2. The instantiation of Map<Coord, int>::insert calls the instantiation of Map<Coord, int>::doInsertOrUpdate, which calls the instantiation of Map<Coord, int>::find, which contains the expression p->m\_key != key, where both operands are Coord. We never defined operator!= for Coord operands.

3. b. Without any static or global variables or any additional containers, there would be no way to keep track of the path from the root to each node of the tree.

4. a. Consider the code in the k loop:

for (int k = 0; k < N; k++)

{

if (k == i || k == j)

continue;

if (hasCommunicatedWith[i][k] && hasCommunicatedWith[k][j])

numIntermediaries[i][j]++;

}

This involves one initialization (int k = 0), which we can ignore, since it's dominated by the N repetitions of everything else. The most work that each of the N iterations of the loop might do is a comparison (k < N), an increment (k++), two equality tests (k == i and k == j), an and test (&&), an increment (++), and 3 double subscriptings. These basic operations are each constant time, so this loop does no more than mN basic operations, for some constant m. This inner loop is O(N).

For the rest of this analysis, we won't be so meticulous: We'll drop low order terms where it won't affect the result.

The number of operations performed during one execution of the innermost loop body (which obviously accounts for most of the operations, since it's executed the most) is bounded by some constant m. In all, that body accounts for sum(i from 0 to N-1) of sum(j from 0 to N-1) of sum(k from 0 to N-1) of m operations.

sum(i from 0 to N-1) of {sum(j from 0 to N-1) of [sum(k from 0 to N-1) of m]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to N-1) of [sum(k from 0 to N-1) of 1]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to N-1) of N} ~  
m \* sum(i from 0 to N-1) of N\*N ~  
m \* N\*N\*N = O(N3)

4. b. Again, the innermost statement accounts for the most operations performed, and it accounts for sum(i from 0 to N-1) of sum(j from 0 to i-1) of sum(k from 0 to N-1) of m operations. We'll retain the constant of proportionality just to get a feel for how much faster this can be. (We assume m is about the same as Part a.)

sum(i from 0 to N-1) of {sum(j from 0 to i-1) of [sum(k from 0 to N-1) of m]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to i-1) of [sum(k from 0 to N-1) of 1]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to i-1) of N} ~  
m \* sum(i from 0 to N-1) of i\*N} ~  
m \* N \* sum(i from 0 to N-1) of i} ~  
m \* N \* ((N-1) \* N / 2)

Since ((N-1) \* N / 2) ~ N\*N/2, the full sum is about (m/2) \* N\*N\*N = O(N3). For large N, this algorithm is about twice as fast as the one in Part a, but it's still order N3; doubling the size of a problem increases the running time about eightfold.

5. First, notice that our implementation of the three-argument form of get takes a length of time proportional to the distance between the desired position and the nearest end of the list. For position k in a list of size N, this is min(k, N-k). insert, erase, and the two-argument form of get each call find, which can visit up to N nodes.

The size function is constant time, so the initial block of code to determine the bigger Map is constant time. The final swap is constant time as well. The call to the copy constructor just before the for loop visits N nodes, so is O(N). Now let's examine the loop.

The statements in the loop are executed N times. Each time, the first call to get function visits min(k,N-k) nodes (which is bounded by N). The get in the if condition visits up to N nodes, and each branch of the if (the insert or erase) visits up to N nodes. Thus, each loop iteration visit up to 3N nodes, which is O(N). Since the loop visits O(N) nodes on each of the N iterations, the loop overall visits O(N2) nodes. This dominates the O(N) and O(1) parts of the function, so the overall time complexity is O(N2).