a, const vector<vector<T>>& b) {
1140     if (a.empty() || b.empty()) {
1141         return {{}};
1142     }
1143     vector<vector<T>> c(a.size(), vector<T>(b[0].size()));
1144     for (int i = 0; i < static_cast<int>(c.size()); i++) {
1145         for (int j = 0; j < static_cast<int>(c[0].size()); j++) {
1146             c[i][j] = 0;
1147             for (int k = 0; k < static_cast<int>(b.size()); k++) {
1148                 c[i][j] += a[i][k] * b[k][j];
1149             }
1150         }
1151     }
1152     return c;
1153 }
1154
1155 template <typename T>
1156 vector<vector<T>>& operator*=(vector<vector<T>>& a, const vector<vector<T>>& b) {
1157     return a = a * b;
1158 }
1159
1160 template <typename T, typename U>
1161 vector<vector<T>> power(const vector<vector<T>>& a, const U& b) {

```

```

1161     assert(b >= 0);
1162     vector<U> binary;
1163     U bb = b;
1164     while (bb > 0) {
1165         binary.push_back(bb & 1);
1166         bb >>= 1;
1167     }
1168     vector<vector<T>> res(a.size(), vector<T>(a.size()));
1169     for (int i = 0; i < static_cast<int>(a.size()); i++) {
1170         res[i][i] = 1;
1171     }
1172     for (int j = (int) binary.size() - 1; j >= 0; j--) {
1173         res *= res;
1174         if (binary[j] == 1) {
1175             res *= a;
1176         }
1177     }
1178     return res;
1179 }
1180
1181
1182 ===== mint_binpow.cpp =====
1183 const int mod = 1e9 + 7;
1184
1185 int add(int a, int b)
1186 {
1187     a %= mod; b %= mod;
1188     int res = a + b;
1189     if (res >= mod) res -= mod;
1190     return res;
1191 }
1192
1193 int sub(int a, int b)
1194 {
1195     a %= mod; b %= mod;
1196     int res = a - b;
1197     if (res < 0) res += mod;
1198     return res;
1199 }
1200
1201 int mult(int a, int b)
1202 {
1203     return 1LL * (a % mod) * (b % mod) % mod;
1204 }
1205
1206 int power(int a, int b, int m = mod)
1207 {
1208     a %= m;
1209     long long res = 1;
1210     while (b > 0)
1211     {
1212         if (b & 1) res = res * a % m;
1213         a = 1LL * a * a % m;
1214         b >>= 1;
1215     }
1216     return res;
1217 }
1218
1219 int div(int a, int b)
1220 {
1221     return mult(a, power(b, mod - 2, mod));
1222 }
1223
1224
1225

```

```

1226 ===== offline_lca.cpp =====
1227 vector<int> offline_lca_chan(int n, int r, const vector<vector<int>> &adj, vector<pair<int
    , int>> query)
1228 {
1229     //dsu
1230     // O(n+m) pre, O(1) query
1231     vector<int> par(n), siz(n, 1);
1232     iota(par.begin(), par.end(), 0);
1233     auto get = [&](int u, auto &&get) -> int
1234     {
1235         return (par[u] == u ? u : par[u] = get(par[u], get));
1236     };
1237     auto unite = [&](int u, int v) -> void
1238     {
1239         u = get(u, get), v = get(v, get);
1240         if(u == v)
1241             return;
1242         if(siz[u] < siz[v])
1243             swap(u, v);
1244         par[v] = u, siz[u] += siz[v];
1245     };
1246
1247     assert(!query.empty());
1248     int m = query.size();
1249
1250     vector<int> ans(m);
1251     vector<bool> see(m);
1252     vector<vector<int>> store(n);
1253
1254     for(int i = 0; i < m; i++)
1255     {
1256         auto [u, v] = query[i];
1257         store[u].push_back(i), store[v].push_back(i);
1258     }
1259
1260     auto dfs = [&](int u, int p, auto &&dfs) -> void
1261     {
1262         for(auto i : store[u])
1263         {
1264             if(see[i])
1265                 ans[i] = get(query[i].first == u ? query[i].second : query[i].first, get);
1266             see[i] = true;
1267         }
1268
1269         for(auto v : adj[u])
1270             if(v != p)
1271                 dfs(v, u, dfs);
1272
1273         if(p != 0)
1274             unite(u, p);
1275     };
1276     dfs(r, 0, dfs);
1277
1278     return ans;
1279 };
1280
1281
1282 ===== ordered_set.cpp =====
1283 // #include<bits/extc++.h>
1284 #include <ext/pb_ds/assoc_container.hpp>
1285 #include <ext/pb_ds/tree_policy.hpp>
1286 using namespace __gnu_pbds;
1287
1288 template <typename K, typename V, typename Comp = std::less<K>>

```

