1e. Make sure you understand why the code below passes the first two tests but fails the third. Draw pictures if necessary. Don't forget the lesson you learn from this problem when working on Project 3.  
#include <iostream>

#include <vector>

#include <list>

using namespace std;

const int MAGIC = 11223344;

void test()

{

bool allValid = true;

vector<int> v1(5, MAGIC);

int k = 0;

for ( ; k != v1.size(); k++)

{

if (v1[k] != MAGIC)

{

cout << "v1[" << k << "] is " << v1[k] << ", not " << MAGIC <<"!" << endl;

allValid = false;

}

if (k == 2)

{

for (int i = 0; i < 5; i++)

v1.push\_back(MAGIC);

}

}

if (allValid && k == 10)

cout << "Passed test 1" << endl;

else

cout << "Failed test 1" << endl;

allValid = true;

list<int> l1(5, MAGIC);

k = 0;

for (list<int>::iterator p = l1.begin(); p != l1.end(); p++, k++)

{

if (\*p != MAGIC)

{

cout << "Item# " << k << " is " << \*p << ", not " << MAGIC <<"!" << endl;

allValid = false;

}

if (k == 2)

{

for (int i = 0; i < 5; i++)

l1.push\_back(MAGIC);

}

}

if (allValid && k == 10)

cout << "Passed test 2" << endl;

else

cout << "Failed test 2" << endl;

allValid = true;

vector<int> v2(5, MAGIC);

k = 0;

for (vector<int>::iterator p = v2.begin(); p != v2.end(); p++, k++)

{

if (k >= 20) // prevent infinite loop

break;

if (\*p != MAGIC)

{

cout << "Item# " << k << " is " << \*p << ", not " << MAGIC <<"!" << endl;

allValid = false;

}

if (k == 2)

{

for (int i = 0; i < 5; i++)

v2.push\_back(MAGIC);

}

}

if (allValid && k == 10)

cout << "Passed test 3" << endl;

else

cout << "Failed test 3" << endl;

}

int main()

{

test();

}

Explain in a sentence or two what happens during the execution of test case 3 that eventually leads to test case 3 failing.

Because you are inserting new values into the vector when k == 2 (v2.push\_back(MAGIC)), the iterator p becomes undefined which eventually leads to test case 3 failing since using p in any way will be undefined until you assign p a new value such as p = v2.begin() + 2;

p becomes undefined because v2 originally had 5 elements. The original vector will run out of space after we add additional elements, so it will be relocated to another address. The vector iterator is a pointer, so it would not be pointing to the correct location after this relocation,

3. Consider this program:  
 #include "Map.h" // class template from problem 2

class Coord

{

public:

Coord(double r, double c) : m\_r(r), m\_c(c) {}

Coord() : m\_r(0), m\_c(0) {}

double r() const { return m\_r; }

double c() const { return m\_c; }

private:

double m\_r;

double m\_c;

};

int main()

{

Map<int, double> mid;

mid.insert(42, -1.25); // OK

Map<Coord, int> mpi;

mpi.insert(Coord(40,10), 32); // error!

}

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 ultimate root cause of the problem and why that is connected to the call to Map<Coord, int>::insert.

The main problem is that no operators are defined for the Coord class. Therefore, the compiler has no way of using those operators in the Map functions. For example, the doInsertOrUpdate function requires the comparison operator ==. The compiler doesn’t know how to compare whether or not two Coords are equal unless you tell it what to do. Therefore, you would need to define an operator== for the Coord class. You would also need to define an operator< (although the details of what that would be are a little confusing since you would have to determine what makes a coordinate greater than another coordinate).

4b. 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?

We could not solve this problem if we had only a one-parameter listAll because the constraints in part a were that queues and stacks were not allowed. We cannot visit nodes more than once without a queue/stack, so we wouldn’t be able to print repeated nodes (ie. edu would not be printed multiple times). By using the path parameter, we are able to keep adding the labels and increasing the path, so we are able to print repeated domains.

5a. In order to prepare for the inevitable spread of another dangerous virus, the local health department is developing software that maintains, for N people numbered 0 through N-1, a two-dimensional array of bool hasContacted that records which people have been in close contact with others: hasContacted[i][j] is true if and only if person i and person j have been in close contact. If person i has been in close contact with person k, and person k has been in close contact with person j, we call person k a *direct intermediary* between person i and person j.  
The department has developed an algorithm that, for every pair of people i and j, determines how many direct intermediaries they have between them. Here's the code:  
 const int N = *some value*;

bool hasContacted[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 (hasContacted[i][k] && hasContacted[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(N4+7N2).)

