```
//========================================================================
//  PROJECT:            Sally's problem
//  FILE:               perms.cc
//  PURPOSE:            permutation and inversion utilities
//  UPDATE HISTORY:     version 1.0             3/15/10
//========================================================================

//----------------------------- Includes -------------------------------

#include <iostream>
#include <vector>
#include <list>
#include <iterator>
#include <algorithm>
#include <utility>
#include <cassert>

using namespace std;

//--------------------- Pair sort utilities ------------------------

bool operator== (const pair<int, int> & p1, const pair<int, int> & p2) {
        return (p1.first == p2.first) && (p1.second == p2.second) ||
                        (p1.first == p2.second) && (p1.second == p2.first);
}

bool operator< (const pair<int, int> & p1, const pair<int, int> & p2) {
        // First, sort the pairs
        pair<int, int> q1 = p1;
        pair<int, int> q2 = p2;
        if (q1.first > q1.second) {
                int temp = q1.first;
                q1.first = q1.second;
                q1.second = temp;
        }
        if (q2.first > q2.second) {
                int temp = q2.first;
                q2.first = q2.second;
                q2.second = temp;
        }
        // Now compare in lexicographic order.
        if (q1.first < q2.first) {
                return true;
        }
        if ((q1.first == q2.first) && (q1.second < q2.second)) {
                return true;
```

```
        }
        return false;
}

//================ Permutation CLASS DECLARATION ================

/*****************************************************************
        A permutation is represented internally as a vector of ints
        <p[0], p[1], ... , p[n - 1]>, where each p[i] is a unique
        integer in the range 0 .. n - 1. This is different from the
        usual representation starting from 1 and ending with n,
        but we all know that computer scientists start with zero.

        This class declares the Inversion class as a friend, for
        ease of implementation. Along with the usual constructors,
        we allow a permutation to be constructed from an inversion
        list.

        We supply ==, !=, and < as member functions for use in
        global sorting routines. In addition we supply a global
        output routine for printing permutations.
*****************************************************************/

class Permutation {
public:
        friend class Inversion;

        // Construct an n-element identity permutation
        Permutation(int n);

        // Construct a permutation from a vector of ints
        // NOTE: this doesn't check whether the vector is a permutation
        Permutation(vector<int> v) : place(v)
        { }

        // Copy ctor
        Permutation(const Permutation & p);

        // Construct a permutation from an inversion list
        Permutation(const Inversion & inv);

        int size() const {
                return place.size();
        }

        int at(int n) const {
```

```
                return place.at(n);
        }

        // RETURN the order of this permutation
        int ord() const;

        // RETURN true iff self and p are the same permutation
        bool operator== (const Permutation & p) const;

        // RETURN false iff self and p are the same permutation
        bool operator!= (const Permutation & p) const {
                return !(*this == p);
        }

        // RETURN true if self is lexicographically less than p
        // PRE: the perms are over the same base set
        bool operator< (const Permutation & p) const;

        // RETURN a copy of the product permutation self * p
        Permutation operator* (const Permutation & p) const;

        // RETURN the result of self applied to n
        int apply_to(int n) const {
                assert(n >= 0 && n < place.size());

                return place[n];
        }

        // RETURN the inverse of this permutation
        Permutation inverse();

private:
        vector<int> place;
};

//================== Eq_class CLASS DECLARATION ==================

/*****************************************************************
        An equivalence class object is a vector of Permutations. In
        this program (though it doesn't have to be), two permutations
        \pi and \sigma in S_n are equivalent if there is a permutation
        \tau such that \tau applied to the inversion list of \pi
        gives the inversion list of \sigma. For example, if
           \pi = (1 0 2 3) with inversion list (0, 1)
        then \tau = (1 2 0 3) applied to (0, 1) gives (1, 2), which
        is the inversion list of (0 2 1 3) so (1 0 2 3) \equiv (0 2 1 3).
