/* Fenwick Tree */

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

/* Ordered Set */

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

/* Sparse Table && LCA */

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

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

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

```
        Node* node = root;
        for (int i = 0; i < prefix.length(); i++) {
            if (!node->containsKey(prefix[i])) {return false;}
            node = node->get(prefix[i]);
        }
        return true;
    }
    // return the count of the occurrences of the string word in the Trie
    int cntWord(string word) {
        Node* node = root;
        for (int i = 0; i < word.length(); i++) {
            if (!node->containsKey(word[i])) {return 0;}
            node = node->get(word[i]);
        }
        return node->endCount;
    }
    // return the count of words starting with the given prefix
    int cntPrefix(string word) {
        Node* node = root;
        // int res = 0;
        for (int i = 0; i < word.length(); i++) {
            if (!node->containsKey(word[i])) {return 0; // return res;}
            node = node->get(word[i]); // res += node->prefixCount;
        }
        return node->prefixCount; // return res;
    }
    // erase a word in the given trie
    void erase(string word) {
        Node* node = root;
        for (int i = 0; i < word.length(); i++) {
            node = node->get(word[i]);
            node->prefixCount--;
        }
        node->endCount--;
    }
};
/* Segment Tree */
template<typename S>
struct SegmentTree {
    int n;
    vector<S> tree;

    SegmentTree(int n) : n(n) {
        tree.assign(2 * n, e());
    }
```

```
    // identity element
    static S e() {
        return S();   // default identity (user should override by
specialization)
    }

    // merge operation
    static S op(const S& a, const S& b) {
        return a + b; // default: sum (user should override)
    }

    void build() {
        for (int i = n - 1; i > 0; --i)
            tree[i] = op(tree[i << 1], tree[i << 1 | 1]);
    }

    void modify(int p, S val) {
        for (tree[p += n] = val; p > 1; p >>= 1)
            tree[p >> 1] = op(tree[p], tree[p ^ 1]);
    }

    S query(int l, int r) {
        S resL = e(), resR = e();
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l & 1) resL = op(resL, tree[l++]);
            if (r & 1) resR = op(tree[--r], resR);
        }
        return op(resL, resR);
    }
};

/* Lazy Segment Tree */
template<typename S, typename L>
struct LazySegTree {
    int n, h;
    vector<S> t;
    vector<L> d;

    LazySegTree(int n) : n(n) {
        t.assign(2 * n, e());
        d.assign(n, id());
        h = sizeof(int) * 8 - __builtin_clz(n);
```

```
}

/* ---- user must define these ---- */

static S op(const S& a, const S& b);
static S e();

static S mapping(const L& f, const S& x);
static L composition(const L& f, const L& g);
static L id();

/* ---- internal logic ---- */

void build() {
    for (int i = n - 1; i > 0; --i)
        t[i] = op(t[i << 1], t[i << 1 | 1]);
}
void apply(int p, const L& v) {
    t[p] = mapping(v, t[p]);
    if (p < n) d[p] = composition(v, d[p]);
}
void build(int p) {
    while (p > 1) {
        p >>= 1;
        t[p] = op(t[p << 1], t[p << 1 | 1]);
        t[p] = mapping(d[p], t[p]);
    }
}
void push(int p) {
    for (int s = h; s > 0; --s) {
        int i = p >> s;
        if (d[i] != id()) {
            apply(i << 1, d[i]);
            apply(i << 1 | 1, d[i]);
            d[i] = id();
        }
    }
}
void range_apply(int l, int r, const L& v) {
    l += n;
    r += n;
    int l0 = l, r0 = r;
```

```
        for (; l < r; l >>= 1, r >>= 1) {
            if (l & 1) apply(l++, v);
            if (r & 1) apply(--r, v);
        }
        build(l0);
        build(r0 - 1);
    }
    S range_query(int l, int r) {
        l += n;
        r += n;
        push(l);
        push(r - 1);
        S resL = e(), resR = e();
        for (; l < r; l >>= 1, r >>= 1) {
            if (l & 1) resL = op(resL, t[l++]);
            if (r & 1) resR = op(t[--r], resR);
        }
        return op(resL, resR);
    }
};

/* Disjoint Set Union */
struct DSU {
 private:
    vector<int> par, sizes;

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

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

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

/* tarjan's algorithm for finding the bridges */
vector<int> dp(n, 0), tin(n), low(n);
vector<bool> vis(n, false);
vector<pair<int, int>> bridges;
int t = 0;
function<void(int, int)> dfs = [&](int u, int p) {
   vis[u] = true;
   dp[u] = 1;
   tin[u] = low[u] = ++t;
   for (auto& v : adj[u]) {
       if (v == p) continue;
       if (vis[v]) {
           low[u] = min(low[u], low[v]);
       } else {
           dfs(v, u);
           dp[u] += dp[v];
           low[u] = min(low[u], low[v]);
           if (low[v] > tin[u]) {
               bridges.push_back({u,v});
           }
       }
   }
```

```
};
dfs(0, -1);
/*
* articulation points (cut vertices)
* are vertices in an undirected graph whose removal increases the number of
connected components.
*/
vector<bool> vis(n, false);
vector<int> tin(n), low(n);
set<int> st;
int t = 0;
function<void(int, int)> dfs = [&](int u, int p) {
   int child = 0;
   vis[u] = true;
   tin[u] = low[u] = ++t;
   for (auto& v : adj[u]) {
       if (v == p) continue;
       if (vis[v]) {
           low[u] = min(low[u], tin[v]);
       } else {
           dfs(v, u);
           low[u] = min(low[u], low[v]);
           if (low[v] >= tin[u] && p != -1) st.insert(u);
           child++;
       }
   }
   if (p == -1 && child > 1) st.insert(u);
};

for (int i = 0; i < n; i++) {
   if (!vis[i]) dfs(i, -1);
}
/* binary uplifting */
   for (int i = 0; i < n; i++) {
       cin >> par[0][i], --par[0][i];
   }
   for (int j = 1; j < 30; j++)
       for (int i = 0; i < n; i++)
           par[j][i] = par[j - 1][par[j - 1][i]];

/* Matrix Exponentiation */
const int mod = 1e9 + 7;
```

```cpp
using matrix = vector<vector<int>>;
const matrix I = {
   {1, 0},
   {0, 1}
};
const matrix M = {
   {3, 2},
   {2, 2}
};

static matrix operator*(const matrix& A, const matrix& B) {
   const int m = A.size(), n = A[0].size(), p = B[0].size();
   matrix C(m, vector<int>(p, 0));
   for (int i = 0; i < m; i++) {
       for (int k = 0; k < n; k++) {
           for (int j = 0; j < p; j++) {
               C[i][j] = (C[i][j] + 1LL * A[i][k] * B[k][j]) % mod;
           }
       }
   }
   return C;
}
// Matrix exponentiation (LSBF)
static matrix pow(const matrix& M, int n) {
   matrix ans = I, P = M;
   for (; n > 0; n >>= 1) {
       if (n & 1)
           ans = ans * P;
       P = P * P;
   }
   return ans;
}
class Solution {
public:
   int numOfWays(int n) {
       if (n == 1) return 12;
       matrix A = pow(M, n - 1);
       return 6LL * (1LL * A[0][0] + A[0][1] + A[1][0] + A[1][1]) % mod;
   }
};
```