```

1290 using ordered_map = __gnu_pbds::tree<
1291     K, V, Comp,
1292     __gnu_pbds::rb_tree_tag,
1293     __gnu_pbds::tree_order_statistics_node_update
1294 >;
1295
1296 template<class T> using
1297 ordered_multiset = tree<T, null_type, less_equal<T>, rb_tree_tag,
1298     tree_order_statistics_node_update>;
1299 template <typename K, typename Comp = std::less<K>>
1300 using ordered_set = ordered_map<K, __gnu_pbds::null_type, Comp>;
1301
1302 // Supports
1303 // auto iterator = ordered_set().find_by_order(idx); // (0-indexed)
1304 // int num_strictly_smaller = ordered_set().order_of_key(key);
1305
1306
1307 ===== phi.cpp =====
1308 int phi(int n) { // 0(sqrt(n))
1309     int result = n;
1310     for (int i = 2; i * i <= n; i++) {
1311         if (n % i == 0) {
1312             while (n % i == 0) n /= i;
1313             result -= result / i;
1314         }
1315     }
1316     if (n > 1) result -= result / n;
1317     return result;
1318 }
1319 vector<int> phi; // phi[1] is 1
1320 void phi_1_to_n(int n) { // 1 to n in O(nloglogn)
1321     phi.resize(n + 1);
1322     for (int i = 0; i <= n; i++) phi[i] = i;
1323     for (int i = 2; i <= n; i++) {
1324         if (phi[i] == i) {
1325             for (int j = i; j <= n; j += i) phi[j] -= phi[j] / i;
1326         }
1327     }
1328 }
1329
1330
1331 ===== pragma.cpp =====
1332 #include <bits/allocator.h>
1333 #pragma GCC optimize("O3,unroll-loops")
1334 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
1335
1336
1337 ===== rng.cpp =====
1338 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
1339
1340
1341
1342 ===== scc.cpp =====
1343 class condenser
1344 {
1345     /*
1346     https://github.com/welcome-to-the-sunny-side/libra/
1347
1348     tc: O(n + m)
1349     ml: O(n + m)
1350
1351     info:
1352         0-indexed
1353         SCC u occurs before SCC v (u < v) in some topological ordering of SCCs

```

```

1354 warning:
1355     there may be multiple edges between different SCCs
1356
1357
1358 var:
1359
1360     [n -> number of nodes]
1361     [c -> number of SCCs]
1362     [adj] -> G
1363     [rdj] -> G.T
1364     [comp[u] -> component number of node u]
1365     [scc[u] -> outedge list for scc u]
1366     [grp[u] -> node list for scc u]
1367
1368 */
1369 public:
1370     int n, c;
1371     vector<bool> vis;
1372     vector<int> stak, comp;
1373     vector<vector<int>> adj, rdj, scc, grp;
1374
1375     condenser(int n, const vector<vector<int>> &adj) :
1376     n(n), c(0), vis(n), adj(adj), rdj(n), scc(n), grp(n), comp(n, -1)
1377     {
1378         for(int u = 0; u < n; u++)
1379             for(auto v : adj[u])
1380                 rdj[v].push_back(u);
1381     };
1382
1383     void condense()
1384     {
1385         auto dfs1 = [&](int u, auto &&dfs1) -> void
1386         {
1387             vis[u] = true;
1388             for(auto v : adj[u])
1389                 if(!vis[v])
1390                     dfs1(v, dfs1);
1391             stak.push_back(u);
1392         };
1393         for(int u = 0; u < n; u++)
1394             if(!vis[u])
1395                 dfs1(u, dfs1);
1396
1397         auto dfs2 = [&](int u, auto &&dfs2) -> void
1398         {
1399             comp[u] = c;
1400             for(auto v : rdj[u])
1401                 if(comp[v] == -1)
1402                     dfs2(v, dfs2);
1403         };
1404
1405         reverse(stak.begin(), stak.end());
1406         for(auto u : stak)
1407             if(comp[u] == -1)
1408                 dfs2(u, dfs2), ++ c;
1409
1410         for(int u = 0; u < n; u++)
1411             for(auto v : adj[u])
1412                 if(comp[u] != comp[v])
1413                     scc[comp[u]].push_back(comp[v]);
1414
1415         for(int u = 0; u < n; u++)
1416             grp[comp[u]].push_back(u);
1417     }
1418
1419     void fix() //remove multiple edges [O(m log(m))]

```