```

```
        This condition also requires that the application of \tau
        yields a list that is ordered: the (i, j) in the transformed list
        must all have i < j or must all have i > j.

        As with Permutations, we supply ==, and < as member functions for
use in
        global sorting routines and supply a global output routine for
        printing permutations.
*****************************************************************/

class Eq_class {
public:
        Eq_class() : data()
        {}

        // PRE: the vector is sorted
        Eq_class(const vector<Permutation> & v) : data(v)
        {}

        // Put a new permutation into this equivalence class
        void append(const Permutation & p) {
                data.push_back(p);
        }

        // Merge two equivalence classes into one
        void join(const Eq_class & c);

        // RETURN the number of permutations in this equivalence class
        int size() const {
                return data.size();
        }

        // RETURN the i-th permutation in this equivalence class.
        Permutation at(int i) const {
                return data.at(i);
        }

        // PRE: both equivalence classes are sorted by the underlying
        // permutation orders.
        bool operator== (const Eq_class & ec) const;

        // PRE: both equivalence classes are sorted by the underlying
        // permutation orders.
        bool operator< (const Eq_class & ec) const;

private:
```

```
        vector<Permutation> data;
};

//=================== Inversion CLASS DECLARATION ===================

/*****************************************************************
        An Inversion object is a vector of pairs, representing the
        inversions of a given permutation.

        This class declares Permutation as a friend, so that we can
        construct an Inversion from its Permutation.
*****************************************************************/

class Inversion {
public:
        friend class Permutation;

        // Construct an empty list of inversions
        Inversion() : num(0), pairs()
        { }

        // Copy ctor
        Inversion(const Inversion & inv) : num(inv.num), pairs(inv.pairs)
        { }

        // Construct an inversion list from a permutation
        Inversion(const Permutation & p);

        bool operator== (const Inversion & inv) const;

        bool operator!= (const Inversion & inv) const {
                return !(*this == inv);
        }

        // RETURN the result of applying a perm to to this inversion list
        Inversion apply(const Permutation & p) const;

        // RETURN the inversion vector corresponding to this inversion list
        vector<int> inversion_vector() const;

        // RETURN true iff Sally's criterion is met, i.e., every inversion
        // pair is ordered (first < second) or (first > second)
        bool pair_ordered() const;

        // RETURN true iff this inversion is that of some permutation
        bool is_valid() const;
```

```
        // RETURN the number of pairs in this inversion
        int size() const {
                return pairs.size();
        }

        // RETURN the i-th pair in this inversion list
        pair<int, int> at(int i) const {
                return pairs.at(i);
        }

        // RETURN the size of the base set of this inversion list
        int num_elts() const {
                return num;
        }

private:
        int num;
        vector<pair<int, int> > pairs;

        Inversion restore_order() const;
};

//-------------- Permutation CLASS MEMBER FUNCTIONS ----------------

Permutation::Permutation(int n) : place(n) {

        for (int i = 0; i < n; i++) {
                place[i] = i;
        }
}

Permutation::Permutation(const Permutation & p) : place(p.size()) {

        for (int i = 0; i < p.size(); i++) {
                place[i] = p.place[i];
        }
}

// NOTE: not every list of pairs can result from some permutation
// (like (0, 2), (1, 3)). This should check that a list actually
// can be the list of a permutation.
