Solutions to problems 2, 4b, 5a, 5b, 6a, and 6b

2) The one-argument form of Sequence<Coord>::insert causes a compilation error because the function that it calls in Sequence uses a comparison operator(greater-than operator) in the implementation. Since the item being inserted is an object of type Coord, we did not define how the comparison operator is supposed to compare the Coords in the Sequence, and thus causes a compilation error.

4b) We cannot solve this problem given the constraints in part a if we had only a one-parameter listAll since the recursive program relies on the string path to store all the possible paths that it is supposed to print out. Without the second parameter, the function is unable to return to the root node initially passed in to print out the paths, since the program has to keep iterating through the children nodes.

5a)

const int N = *some value*;

bool isFriend[N][N];

...

int numMutualFriends[N][N];

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

{

numMutualFriends[i][i] = -1; // the concept of mutual friend

// makes no sense in this case

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

{

if (i == j)

continue;

numMutualFriends[i][j] = 0;

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

{

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

continue;

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

numMutualFriends[i][j]++;

}

}

}

We can see that the time complexity of this algorithm is O(N^3). For the outer and middle for loop, we know it enters N times, with time complexity O(N) for both. For the third most inner for loop, we can see that it only runs when i != j. This means that it will run one less time when i == j. In our big O notation, we only look at the worst-case scenario, so this means the loop runs N\*N\*N = N^3 times with a big O notation of O(N^3).

5b) Taking into account the symmetry of friendship, the time complexity is still O(N^3). We know this because if we look at the outer and middle for loop relationships, we can see that the outer loop causes the middle loop to be run a certain amount of time, where it runs 1+2+…+N times total, which is equal to the equation 1/2N^2+1/2N. This still makes the two for loops of order N^2 since we only look at the highest order term. Lastly, the inner loops is till order N, so together, the function is O(N^3).

6a)

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result)

{

Sequence res;

int n1 = seq1.size(); //============================> just returns m\_size

int n2 = seq2.size();

int nmin = (n1 < n2 ? n1 : n2);

int resultPos = 0;

for (int k = 0; k < nmin; k++) //=========================> nmin is O(N)

{

ItemType v;

seq1.get(k, v); //====> calls nodeAtPos(pos), which goes

//through the list from either the front or back, so the maximum number of //iterations is 1/2N. We say N since we look at the term of highest order.

//O(N)

res.insert(resultPos, v); //====> calls nodeAtPos(pos), which goes //through the list from either the front or back, so the maximum number of //iterations is 1/2N. We say N since we look at the term of highest order.

//O(N)

resultPos++;

seq2.get(k, v); //====> calls nodeAtPos(pos), which goes

//through the list from either the front or back, so the maximum number of //iterations is 1/2N. We say N since we look at the term of highest order.

//O(N)

res.insert(resultPos, v); //====> calls nodeAtPos(pos), which goes //through the list from either the front or back, so the maximum number of //iterations is 1/2N. We say N since we look at the term of highest order. //O(N)

resultPos++;

}

const Sequence& s = (n1 > nmin ? seq1 : seq2); // calls copy constructor //which goes through each element in the sequence and copies it, O(N)

int n = (n1 > nmin ? n1 : n2);

for (int k = nmin ; k < n; k++) //=========> does not run since both //sequences have N elements

{

ItemType v;

s.get(k, v); //====> calls nodeAtPos(pos), which goes

//through the list from either the front or back, so the maximum number of //iterations is 1/2N. We say N since we look at the term of highest order. //O(N). Does not run, however.

res.insert(resultPos, v); //====> calls nodeAtPos(pos), which goes //through the list from either the front or back, so the maximum number of //iterations is 1/2N. We say N since we look at the term of highest order. //O(N) . Does not run, however.

resultPos++;

}

result.swap(res); // only swaps pointers and int value, doesn’t affect O

}

In total, the function has a time complexity of O(N^2), as explained in the code above, where the get and insert function call nodeAtPos, which goes through the list N/2 times, which means it has order N. This happens 4 times inside the first for loop, meaning it has order 4N. Then, since we do not care about the constant in front of N, we see that the first for loop is of order N^2. The copy constructor call has order N since it has a loop going through all the elements in the sequence to copy over into the other one. The last for loop does not run since we are saying that the sizes of all three sequences passed in are N. The loop only runs the difference between the length of seq1 and seq2. Then, the order for the function is O(N^2+N), which is then O(N^2) if we just take the highest order term.

6b)

void Sequence::interleave(const Sequence& seq1, const Sequence& seq2)

{

Sequence res;

Node\* p1 = seq1.m\_head->m\_next;

Node\* p2 = seq2.m\_head->m\_next;

for ( ; p1 != seq1.m\_head && p2 != seq2.m\_head;

p1 = p1->m\_next, p2 = p2->m\_next)

// the for loop runs N times since the sequences have N elements O(N)

{

res.insertBefore(res.m\_head, p1->m\_value);

res.insertBefore(res.m\_head, p2->m\_value);

}

Node\* p = (p1 != seq1.m\_head ? p1 : p2);

Node\* pend = (p1 != seq1.m\_head ? seq1 : seq2).m\_head;

for ( ; p != pend; p = p->m\_next) // this loop does not run since the //list is a doubly linked list, where p and pend are both pointing to m\_head

res.insertBefore(res.m\_head, p->value);

// Swap \*this with res

swap(res); // swaps pointers, does not affect time complexity

// Old value of \*this (now in res) is destroyed when function returns.

}

We can see that the first for loop runs through the lists once, so it has order N. The insertBefore function is called for them, but this does not traverse through the list, so it does not affect the time complexity. The second for loop has . As a result, the function has order O(N), which is a lot better than the previous function.