```

1419 {
1420     for(auto &v : scc)
1421     {
1422         sort(v.begin(), v.end());
1423         v.erase(unique(v.begin(), v.end()), v.end());
1424     }
1425 }
1426 };
1427
1428
1429 ===== segmented_sieve.cpp =====
1430 void segmented_sieve()
1431 {
1432
1433     int l,r;
1434     cin>>l>>r;
1435
1436     int lim = sqrt(r)+1;
1437
1438     vi base_primes;
1439     vector<bool> is_prime(lim+1,1);
1440     for(int i=2;i*i<=r;i++){
1441         if(!is_prime[i]) continue;
1442         for(int j = i*i;j<=lim;j+=i) {
1443             // dbg(j);
1444             is_prime[j] = false;
1445         }
1446     }
1447
1448     for(int i = 2;i*i<=r;i++) if(is_prime[i]) base_primes.pb(i);
1449
1450     vi is_prime_seg(r-l+1,1);
1451     for(int p : base_primes){
1452         int st = ((l+p-1)/p) * p;
1453         for(int j = st;j<=r;j+=p) is_prime_seg[j-l] = p;
1454     }
1455     if(l==1) is_prime_seg[0] = false;
1456     // is_prime_seg[i] = true ==> l+i is prime
1457 }
1458
1459
1460
1461
1462 ===== simple_segtree.cpp =====
1463 template <class S, auto op, auto e, class F, auto mapping>
1464 struct simple_segtree
1465 {
1466     int _n;
1467     int size;
1468     std::vector<S> d;
1469     simple_segtree() : simple_segtree(0) {}
1470     explicit simple_segtree(int n) { init(n); }
1471     explicit simple_segtree(const std::vector<S> &v)
1472     {
1473         _n = (int)v.size();
1474         size = 1;
1475         while (size < _n) size <= 1;
1476         d.assign(2 * size, e());
1477         for (int i = 0; i < _n; ++i) d[size + i] = v[i];
1478         for (int i = size - 1; i >= 1; --i) d[i] = op(d[2*i], d[2*i+1]);
1479     }
1480
1481     void init(int n)
1482     {
1483         _n = n;

```

```

1484     size = 1;
1485     while (size < _n) size <= 1;
1486     d.assign(2 * size, e());
1487 }
1488
1489 void set(int p, S x)
1490 {
1491     assert(0 <= p && p < _n);
1492     int pos = size + p;
1493     d[pos] = x;
1494     pos >>= 1;
1495     while (pos >= 1)
1496     {
1497         d[pos] = op(d[2*pos], d[2*pos+1]);
1498         pos >>= 1;
1499     }
1500 }
1501
1502 S get(int p)
1503 {
1504     assert(0 <= p && p < _n);
1505     return d[size + p];
1506 }
1507
1508 // prod on [l, r)
1509 S prod(int l, int r)
1510 {
1511     assert(0 <= l && l <= r && r <= _n);
1512     if (l == r) return e();
1513     l += size; r += size;
1514     S sml = e(), smr = e();
1515     while (l < r)
1516     {
1517         if (l & 1) sml = op(sml, d[l++]);
1518         if (r & 1) smr = op(d[--r], smr);
1519         l >>= 1; r >>= 1;
1520     }
1521     return op(sml, smr);
1522 }
1523
1524 // apply range [l, r) with mapping(f, s) done as point-wise updates (O(range * (log n)
1525 //    ))
1526 void apply(int l, int r, F f)
1527 {
1528     assert(0 <= l && l <= r && r <= _n);
1529     if (l == r) return;
1530     for (int i = l; i < r; ++i)
1531     {
1532         S cur = get(i);
1533         S nxt = mapping(f, cur);
1534         set(i, nxt);
1535     }
1536 }
1537
1538 template <bool (*g)(S)>
1539 int max_right(int l)
1540 {
1541     return max_right(l, [](S x){ return g(x); });
1542 }
1543
1544 template <class G>
1545 int max_right(int l, G g)
1546 {
1547     assert(0 <= l && l <= _n);
1548     assert(g(e()));

```