Permutation::Permutation(const Inversion & inv) : place(inv.num_elts()) {
        // First, build the inversion vector
        vector<int> in = inv.inversion_vector();
```

```
        // Now use the inversion vector to build the permutation list
        list<int> p;
        p.push_back(in.size() - 1);
        list<int>::iterator it;
        for (int n = in.size() - 2; n >= 0; n--) {
                int count = 0;
                it = p.begin();
                while (count < in[n]) {
                        if (*it > n) {
                                count++;
                        }
                        it++;
                }
                p.insert(it, n);
        }

        // Finally, use the list to make the permutation vector
        int k = 0;
        for (it = p.begin(); it != p.end(); it++, k++) {
                place[k] = *it;
        }

}

int Permutation::ord() const {
        Permutation id(place.size());   // construct the identity
permutation
        Permutation q = *this;
        int i = 1;

        while (q != id) {
                i++;
                q = q * (*this);
        }
        return i;
}

bool Permutation::operator== (const Permutation & p) const {
        if (place.size() != p.size()) {
                return false;
        }
        for (int i = 0; i < place.size(); i++) {
                if (place[i] != p.place[i]) {
                        return false;
                }
        }
```

```
        return true;
}

bool Permutation::operator< (const Permutation & p) const {
        assert(place.size() == p.place.size());

        for (int i = 0; i < place.size(); i++) {
                if (place[i] < p.place[i]) {
                        return true;
                }
                if (place[i] > p.place[i]) {
                        return false;
                }
        }
        return false;
}

Permutation Permutation::operator* (const Permutation & p) const {
        assert(place.size() == p.size());
        vector<int> v(place.size());

        for (int i = 0; i < place.size(); i++) {
                v[i] = place[p.place[i]];
        }
        return Permutation(v);
}

Permutation Permutation::inverse() {
        vector<int> v(place.size());

        for (int i = 0; i < place.size(); i++) {
                v[place[i]] = i;
        }
        return Permutation(v);
}

//----------------- Eq_class CLASS MEMBER FUNCTIONS -----------------

void Eq_class::join(const Eq_class & c) {
        for (int i = 0; i < c.data.size(); i++) {
                data.push_back(c.data[i]);
        }
        sort(data.begin(), data.end());
        vector<Permutation>::iterator it = unique(data.begin(), data.end
());
        data.erase(it, data.end());
```

```
}

bool Eq_class::operator== (const Eq_class & ec) const {
        if (data.size() != ec.data.size()) {
                return false;
        }
        for (int i = 0; i < data.size(); i++) {
                if (data[i] != ec.data[i]) {
                        return false;
                }
        }
        return true;
}

bool Eq_class::operator< (const Eq_class & ec) const {
        if (data.size() < ec.data.size()) {
                return true;
        }
        if (data.size() > ec.data.size()) {
                return false;
        }
        for (int i = 0; i < data.size(); i++) {
                if (data[i] < ec.data[i]) {
                        return true;
                }
                if (ec.data[i] < data[i]) {
                        return false;
                }
        }
        return false;
}

//---------------- Inversion CLASS MEMBER FUNCTIONS ----------------

Inversion::Inversion(const Permutation & p) : num(p.size()), pairs() {

        for (int i = 0; i < num; i++) {
                for (int j = 0; j < i; j++) {
                        if (p.at(j) > p.at(i)) {
                                pairs.push_back(make_pair(p.at(i), p.at
(j)));
                        }
                }
        }
}
```

```
bool Inversion::operator== (const Inversion & inv) const {
	if (pairs.size() != inv.pairs.size()) {
		return false;
	}
	vector<pair<int, int> > v1 = pairs;
	vector<pair<int, int> > v2 = inv.pairs;
	sort(v1.begin(), v1.end());
	sort(v2.begin(), v2.end());
	for (int i = 0; i < pairs.size(); i++) {
		if (v1[i] != v2[i]) {
			return false;
		}
	}
	return true;
}

Inversion Inversion::apply(const Permutation & p) const {