The time complexity is O(N3) because there are three for loops nested together that are each iterated until N. The time complexity of the innermost for loop is N. Then, the middle for loop is iterated N times, and for each of those times, it would go through the innermost for loop N times. Therefore, the time complexity of the inner two for loops is N\*N or N2. Finally, the outermost for loop goes through the middle for loop N times (which in turn goes through the innermost for loop N times). Therefore, the final time complexity for all three nested for loops is N\*(N\*N) which is equivalent to O(N3).

5b. The algorithm in part a doesn't take advantage of the symmetry of contact: for every pair of persons i and j, hasContacted[i][j] == hasContacted[j][i]. One can skip a lot of operations and compute the number of direct intermediaries more quickly with this algorithm:  
 const int N = *some value*;

bool hasContacted[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 < **i**; j++) **// loop limit is now i, not N**

{

numIntermediaries[i][j] = 0;

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

{

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

continue;

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

numIntermediaries[i][j]++;

}

**numIntermediaries[j][i] = numIntermediaries[i][j];**

}

}

What is the time complexity of this algorithm? Why?

The time complexity is still O(N3). The middle loop occurs (0, 1, 2, 3… N-1) times depending on which iteration the first loop is on. Based on the arithmetic series formula, this is equal to (0 + N-1) \* (N/2). This is then multiplied by the time complexity of the third for loop which is N. Therefore, the answer is (N-1)\*(N/2) \* N. Because we don’t care about the constants, we can simplify this to N \* N \* N, which is still O(N3).

6a. Here again is the non-member reassign function for Map from [Map.cpp](https://web.cs.ucla.edu/classes/winter24/cs32/Homeworks/4/Map.cpp):  
void reassign(const Map& m, Map& result)

{

// Guard against the case that result is an alias for m (i.e., that

// result is a reference to the same map that m 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.

Map res;

if (!m.empty())

{

KeyType prevKey;

ValueType value0;

// Get pair 0, which must succeed since m is not empty

m.get(0, prevKey, value0);

// For each pair i after pair 0, insert into res a pair with

// pair i-1's key and pair i's value. (This loop executes 0 times

// if m has only one pair.)

for (int i = 1; i < m.size(); i++)

{

KeyType k;

ValueType v;

m.get(i, k, v);

res.insert(prevKey, v);

prevKey = k;

}

// Insert a final pair with last pair's key and pair 0's value.

res.insert(prevKey, value0);

}

result.swap(res);

}

Assume that m 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?

Calling the get function at the beginning has a time complexity of O(1), so we can ignore that in the overall calculation since it is insignificant. Then, the reassign function will go through the loop N times, each time calling both the insert function and the get function. The get function will have a worst case time complexity of N/2, which is simplified to N. The insert function calls the doInsertOrUpdate function which calls the findFirstAtLeast function. The findFirstAtLeast function has a worst case time complexity of O(N) (in case the node is at the very end), and the other statements in the doInsertOrUpdate function are insignificant in comparison. This means the doInsertOrUpdate function and the insert function both have O(N) as the time complexity. So the loop in the reassign function has a time complexity of O(N\*(N + N)) which simplifies to O(N2). Finally, the reassign function calls the swap function which has an overall time complexity of O(1) and is therefore insignificant in the overall calculation. Therefore, the time complexity for the reassign function is O(N2).

6b. Here is an implementation of a related member function. The call  
m.reassign();

has the same effect as calling the non-member function above as reassign(m, m);. The implementation is  
void Map::reassign()

{

Node\* p = m\_head->m\_next;

if (p != m\_head)

{

ValueType value0 = p->m\_value;

for ( ; p->m\_next != m\_head; p = p->m\_next)

p->m\_value = p->m\_next->m\_value;

p->m\_value = value0;

}

}

Assume that \*this has N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its time complexity? Why? Is it the same, better, or worse, than the implementation in part a?

Assuming \*this has N elements, the time complexity is O(N). This is because there is only one for loop which loops through all the nodes from the head to the final node. This loops through N times, generating a time complexity of O(N). This is better than the implementation in part a.