```

1548     if (l == _n) return _n;
1549     int idx = l + size;
1550     S sm = e();
1551     do
1552     {
1553         while ((idx & 1) == 0) idx >>= 1;
1554         if (!g(op(sm, d[idx])))
1555         {
1556             while (idx < size)
1557             {
1558                 idx <<= 1;
1559                 if (g(op(sm, d[idx])))
1560                 {
1561                     sm = op(sm, d[idx]);
1562                     ++idx;
1563                 }
1564             }
1565             return idx - size;
1566         }
1567         sm = op(sm, d[idx]);
1568         ++idx;
1569     } while ((idx & -idx) != idx);
1570     return _n;
1571 }
1572
1573 template <bool (*g)(S)>
1574 int min_left(int r)
1575 {
1576     return min_left(r, [](S x){ return g(x); });
1577 }
1578
1579 template <class G>
1580 int min_left(int r, G g)
1581 {
1582     assert(0 <= r && r <= _n);
1583     assert(g(e()));
1584     if (r == 0) return 0;
1585     int idx = r + size;
1586     S sm = e();
1587     do
1588     {
1589         --idx;
1590         while (idx > 1 && (idx & 1))
1591             idx >>= 1;
1592         if (!g(op(d[idx], sm)))
1593         {
1594             while (idx < size)
1595             {
1596                 idx = idx * 2 + 1;
1597                 if (g(op(d[idx], sm)))
1598                 {
1599                     sm = op(d[idx], sm);
1600                     --idx;
1601                 }
1602             }
1603             return idx + 1 - size;
1604         }
1605         sm = op(d[idx], sm);
1606     } while ((idx & -idx) != idx);
1607     return 0;
1608 }
1609 };
1610
1611 /** 0 based indexing
1612 /** [l,r)

```

```

1613 /** int max_right(int l, G g) - max r s.t g(l...r-1) is true and g(l...r) is false; true
      for e()
1614 /** int min_left(int r, G g) - min l s.t g(l...r-1) is true and g(l-1....r-1) is false;
      true for e()
1615 /** monoid - op is associative and e() is identity
1616
1617
1618
1619 ===== sparse_table.cpp =====
1620 class sparse_table
1621 {
1622     vector<vector<int>>> st;
1623     vector<int> log;
1624     int func(int a, int b)
1625     {
1626         return min(a, b);
1627     }
1628
1629 public:
1630     sparse_table(vector<int> &arr)
1631     {
1632         int n = arr.size();
1633         log.resize(n + 1);
1634         for (int i = 2; i <= n; ++i)
1635         {
1636             log[i] = log[i / 2] + 1;
1637         }
1638         int k = log[n] + 1;
1639         st.assign(n, vector<int>(k));
1640         for (int i = 0; i < n; ++i)
1641         {
1642             st[i][0] = arr[i];
1643         }
1644         for (int j = 1; (1 << j) <= n; ++j)
1645         {
1646             for (int i = 0; i + (1 << j) - 1 < n; ++i)
1647             {
1648                 st[i][j] = func(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
1649             }
1650         }
1651     }
1652
1653     int query(int L, int R) // 0 based indexing [l,r]
1654     {
1655         int j = log[R - L + 1];
1656         return func(st[L][j], st[R - (1 << j) + 1][j]);
1657     }
1658 };
1659
1660
1661
1662 ===== string_hash.cpp =====
1663 #include <bits/stdc++.h>
1664 using namespace std;
1665 #define int long long
1666 #define INF (int)1e18
1667
1668 mt19937_64 RNG(chrono::steady_clock::now().time_since_epoch().count());
1669
1670 struct Hash{
1671     int b, n; // b = number of hashes
1672     const int mod = 1e9 + 7;
1673     vector<vector<int>>> fw, bc, pb, ib;
1674     vector<int> bases;
1675

```

```

1676 inline int power(int x, int y){
1677     if (y == 0){
1678         return 1;
1679     }
1680
1681     int v = power(x, y / 2);
1682     v = 1LL * v * v % mod;
1683     if (y & 1) return 1LL * v * x % mod;
1684     else return v;
1685 }
1686
1687 inline void init(int nn, int bb, string str)//nn=size,bb=number of mods, str=string
1688 {
1689     n = nn;
1690     b = bb;
1691     fw = vector<vector<int>>(b, vector<int>(n + 2, 0));
1692     bc = vector<vector<int>>(b, vector<int>(n + 2, 0));
1693     pb = vector<vector<int>>(b, vector<int>(n + 2, 1));
1694     ib = vector<vector<int>>(b, vector<int>(n + 2, 1));
1695     bases = vector<int>(b);
1696     str = "0" + str;
1697
1698     for (auto &x : bases) x = RNG() % (mod / 10);
1699
1700     for (int i = 0; i < b; i++){
1701         for (int j = 1; j <= n + 1; j++){
1702             pb[i][j] = 1LL * pb[i][j - 1] * bases[i] % mod;
1703         }
1704         ib[i][n + 1] = power(pb[i][n + 1], mod - 2);
1705         for (int j = n; j >= 1; j--){
1706             ib[i][j] = 1LL * ib[i][j + 1] * bases[i] % mod;
1707         }
1708
1709         for (int j = 1; j <= n; j++){
1710             fw[i][j] = (fw[i][j - 1] + 1LL * (str[j] - 'a' + 1) * pb[i][j]) % mod;
1711         }
1712         for (int j = n; j >= 1; j--){
1713             bc[i][j] = (bc[i][j + 1] + 1LL * (str[j] - 'a' + 1) * pb[i][n + 1 - j]) %
1714                 mod;
1715         }
1716     }
1717
1718 inline int getfwhash(int l, int r, int i){ //[l,r] in 1 based indexing
1719     int ans = fw[i][r] - fw[i][l - 1];
1720     ans = 1LL * ans * ib[i][l - 1] % mod;
1721
1722     if (ans < 0) ans += mod;
1723
1724     return ans;
1725 }
1726
1727 inline int getbchash(int l, int r, int i){ //[l,r] in 1 based indexing
1728     int ans = bc[i][l] - bc[i][r + 1];
1729     ans = 1LL * ans * ib[i][n - r] % mod;
1730
1731     if (ans < 0) ans += mod;
1732
1733     return ans;
1734 }
1735
1736 inline bool equal(int l1, int r1, int l2, int r2){ //checks if s[l1,r1]==s[l2,r2] in
1737     one based indexing
1738     for (int i = 0; i < b; i++){
1739         int v1, v2;

```