	Inversion r = *this;

	for (int i = 0; i < r.size(); i++) {
		r.pairs[i].first = p.apply_to(pairs[i].first);
		r.pairs[i].second = p.apply_to(pairs[i].second);
	}
	return r;
}

vector<int> Inversion::inversion_vector() const {
	vector<int> v(num);

	for (int i = 0; i < num; i++) {
		v[i] = 0;
		for (int j = 0; j < pairs.size(); j++) {
			int min = pairs[j].first;
			if (pairs[j].second < min) {
				min = pairs[j].second;
			}
			if (min == i) {
				v[i]++;
			}
		}
	}
	return v;
}

bool Inversion::pair_ordered() const {
	if (pairs.size() == 0) {
```

```
                return true;
        }
        bool less = true;
        if (pairs[0].first > pairs[0].second) {
                less = false;
        }
        for (int i = 1; i < pairs.size(); i++) {
                if (((pairs[i].first < pairs[i].second) && !less) ||
                        ((pairs[i].first > pairs[i].second) && less)) {
                        return false;
                }
        }
        return true;
}

Inversion Inversion::restore_order() const {
        Inversion ret = *this;
        for (int i = 0; i < pairs.size(); i++) {
                if (ret.pairs[i].first > ret.pairs[i].second) {
                        swap(ret.pairs[i].first, ret.pairs[i].second);
                }
        }
        return ret;
}

// Uses Sally's result: a list of pairs is the inversion list of a
permutation
// iff for all i < j < k
// (1) The list is transitive: (i, j) and (i, k) in the list ==> (i, k) is
too
// (2) The median property: (i, k) is in ==> either (i, j) or (j, k) is
too.
bool Inversion::is_valid() const {
        Inversion r = restore_order();
        bool b1, b2;
        pair<int, int> p;

        // First, check for transitivity
        for (int i = 0; i < r.pairs.size(); i++) {
                b1 = b2 = true;
                for (int j = i + 1; j < r.pairs.size(); j++) {
                        if (r.pairs[i].second == r.pairs[j].first) {
                                // See if (r.pairs[i].first, r.pairs
[j].second) in r.pairs
                                p = make_pair(r.pairs[i].first, r.pairs
[j].second);
```

```
                                b1 = find(r.pairs.begin(), r.pairs.end(),
p) != r.pairs.end();
                        }
                        if (r.pairs[i].first == r.pairs[j].second) {
                                // See if (r.pairs[i].second, r.pairs
[j].first) in r.pairs
                                make_pair(r.pairs[i].second, r.pairs
[j].first);
                                b2 = find(r.pairs.begin(), r.pairs.end(),
p) != r.pairs.end();
                        }
                        if (!(b1 || b2)) {
                                return false;
                        }
                }
        }
        // Then check each pair for the median property
        for (int i = 0; i < r.pairs.size(); i++) {
                b1 = b2 = true;
                for (int j = r.pairs[i].first + 1; j < r.pairs[i].second; j
++) {
                        // See if (r.pairs[i].first, j) in ir.pairs
                        p = make_pair(r.pairs[i].first, j);
                        b1 = find(r.pairs.begin(), r.pairs.end(), p) !=
r.pairs.end();
                        // See if (j, r.pairs[i].second) in ir.pairs
                        p = make_pair(j, r.pairs[i].second);
                        b2 = find(r.pairs.begin(), r.pairs.end(), p) !=
r.pairs.end();
                        if (!(b1 || b2)) {
                                return false;
                        }
                }
        }
        return true;
}

//----------------------- GLOBAL UTILITIES -----------------------

// RETURN a vector of all possible permutations on {0, 1, ... , n - 1}
vector<Permutation> make_all_perms(int n) {
        vector<Permutation> pv;
        vector<int> v;
        for (int i = 0; i < n; i++) {
                v.push_back(i);
        }
```

```
        Permutation p(v);
        pv.push_back(p);

        while (next_permutation(v.begin(), v.end())) {
                Permutation p(v);
                pv.push_back(p);
        }
        return pv;
}

//-------------------------------------------------------------------------

// This changes from the internal representation of a perm on {0, ..., n-1}
// to the customary notation on {1, ..., n}
ostream & operator<< (ostream & os, const Permutation & p) {
        os << "(";
        for (int i = 0; i < p.size(); i++) {
