**CS 32 Homework 4**

2. The call to Map<Coord, int>::insert causes a compilation error as it calls the find function. The find function’s implementation compares the current Map’s key with the key passed in the function arguments. When the KeyType is the Coord class, we get a compilation error because there is no definition or implementation for the comparison operator ‘!=’ for the Coord class. Thus, there is no way of knowing how to compare the Coord classes.

3b. We cannot solve the problem if we have only the one parameter listAll function and were to implement it recursively because there is no way to keep track of the base class when printing the inheritance path. If you had a string within the recursive function, each recursive call would reset the contents of the string. Since we cannot use global variables or variables declared with the keyword static there is no way to keep track of where in the hierarchy we are and thus printing a full inheritance path is not possible. Each iteration of the loop traversing through the vector leads to a subclass that is itself a base class in its inheritance hierarchy.

4a. Time Complexity: O(N3)

This is because we have a nested loop structure consisting of three for loops all having ‘N’ iterations. All the statements within and outside these for loops have a constant time complexity, O(1). Thus, the overall time complexity is O(N3).

const int N = *some value*; **//O(1)**

bool hasCommunicatedWith[N][N]; **//O(1)**

...

int numIntermediaries[N][N]; **//O(1)**

for (int i = 0; i < N; i++) **//O(N3)**

{ **//O(N2)**

numIntermediaries[i][i] = -1; // the concept of intermediary

// makes no sense in this case

for (int j = 0; j < N; j++) **//O(N2)**

{ **//O(N)**

if (i == j) **//O(1)**

continue;

numIntermediaries[i][j] = 0; **//O(1)**

for (int k = 0; k < N; k++) **//O(N)**

{ **// O(1)**

if (k == i || k == j) **//O(1)**

continue;

if (hasCommunicatedWith[i][k] && hasCommunicatedWith[k][j]) **// O(1)**

numIntermediaries[i][j]++; **//O(1)**

}

}

}

4b. Time Complexity: O(N3)

This is because we have a nested loop structure, two of which have ‘N’ iterations. The third loop has ‘i’ iterations. The control variable ‘i’ itself goes to ‘N’ eventually. Thus, at some point the program will have all three loops with ‘N’ iterations. Every statement within and outside these loops has a constant time complexity of O(1). Thus, the overall time complexity is O(N3).

const int N = *some value*; **//O(1)**

bool hasCommunicatedWith[N][N]; **//O(1)**

...

int numIntermediaries[N][N]; **//O(1)**

for (int i = 0; i < N; i++) **//O(N2i)**

{ **//O(Ni)**

numIntermediaries[i][i] = -1; // the concept of intermediary

// makes no sense in this case

for (int j = 0; j < **i**; j++) **// loop limit is now i, not N, 🡪 O(Ni)**

{ **//O(N)**

numIntermediaries[i][j] = 0; **//O(1)**

for (int k = 0; k < N; k++) **//O(N)**

{ **//O(1)**

if (k == i || k == j) **//O(1)**

continue;

if (hasCommunicatedWith[i][k] && hasCommunicatedWith[k][j]) **//O(1)**

numIntermediaries[i][j]++; **//O(1)**

}

**numIntermediaries[j][i] = numIntermediaries[i][j]; //O(1)**

}

}

5. Time Complexity: O(N2)

This is because the function calls functions erase, insert and get within a for loop that has N iterations. The functions erase and insert have complexities of O(N) whereas the function get has a time complexity of O(N/2). Thus, the overall complexity of the for loop is O(N2). i.e. the highest degree of N in the sum, N+ N/2 + 1 times N iterations.

Outside the loop, the copy constructor implementation has an O(N) complexity whereas every other statement has a constant time complexity, O(1). Thus, the overall complexity of the function is O(N2). i.e. the highest degree of N in the sum, N2 + N + 1.

bool combine(const Map& m1, const Map& m2, Map& result)

{ **//O(N2)**

// For better performance, the bigger map should be the basis for

// the result, and we should iterate over the elements of the

// smaller one, adjusting the result as required.

const Map\* bigger; **//O(1)**

const Map\* smaller; **//O(1)**

if (m1.size() >= m2.size())

{

bigger = &m1; **//O(1)**

smaller = &m2; **//O(1)**

}

else

{

bigger = &m2; **//O(1)**

smaller = &m1; **//O(1)**

}

// Guard against the case that result is an alias for m1 or m2

// (i.e., that result is a reference to the same map that m1 or m2

// refers to) by building the answer in a local variable res. When

// done, swap res with result; the old value of result (now in res) will

// be destroyed when res is destroyed.

bool status = true; **//O(1)**

Map res(\*bigger); // res starts as a copy of the bigger map, **O(N)**

for (int n = 0; n < smaller->size(); n++) // for each pair in smaller, **O(N2)**

{ **//O(N)**

KeyType k; **//O(1)**

ValueType vsmall; **//O(1)**

smaller->get(n, k, vsmall); **//Get has N/2 complexity**

ValueType vbig; **//O(1)**

if (!res.get(k, vbig)) // key in smaller doesn't appear in bigger

res.insert(k, vsmall); // so add it to res, **Insert has O(N) complexity**

else if (vbig != vsmall) // same key, different value,

{ // so pair shouldn't be in res

res.erase(k); **// Erase has O(N) worst case complexity**

status = false; **//O(1)**

}

}

result.swap(res); **//Swap has O(1) complexity**

return status; **//O(1)**

}