```

1739         if (l1 <= r1) v1 = getfwhash(l1, r1, i);
1740         else v1 = getbchash(r1, l1, i);
1741
1742         if (l2 <= r2) v2 = getfwhash(l2, r2, i);
1743         else v2 = getbchash(r2, l2, i);
1744
1745         if (v1 != v2) return false;
1746     }
1747     return true;
1748 }
1749
1750 inline bool pal(int l, int r){ //checks if a substring [l,r] is pallindrome
1751     return equal(l, r, r, l);
1752 }
1753 };
1754
1755 int32_t main()
1756 {
1757     string s;
1758     cin>>s;
1759     int n=s.length();
1760     Hash h;
1761     h.init(n,2,s);
1762     cout<<h.getfwhash(1,2,1)<<endl;
1763     cout<<h.getbchash(3,4,1)<<endl;
1764 }
1765
1766
1767 ===== trie.cpp =====
1768 class Trie
1769 {
1770 public:
1771     static constexpr int ALPHA = 26;
1772     vector<array<int,ALPHA>> nxt;
1773     vector<int> cnt, pref;
1774     int nodes;
1775
1776     Trie(int mx = 1e6+1)
1777     {
1778         nxt.reserve(mx);
1779         cnt.reserve(mx);
1780         pref.reserve(mx);
1781         newNode(); // create root
1782     }
1783
1784     int newNode()
1785     {
1786         nxt.push_back({});
1787         cnt.push_back(0);
1788         pref.push_back(0);
1789         return nodes++;
1790     }
1791
1792     // insert s with frequency f (use f=-1 to remove)
1793     void insert(const string &s, int f = 1)
1794     {
1795         int u = 0;
1796         for (char ch : s) {
1797             int c = ch - 'a';
1798             if (!nxt[u][c]) {
1799                 nxt[u][c] = newNode();
1800             }
1801             u = nxt[u][c];
1802             pref[u] += f;
1803         }

```

```

1804         cnt[u] += f;
1805     }
1806
1807     // exact string count
1808     int count(const string &s) const
1809     {
1810         int u = 0;
1811         for (char ch : s)
1812         {
1813             int c = ch - 'a';
1814             if (!nxt[u][c] || pref[nxt[u][c]] <= 0)
1815                 return 0;
1816             u = nxt[u][c];
1817         }
1818         return cnt[u];
1819     }
1820
1821     // count of strings with prefix s
1822     int prefCount(const string &s) const
1823     {
1824         int u = 0;
1825         for (char ch : s) {
1826             int c = ch - 'a';
1827             if (!nxt[u][c] || pref[nxt[u][c]] <= 0)
1828                 return 0;
1829             u = nxt[u][c];
1830         }
1831         return pref[u];
1832     }
1833
1834     // sum of prefCounts along path matching s
1835     int query(const string &s) const
1836     {
1837         // depends on use case
1838
1839         // int u = 0;
1840         // int sum = 0;
1841         // for (char ch : s) {
1842         //     int c = ch - 'a';
1843         //     if (!nxt[u][c]) break;
1844         //     u = nxt[u][c];
1845         //     sum += pref[u];
1846         // }
1847         // return sum;
1848     }
1849
1850     // remove s (decrement counts)
1851     void remove(const string &s)
1852     {
1853         insert(s, -1);
1854     }
1855 };
1856
1857
1858 ===== debug.hpp =====
1859 #include <bits/stdc++.h>
1860 using namespace std;
1861 void __print(long long x) {cerr << x;}
1862 void __print(int64_t x) {cerr << x;}
1863 void __print(int32_t x) {cerr << x;}
1864 void __print(unsigned x) {cerr << x;}
1865 void __print(unsigned long x) {cerr << x;}
1866 void __print(unsigned long long x) {cerr << x;}
1867 void __print(float x) {cerr << x;}
1868 void __print(double x) {cerr << x;}

```