                os << " " << p.at(i) + 1;
        }
        os << " )";
        return os;
}

ostream & operator<< (ostream & os, const Eq_class & ec) {
        os << "[";
        for (int i = 0; i < ec.size(); i++) {
                os << " " << ec.at(i);
        }
        os << " ]";
        return os;
}

// Ditto. See above
ostream & operator<< (ostream & os, const Inversion & inv) {
        for (int i = 0; i < inv.size(); i++) {
                os << "(" << inv.at(i).first+1 << ", " << inv.at(i).second
+1 << ") ";
        }
        return os;

}

// Ditto. See above
void print_pair_list(list<pair<int, int> > ls) {
        list<pair<int, int> >::iterator it;
```

```
        for (it = ls.begin(); it != ls.end(); it++) {
                cout << "(" << it->first + 1 << "," << it->second + 1 << ")
";
        }
        cout << endl;
}

//------------------------------------------------------------------------

list<pair<int, int> > make_node_list(const Inversion & inv) {
        list<pair<int, int> > ret;
        vector<int> count(inv.num_elts(), 0);

        for (int i = 0; i < inv.size(); i++) {
                count[inv.at(i).second]++;
        }
        for (int i = 0; i < inv.num_elts(); i++) {
                ret.push_back(make_pair(i, count[i]));
        }
        return ret;
}

// Utility function for make_class(). This takes a list of nodes and
// their in-degrees and returns the list that represents the
// digraph with node x removed.
list<pair<int, int> > trim(     int x,
                                                        list<pair<int, int>
> pl,
                                                        const Inversion &
inv) {
        // First, kill the (x, -) node from pl
        list<pair<int, int> >::iterator it;
        for (it = pl.begin(); it->first != x; it++)
        ;
        pl.erase(it);
        // Now decrement the in-degrees where needed
        for (int i = 0; i < inv.size(); i++) {
                if (inv.at(i).first == x) {
                        for (it = pl.begin(); it != pl.end(); it++) {
                                if (it->first == inv.at(i).second) {
                                        (it->second)--;
                                }
                        }
                }
        }
        return pl;
```

```
}

// RETURN the equivalence class of all permutations
// equivalent to the permutation of the inversion list using Sally's
// first criterion. This uses topological sorting on the DAG associated
// with the inversion list.
// It works, but does a lot of recursion that leads to duplicate results
// and it gives a lot of invalid results that don't get removed until
// the very end. In fact, this runs in time O(n!).
Eq_class make_class(const Inversion & inv,
                                        vector<int> perm,
                                        list<pair<int, int> > node_list) {
        Eq_class ret;
        if (node_list.empty()) {
                Permutation p(perm);
                p = p.inverse();
                Inversion in = inv.apply(p);
                if (in.is_valid()) {
                        Permutation q(in);
                        ret.append(q);
                }
                return ret;
        } else {
                for (list<pair<int, int> >::iterator it = node_list.begin
();
                                it != node_list.end();
                                it++) {
                        if (it->second == 0) {
                                vector<int> vv = perm;
                                vv.push_back(it->first);
                                ret.join(make_class(inv, vv,

trim(it->first, node_list, inv)));
                        }
                }
                return ret;
        }
}

// RETURN all permutations equivalent to the argument, using both
// the permutation and its inverse.
Eq_class find_all(Permutation p) {
        Inversion in1(p);
        vector<int> w;

        Eq_class eq1 = make_class(in1, w, make_node_list(in1));
```

```
        if (p != p.inverse()) {
                p = p.inverse();
                Inversion in2(p);
                Eq_class eq2 = make_class(in2, w, make_node_list(in2));
                eq1.join(eq2);
        }

        return eq1;
}

//------------------------------------------------------------------------

// RETURN a list of all permutations on {0, ... , n-1} with k inversions.
