Jason Tieu

304047667

HW4

**2) Consider this program:**

1. #include "Map.h" // class template from problem 1
2. class Coord
3. {
4. public:
5. Coord(int r, int c) : m\_r(r), m\_c(c) {}
6. Coord() : m\_r(0), m\_c(0) {}
7. double r() const { return m\_r; }
8. double c() const { return m\_c; }
9. private:
10. double m\_r;
11. double m\_c;
12. };
13. int main()
14. {
15. Map<int, double> mid;
16. mid.insert(42, -1.25); // OK
17. Map<Coord, int> mpi;
18. mpi.insert(Coord(40,10), 32); // error!
19. }

***Explain in a sentence or two why the call to Map<Coord, int>::insert causes at least one compilation error. (Notice that the call to Map<int, double>::insert is fine.) Don't just transcribe a compiler error message; your answer must indicate you understand the reason.***

**Answer:** There is no defined way on how to check if the Coord object created currently exists inside the Map. We would have to overload the != operators to use Coord objects as arguments to allow the insert and find functions of Map to determine whether or not the Coord object can be inputted.

***3b) We introduced the two-parameter overload of listAll. Why could you not solve this problem given the constraints in part a if we had only a one-parameter listAll, and you had to implement*it*as the recursive function?***

**Answer:** It would be difficult to do a one-parameter listAll as a recursive function since the 2nd parameter (string path) is required to make the recursion call much easier. We wouldn't be able to continually pass through a string to each recursive loop since the only argument would be a Class pointer. Also, there would not be a defined way of displaying a full path since there would be no way to check what the current path is in each recursive loop.

**4a) A secretive government agency has reluctantly admitted that for N phone numbers, numbered 0 through N-1, it has a two-dimensional array of bool hasCommunicatedWith that records which phones have been in communication with others: hasCommunicatedWith[i][j] is true if and only if phone i and phone j have been in communication. If phone i has communicated with phone k, and phone k has communicated with phone j, we call phone k a direct intermediary between phone i and phone j.**

**The agency has an algorithm that, for every pair of phones i and j, determines how many direct intermediaries they have between them. Here's the code:**

const int N = some value;

bool hasCommunicatedWith[N][N];

...

int numIntermediaries[N][N];

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

{

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

// makes no sense in this case

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

{

if (i == j)

continue;

numIntermediaries[i][j] = 0;

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

{

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

continue;

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

numIntermediaries[i][j]++;

}

}

}

***What is the time complexity of this algorithm, in terms of the number of basic operations (e.g., additions, assignments, comparisons) performed: Is it O(N), O(N log N), or what? Why? (Note: In this homework, whenever we ask for the time complexity, we care only about the high order term, so don't give us answers like O(N2+4N).)***

**Answer:** The time complexity of this algorithm is O() because there are three nested for loops (so O(N) \* O(N) \* O(N) ) and the if statements are O(1) (constant time). Since we only care about the higher order term, it is O().

**The algorithm in part a doesn't take advantage of the symmetry of communication: for every pair of phones i and j, hasCommunicatedWith[i][j] == hasCommunicatedWith[j][i]. One can skip a lot of operations and compute the number of direct intermediaries more quickly with this algorithm:**

1. const int N = some value;
2. bool hasCommunicatedWith[N][N];
3. ...
4. int numIntermediaries[N][N];
5. for (int i = 0; i < N; i++)
6. {
7. numIntermediaries[i][i] = -1; // the concept of intermediary
8. // makes no sense in this case
9. for (int j = 0; j < **i**; j++) **// loop limit is now i, not N**
10. {
11. numIntermediaries[i][j] = 0;
12. for (int k = 0; k < N; k++)
13. {
14. if (k == i || k == j)
15. continue;
16. if (hasCommunicatedWith[i][k] && hasCommunicatedWith[k][j])
17. numIntermediaries[i][j]++;
18. }
19. **numIntermediaries[j][i] = numIntermediaries[i][j];**
20. }
21. }

***What is the time complexity of this algorithm? Why?***

**Answer:** The time complexity of this algorithm is O() since there are three nested for loops. The most inner for loop has time complexity O(N). The middle one has O() since the limit is changing with i. That would be lead to a sum of i = n(n+1)/2 => O(). The outermost for loop is O(N). Since they are nested, we multiply them to get O(N) \* O() \* O(N) = O().

**5) Here again is the non-member combine function for Maps from**[**Map.cpp**](http://cs.ucla.edu/classes/winter14/cs32/Homeworks/4/Map.cpp)**:**

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

{

// 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;

const Map\* smaller;

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

{

bigger = &m1;

smaller = &m2;

}

else

{

bigger = &m2;

smaller = &m1;

}

// 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;

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

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

{

KeyType k;

ValueType vsmall;

smaller->get(n, k, vsmall);

ValueType vbig;

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

res.insert(k, vsmall); // so add it to res

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

{ // so pair shouldn't be in res

res.erase(k);

status = false;

}

}

result.swap(res);

return status;

}

***Assume that m1, m2, and the old value of result each have N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its time complexity? Why?***

The time complexity would be O(). This is because the get() function has time complexity of O(1) if called with an index number, if not then it would be O(N). There is a for loop that is O(N), and inside are both versions of the get() function. Therefore it would be O(N) \* (O(1) + O(N)). Since we only want the higher order, we get O().