```

1869 void __print(long double x) {cerr << x;}
1870 void __print(char x) {cerr << '\'' << x << '\'';}
1871 void __print(const char *x) {cerr << '\"' << x << '\"'}
1872 void __print(const string &x) {cerr << '\"' << x << '\"'}
1873 void __print(bool x) {cerr << (x ? "true" : "false");}
1874
1875 template<typename T, typename V>
1876 void __print(const pair<T, V> &x) {cerr << '{'; __print(x.first); cerr << ','; __print(x.
    second); cerr << '}'}
1877 template<typename T>
1878 void __print(const T &x) {int f = 0; cerr << '{'; for (auto &i: x) cerr << (f++ ? ", " : ""
    ), __print(i); cerr << "}";}
1879 void _print() {cerr << "]\n";}
1880 template <typename T, typename... V>
1881 void _print(T t, V... v) {__print(t); if (sizeof...(v)) cerr << ", "; _print(v...);}
1882
1883
1884 ===== segtree_beats.cpp =====
1885
1886 #include <algorithm>
1887 #include <cassert>
1888 #include <cstdint>
1889 #include <iostream>
1890 #include <vector>
1891 #include <cstdio>
1892 #include <cstdint>
1893
1894 #define REP(i, n) for (int i = 0; (i) < (int)(n); ++ (i))
1895 #define REP3(i, m, n) for (int i = (m); (i) < (int)(n); ++ (i))
1896 #define REP_R(i, n) for (int i = (int)(n) - 1; (i) >= 0; -- (i))
1897 #define REP3R(i, m, n) for (int i = (int)(n) - 1; (i) >= (int)(m); -- (i))
1898 #define ALL(x) std::begin(x), std::end(x)
1899
1900 class segment_tree_beats {
1901     // MEMO: values for queries (max, min, lazy_add, and lazy_update) already apply to the
        current node; but not for children
1902     typedef struct {
1903         int64_t max;
1904         int64_t max_second;
1905         int max_count;
1906         int64_t min;
1907         int64_t min_second;
1908         int min_count;
1909         int64_t lazy_add;
1910         int64_t lazy_update;
1911         int64_t sum;
1912     } value_type;
1913
1914     int n;
1915     std::vector<value_type> a;
1916
1917 public:
1918     segment_tree_beats() = default;
1919     template <class InputIterator>
1920     segment_tree_beats(InputIterator first, InputIterator last) {
1921         int n_ = std::distance(first, last);
1922         n = 1; while (n < n_) n *= 2;
1923         a.resize(2 * n - 1);
1924         REP (i, n_) {
1925             tag<UPDATE>(n - 1 + i, *(first + i));
1926         }
1927         REP3 (i, n_, n) {
1928             tag<UPDATE>(n - 1 + i, 0);
1929         }
1930         REP_R (i, n - 1) {

```

