**Homework 4 Solution**

**Q.2**

The one parameter insert function does not work for the Coord class because the insert function compares the parameter ‘value’ and the data member m\_value of the linked list with a > operator. Since the class Coord does not have a operator> function defined for it, the compiler does not know what to do when comparing two Coord class objects. Therefore, this results in an error message.

**Q.4 (b)**

We could not solve the problem given the constraints in part a if we only had a one-parameter ListAll because without any global variables or any additional containers, it is not possible to keep track of the path from the root to each node of the tree.

**Q.5 (a)**

The time complexity of this algorithm would be O(N^3) because the 3 nested for loops each loop for N number of times.

int numMutualFriends[N][N];

for (int i = 0; i < N; i++) 🡸====================== O(N^3)

{

numMutualFriends[i][i] = -1; 🡸=============== O(1)

for (int j = 0; j < N; j++) 🡸===================== O(N^2)

{

if (i == j) 🡸================================ O(1)

continue;

numMutualFriends[i][j] = 0; 🡸================= O(1)

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

{

if (k == i || k == j) 🡸====================== O(1)

continue;

if (isFriend[i][k] && isFriend[k][j]) 🡸========= O(1)

numMutualFriends[i][j]++; 🡸============= O(1)

}

}

}

**Q.5 (b)**

The time complexity of this algorithm would still remain 0(N^3). The time complexity of the above algorithm and this one both are 0(N^3) because of the 3 nested for loops, but this algorithm would have a lower constant of proportionality because of the 2nd for loop going till ‘i’ instead of ‘N’ and also because of the line “numMutualFriends[j][i] = numMutualFriends[i][j].” Since we are ignoring the constants of proportionalities and the lower order terms, the two algorithms have the same constant of proportionality.

int numMutualFriends[N][N];

for (int i = 0; i < N; i++) 🡸================================= O(N^3)

{ 🡸================================================== O(N^2)

numMutualFriends[i][i] = -1; 🡸=========================== O(1)

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

{ 🡸================================================== O(N)

numMutualFriends[i][j] = 0; 🡸=========================== O(1)

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

{

if (k == i || k == j) 🡸================================ O(1)

continue;

if (isFriend[i][k] && isFriend[k][j]) 🡸==================== O(1)

numMutualFriends[i][j]++; 🡸======================== O(1)

}

numMutualFriends[j][i] = numMutualFriends[i][j]; 🡸========== O(1)

}

}

**Q.6 (a)**

The time complexity of the interleave function here is 0(N^2). This is because the for loops loop for N times and the call to the insert function inside the for loops take a time of order 0(N) because the function also has a for loop inside it, therefore making the function O(N^2).

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

{ 🡸====================================== O(N^2)

Sequence res; 🡸========================== O(1)

int n1 = seq1.size(); 🡸====================== O(1)

int n2 = seq2.size(); 🡸======================= O(1)

int nmin = (n1 < n2 ? n1 : n2); 🡸=============== O(1)

int resultPos = 0; 🡸========================== O(1)

for (int k = 0; k < nmin; k++) 🡸================= O(N^2)

{

ItemType v;

seq1.get(k, v); 🡸========================== O(N)

res.insert(resultPos, v); 🡸=================== O(N)

resultPos++;

seq2.get(k, v); 🡸=========================== O(N)

res.insert(resultPos, v); 🡸====================O(N)

resultPos++;

}

const Sequence& s = (n1 > nmin ? seq1 : seq2);

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

for (int k = nmin ; k < n; k++) 🡸================= O(N^2)

{

ItemType v; 🡸============================ O(1)

s.get(k, v); 🡸============================= O(N)

res.insert(resultPos, v); 🡸=================== O(N)

resultPos++;

}

result.swap(res); 🡸========================== O(1 )

}

**Q.6 (b)**

The time complexity of the interleave function here is 0(N). This is because the for loops loop for N times and the call to the insertBefore function inside the for loops take a time of order 0(1), therefore making the function O(N). Therefore, this implementation of the function is better than the implementation in part a.

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

{ 🡸==================================================== O(N)

Sequence res; 🡸======================================= O(1)

Node\* p1 = seq1.m\_head->m\_next; 🡸===================== O(1)

Node\* p2 = seq2.m\_head->m\_next; 🡸===================== O(1)

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

p1 = p1->m\_next, p2 = p2->m\_next) 🡸============ O(N)

{

res.insertBefore(res.m\_head, p1->m\_value); 🡸============ O(1)

res.insertBefore(res.m\_head, p2->m\_value); 🡸============ O(1)

}

Node\* p = (p1 != seq1.m\_head ? p1 : p2); 🡸================ O(1)

Node\* pend = (p1 != seq1.m\_head ? seq1 : seq2).m\_head; 🡸=== O(1)

for ( ; p != pend; p = p->m\_next) 🡸========================= O(N)

res.insertBefore(res.m\_head, p->value); 🡸=============== O(1)

// Swap \*this with res

swap(res); 🡸========================================== O(1)

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

}