Solution for Assignment 1

There are many possible ways of representing a permutation, the simplest is to use an array p so that $p[i]$ stores the value of the permutation at i. In this case, since the size of the permutation was at most 100, it could be done by just declaring an array of size 100. But this would waste memory in case there are many permutations of smaller size. A better way would be to store just a pointer and allocate an array of required size, whenever constructing a permutation. Whenever this is done, you must have a destructor also to destroy the allocated array, otherwise it leads to allocated memory that is not accessible. This is called memory leak and can quickly lead to memory running out. When using a pointer in a class, it is also necessary to define a copy constructor and assignment operator. The default meaning of these would just assign the pointer of one variable to the other, and not actually copy the values. So a possible declaration could be

```
class permutation
{
     int *p, n;
     public:
     permutation( int *q, int m)
     {
         p = new int[m];
         n = m;
         for (int i = 0; i < n; i++) p[i] = q[i];
     }
     ~permutation()
     {
         delete [] p;
     }
     permutation( permutation const &q)
     {
         n = q.n;
         p = new int[n];
         for (int i = 0; i < n; i++) p[i] = q.p[i];
     }
     permutation const & operator=( permutation const & q)
     {
         if (&q == this) return *this;
         delete [] p;
         n = q.n;
         p = new int[n];
         for (int i = 0; i < n; i++) p[i] = q.p[i];
         return *this;
     }
};
```

Note the difference between a copy constructor and the assignment operator. The first is used when a new variable of type permutation is created. The assignment is used to assign a value to an existing variable. So the initial array must be destroyed. A small catch here is that if an assignment like p = p is used, it would destroy the p array before the values are copied. This is avoided by the check whether q and p have the same address. This will not be done for checking the assignment.

An important point to note is that when a function returns an object of type permutation, or it is passed as a parameter, a copy constructor is used to create a copy of the parameter. This would involve copying all elements in the array. A way of avoiding this is to pass by reference. But to return values by reference, you cannot use local variables declared in the function. These are destroyed when the function terminates and a reference to them is no longer valid. This can be avoided by using a pointer to the variable in the function, and creating the variable using new outside the function. This would be valid even after the function terminates. In this case, there is no memory leak, because a reference to the created variable is returned and can be used in the calling program. A possible way of writing the inverse function would be

```
permutation const & operator-() const
{
     permutation *q = new permutation(*this);
//   creates a copy of the permutation.
     q->n = n;
     for (int i = 0; i < n; i++) q->p[p[i]] = i;
     return *q;
}

//to return an array

int* to_array() const
{
     int *q = new int[n];
     for (int i = 0; i < n; i++) q[i] = p[i];
     return q;
}
```

Composition can be done in the same way. The operators return permutations so that they can be used in expressions. A statement like $p = p * (-q)$ would be valid, for permutations p, q. For this assignment, it is okay if you have done it using an array of fixed size, and there is copying of elements involved.

The remaining part of the assignment depended on the cycle representation of permutations, which was encountered in some other problems discussed, and some material that you would have done in CS 207.

Given a permutation p, if we consider the sequence $i, p(i), p(p(i)) \ldots$ the sequence must eventually come back to i and cycle afterwards. Any permutation can be broken into such

cycles, such that every element is in exactly one cycle. If the length of the cycle containing i is l, then $p^k(i) = p^{k\%l}(i)$. So to find the permutation $p^k(i)$, just process each cycle independently. If $m = k \bmod l$ then starting with i first find $p^k(i) = p^m(i)$ then $p^k(p(i)) = p(p^k(i))$ and so on. This will only require going through each cycle twice. The time required is actually independent of k.

A permutation q which is the square of p has a special property in terms of the cycle representation. Every odd cycle in p will give rise to an odd cycle of the same length in q, while every even cycle in p will break into two smaller cycles of half the length. Every even cycle in q can only arise from the second possibility, so there must be a matching cycle of the same length. Thus q is the square of a permutation p iff for every even number $2k$, q has an even number of cycles of length $2k$. This property can be checked easily, and one possible permutation p can be obtained by combining cycles of the same even length into one cycle in p. Can you modify this to find the number of square roots of q?

The last part about finding the log, required the Chinese Remainder theorem that you have done in CS 207. Suppose there exists a k such that $p = q^k$. Considering the cycles in q, for every i, $p(i)$ must be in the same cycle of q as i. We can check if this is true by traversing the cycle of i in q. If it is true for i, then for all other elements j in the cycle, the values of $p(j)$ must just shift along the cycle in q. That is $p(q(i)) = q(p(i))$, $p(q(q(i))) = q(q(p(i)))$ and so on. This can be checked by traversing the cycle once. Assuming this is satisfied, we know what k must be modulo the length of the cycle. Do this for every cycle.

The final part is to find the smallest k that satisfies the required relations $k = r_i \bmod l_i$ where the l_i are the lengths of the cycles. When does this have a solution, and if so how to find it? The case when the l_i are relatively prime seems to have been done in CS 207. The general case follows the same arguments. Consider two cycles at a time, and take $k = r_1 \bmod l_1$ and $k = r_2 \bmod l_2$. A necessary condition for this to have a solution is $r_1 - r_2$ must be divisible by $g = gcd(l_1, l_2)$. This is easy to see because $k = q_1 l_1 + r_1$ and $k = q_2 l_2 + r_2$ implies $r_1 - r_2 = q_2 l_2 - q_1 l_1$ is divisible by g. The converse is also true. If we write $g = a l_1 + b l_2$, obtained by Euclid's algorithm, then $a * r_2 * l_1/g + b * r_1 * l_2/g = r_1 \bmod l_1$, and also $= r_2 \bmod l_2$. Moreover this solution is unique modulo $l_1 l_2/g$, so we can take it modulo that. Repeating this by taking one cycle at a time, we get the smallest value of k, if it exists, modulo the lcm of the l_i.

An issue was how large can this number be? Or given $l_1 + \cdots + l_k = 100$, how large can the lcm of these numbers be? There is no simple formula for this, but an estimate is $e^{\sqrt{n \ln n}}$. This function is called the Landau function. For 100, it is I guess maximized by taking the cycle lengths to be the primes 2,3,5,7,11,13,17,19,23. This number slightly exceeds 10^9, but all computations can be done comfortably using long long int. Only final answer could be printed modulo $10^9 + 7$, but in most cases, it will be the actual value. This was actually unnecessary, I thought numbers may overflow during computation.