```

1931         update(i);
1932     }
1933 }
1934
1935 void range_chmin(int l, int r, int64_t value) { // 0-based, [l, r)
1936     assert (0 <= l and l <= r and r <= n);
1937     range_apply<CHMIN>(0, 0, n, l, r, value);
1938 }
1939 void range_chmax(int l, int r, int64_t value) { // 0-based, [l, r)
1940     assert (0 <= l and l <= r and r <= n);
1941     range_apply<CHMAX>(0, 0, n, l, r, value);
1942 }
1943 void range_add(int l, int r, int64_t value) { // 0-based, [l, r)
1944     assert (0 <= l and l <= r and r <= n);
1945     range_apply<ADD>(0, 0, n, l, r, value);
1946 }
1947 void range_update(int l, int r, int64_t value) { // 0-based, [l, r)
1948     assert (0 <= l and l <= r and r <= n);
1949     range_apply<UPDATE>(0, 0, n, l, r, value);
1950 }
1951
1952 int64_t range_min(int l, int r) { // 0-based, [l, r)
1953     assert (0 <= l and l <= r and r <= n);
1954     return range_get<MIN>(0, 0, n, l, r);
1955 }
1956 int64_t range_max(int l, int r) { // 0-based, [l, r)
1957     assert (0 <= l and l <= r and r <= n);
1958     return range_get<MAX>(0, 0, n, l, r);
1959 }
1960 int64_t range_sum(int l, int r) { // 0-based, [l, r)
1961     assert (0 <= l and l <= r and r <= n);
1962     return range_get<SUM>(0, 0, n, l, r);
1963 }
1964
1965 private:
1966     static constexpr char CHMIN = 0;
1967     static constexpr char CHMAX = 1;
1968     static constexpr char ADD = 2;
1969     static constexpr char UPDATE = 3;
1970     static constexpr char MIN = 10;
1971     static constexpr char MAX = 11;
1972     static constexpr char SUM = 12;
1973
1974     template <char TYPE>
1975     void range_apply(int i, int il, int ir, int l, int r, int64_t g) {
1976         if (ir <= l or r <= il or break_condition<TYPE>(i, g)) {
1977             // break
1978         } else if (l <= il and ir <= r and tag_condition<TYPE>(i, g)) {
1979             tag<TYPE>(i, g);
1980         } else {
1981             pushdown(i);
1982             range_apply<TYPE>(2 * i + 1, il, (il + ir) / 2, l, r, g);
1983             range_apply<TYPE>(2 * i + 2, (il + ir) / 2, ir, l, r, g);
1984             update(i);
1985         }
1986     }
1987     template <char TYPE>
1988     inline bool break_condition(int i, int64_t g) {
1989         switch (TYPE) {
1990             case CHMIN: return a[i].max <= g;
1991             case CHMAX: return g <= a[i].min;
1992             case ADD: return false;
1993             case UPDATE: return false;
1994             default: assert (false);
1995         }

```

```

1996 }
1997 template <char TYPE>
1998 inline bool tag_condition(int i, int64_t g) {
1999     switch (TYPE) {
2000         case CHMIN: return a[i].max_second < g and g < a[i].max;
2001         case CHMAX: return a[i].min < g and g < a[i].min_second;
2002         case ADD: return true;
2003         case UPDATE: return true;
2004         default: assert (false);
2005     }
2006 }
2007 template <char TYPE>
2008 inline void tag(int i, int64_t g) {
2009     int length = n >> (32 - __builtin_clz(i + 1) - 1);
2010     if (TYPE == CHMIN) {
2011         if (a[i].max == a[i].min or g <= a[i].min) {
2012             tag<UPDATE>(i, g);
2013             return;
2014         }
2015         if (a[i].max != INT64_MIN) {
2016             a[i].sum -= a[i].max * a[i].max_count;
2017         }
2018         a[i].max = g;
2019         a[i].min_second = std::min(a[i].min_second, g);
2020         if (a[i].lazy_update != INT64_MAX) {
2021             a[i].lazy_update = std::min(a[i].lazy_update, g);
2022         }
2023         a[i].sum += g * a[i].max_count;
2024     } else if (TYPE == CHMAX) {
2025         if (a[i].max == a[i].min or a[i].max <= g) {
2026             tag<UPDATE>(i, g);
2027             return;
2028         }
2029         if (a[i].min != INT64_MAX) {
2030             a[i].sum -= a[i].min * a[i].min_count;
2031         }
2032         a[i].min = g;
2033         a[i].max_second = std::max(a[i].max_second, g);
2034         if (a[i].lazy_update != INT64_MAX) {
2035             a[i].lazy_update = std::max(a[i].lazy_update, g);
2036         }
2037         a[i].sum += g * a[i].min_count;
2038     } else if (TYPE == ADD) {
2039         if (a[i].max != INT64_MAX) {
2040             a[i].max += g;
2041         }
2042         if (a[i].max_second != INT64_MIN) {
2043             a[i].max_second += g;
2044         }
2045         if (a[i].min != INT64_MIN) {
2046             a[i].min += g;
2047         }
2048         if (a[i].min_second != INT64_MAX) {
2049             a[i].min_second += g;
2050         }
2051         a[i].lazy_add += g;
2052         if (a[i].lazy_update != INT64_MAX) {
2053             a[i].lazy_update += g;
2054         }
2055         a[i].sum += g * length;
2056     } else if (TYPE == UPDATE) {
2057         a[i].max = g;
2058         a[i].max_second = INT64_MIN;
2059         a[i].max_count = length;
2060         a[i].min = g;

```