// Uses the Effler/Ruskey algorithm
list<Permutation> make_perm_list(int N, int n, int k, vector<int> v) {
        list<Permutation> ret;

        if ((n == 0) && (k == 0)) {
                Permutation p(v);
                ret.push_back(p);
                return ret;
        }
        int binom = (n < 3) ? 0 : ((n - 1) * (n - 2)) / 2;

        // Make a vector of permissible candidates
        vector<int> places;
        for (int i = 0; i < N; i++) {
                bool not_in_v = true;
                for (int j = n; j < N; j++) {
                        if (v[j] == i) {
                                not_in_v = false;
                        }
                }
                if (not_in_v) {
                        places.push_back(i);
                }
        }
        // For all legal x that haven't been used yet, make the rest of
        // the permutation recursively
        for (int i = 0; i < places.size(); i++) {
                if ((n - 1 - i <= k) && (k <= binom + n - 1 - i)) {
                        v[n - 1] = places[i];
                        list<Permutation> l = make_perm_list(N, n - 1, k -
n + 1 + i, v);
                        ret.splice(ret.end(), l);
```

```
                }
        }
        return ret;

}

int factorial(int n) {
        int r = 1;
        for (int i = 2; i <= n; i++) {
                r *= i;
        }
        return r;
}

// Build a permutation from std input, checking that it's legal
Permutation get_perm() {
        cout << "Enter perm, ending with 0: ";
        vector<int> v;
        int d;
        cin >> d;

        while (d > 0) {
                v.push_back(d - 1);
                cin >> d;
        }
        // Check that the input is valid. Inefficient, but it's unlikely
that
        // the user would enter a thousand-element permutation.
        bool legal_input = true;
        for (int i = 0; i < v.size(); i++) {
                bool found_it = false;
                for (int j = 0; j < v.size(); j++) {
                        if (i == v[j]) {
                                found_it = true;
                                break;
                        }
                }
                legal_input = legal_input && found_it;
        }
        assert(legal_input);

        Permutation p(v);
        return p;

}
```

```
//============================= MAIN ==============================

// This produces all equivalence classes, grouped by size
int main() {
        int base;
        cout << "N = ";
        cin >> base;
        int possible = factorial(base);

        vector<Permutation> perms = make_all_perms(base);
        vector<Eq_class> equivs;        // the vector of all equivalence
classes

        for (int i = 0; i < possible; i++) {
                Permutation p = perms[i];
                Inversion in(p);
                vector<Permutation> vp; // the perms in this equivalence
class
                for (int k = 0; k < possible; k++) {
                        Permutation q = perms[k];
                        Inversion ii = in.apply(q);
                        if (ii.is_valid() && ii.pair_ordered()) {
                                Permutation r(ii);
                                vp.push_back(r);
                        }
                }
                // Eliminate duplicates among the vector of permutations.
                sort(vp.begin(), vp.end());
                vector<Permutation>::iterator it = unique(vp.begin(),
vp.end());
                vp.erase(it, vp.end());
                Eq_class ec(vp);
                equivs.push_back(ec);
        }
        sort(equivs.begin(), equivs.end());
        vector<Eq_class>::iterator ite = unique(equivs.begin(), equivs.end
());
        equivs.erase(ite, equivs.end());

        // Look at all the equivalence classes.
        // This produces a lot of output. It can be commented out.
        for (int i = 0; i < equivs.size(); i++) {
                cout << Inversion(equivs[i].at(0)).size() << ": \t";
                cout << equivs[i] << endl;
        }
        cout << endl << endl;
```

```
	// Now we'll count the number of classes of each size
	vector<int> count(100, 0);
	for (int i = 0; i < equivs.size(); i++) {
		count[equivs[i].size()]++;
	}
	int total = 0;
	cout << "size\tcount" << endl;
	for (int i = 1; i < 100; i++) {
		if (count[i] != 0) {
			cout << " " << i << "\t  " << count[i] << endl;
			total += count[i];
		}
	}
	cout << endl << "Total classes = " << total << endl;

	return EXIT_SUCCESS;
}
```