```

2061         a[i].min_second = INT64_MAX;
2062         a[i].min_count = length;
2063         a[i].lazy_add = 0;
2064         a[i].lazy_update = INT64_MAX;
2065         a[i].sum = g * length;
2066     } else {
2067         assert (false);
2068     }
2069 }
2070 void pushdown(int i) {
2071     int l = 2 * i + 1;
2072     int r = 2 * i + 2;
2073     // update
2074     if (a[i].lazy_update != INT64_MAX) {
2075         tag<UPDATE>(l, a[i].lazy_update);
2076         tag<UPDATE>(r, a[i].lazy_update);
2077         a[i].lazy_update = INT64_MAX;
2078         return;
2079     }
2080     // add
2081     if (a[i].lazy_add != 0) {
2082         tag<ADD>(l, a[i].lazy_add);
2083         tag<ADD>(r, a[i].lazy_add);
2084         a[i].lazy_add = 0;
2085     }
2086     // chmin
2087     if (a[i].max < a[l].max) {
2088         tag<CHMIN>(l, a[i].max);
2089     }
2090     if (a[i].max < a[r].max) {
2091         tag<CHMIN>(r, a[i].max);
2092     }
2093     // chmax
2094     if (a[l].min < a[i].min) {
2095         tag<CHMAX>(l, a[i].min);
2096     }
2097     if (a[r].min < a[i].min) {
2098         tag<CHMAX>(r, a[i].min);
2099     }
2100 }
2101 void update(int i) {
2102     int l = 2 * i + 1;
2103     int r = 2 * i + 2;
2104     // chmin
2105     std::vector<int64_t> b { a[l].max, a[l].max_second, a[r].max, a[r].max_second };
2106     std::sort(b.rbegin(), b.rend());
2107     b.erase(std::unique(ALL(b)), b.end());
2108     a[i].max = b[0];
2109     a[i].max_second = b[1];
2110     a[i].max_count = (b[0] == a[l].max ? a[l].max_count : 0) + (b[0] == a[r].max ? a[r].max_count : 0);
2111     // chmax
2112     std::vector<int64_t> c { a[l].min, a[l].min_second, a[r].min, a[r].min_second };
2113     std::sort(ALL(c));
2114     c.erase(std::unique(ALL(c)), c.end());
2115     a[i].min = c[0];
2116     a[i].min_second = c[1];
2117     a[i].min_count = (c[0] == a[l].min ? a[l].min_count : 0) + (c[0] == a[r].min ? a[r].min_count : 0);
2118     // add
2119     a[i].lazy_add = 0;
2120     // update
2121     a[i].lazy_update = INT64_MAX;
2122     // sum
2123     a[i].sum = a[l].sum + a[r].sum;

```

```

2124 }
2125
2126 template <char TYPE>
2127 int64_t range_get(int i, int il, int ir, int l, int r) {
2128     if (ir <= l or r <= il) {
2129         return 0;
2130     } else if (l <= il and ir <= r) {
2131         // base
2132         switch (TYPE) {
2133             case MIN: return a[i].min;
2134             case MAX: return a[i].max;
2135             case SUM: return a[i].sum;
2136             default: assert (false);
2137         }
2138     } else {
2139         pushdown(i);
2140         int64_t value_l = range_get<TYPE>(2 * i + 1, il, (il + ir) / 2, l, r);
2141         int64_t value_r = range_get<TYPE>(2 * i + 2, (il + ir) / 2, ir, l, r);
2142         // mult
2143         switch (TYPE) {
2144             case MIN: return std::min(value_l, value_r);
2145             case MAX: return std::max(value_l, value_r);
2146             case SUM: return value_l + value_r;
2147             default: assert (false);
2148         }
2149     }
2150 }
2151 };
2152 // 0-based, [l, r) in logn
2153 int main() {
2154     int n, q; scanf("%d%d", &n, &q);
2155
2156     std::vector<long long> a(n);
2157     for (int i = 0; i < n; i++) {
2158         scanf("%lld", &a[i]);
2159     }
2160     segment_tree_beats beats(ALL(a));
2161
2162     for (int ph = 0; ph < q; ph++) {
2163         int ty, l, r; scanf("%d%d%d", &ty, &l, &r);
2164         if (ty == 0) {
2165             long long b; scanf("%lld", &b);
2166             beats.range_chmin(l, r, b);
2167         } else if (ty == 1) {
2168             long long b; scanf("%lld", &b);
2169             beats.range_chmax(l, r, b);
2170         } else if (ty == 2) {
2171             long long b; scanf("%lld", &b);
2172             beats.range_add(l, r, b);
2173         } else {
2174             long long sum = beats.range_sum(l, r);
2175             printf("%lld\n", sum);
2176         }
2177     }
2178     return 0;
2179 }

```

Listing 1: C++ Templates