2. Because the one parameter insert function will try to call operator > or < on Coord. However, we didn’t overload these operators, therefore, the compiler doesn’t know how to compare two Coords, which leads to error.

4b. To implement a recursive function, we need to reduce the size of the problem with every recursive call. That means one of our parameters must be a pointer to an instance of Domain, and each time we will pass the pointer to the subdomains to the function. The function must also in some ways “return” the result after it solves the problem. Since the function is void, the only way to pass the result back is to define the second string parameter.

5a For this and the following 3 answers, I will write out the time complexity of each loop and function call in the **blue comment** and state the final answer in the end.

const int N = some value;

bool isFriend[N][N];

...

int numMutualFriends[N][N];

for (int i = 0; i < N; i++) // this loop is O(N)

{

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

// makes no sense in this case

for (int j = 0; j < N; j++) // this loop is O(N)

{

if (i == j)

continue;

numMutualFriends[i][j] = 0;

for (int k = 0; k < N; k++) // this loop is O(N)

{

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

continue;

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

numMutualFriends[i][j]++;

}

}

}

Every loop has O(N), and each is contained within an outer loop(except for the outmost one), so the time complexity should be O(N\*N\*N) = O(N^3).

5b

const int N = *some value*;

bool isFriend[N][N];

...

int numMutualFriends[N][N];

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

{

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

// makes no sense in this case

for (int j = 0; j < **i**; j++) //if i = N, the loop is also O(N)

{

numMutualFriends[i][j] = 0;

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

{

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

continue;

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

numMutualFriends[i][j]++;

}

numMutualFriends[j][i] = numMutualFriends[i][j];

}

}

The structure of the loop is unchanged, although the second loop has upper limit i, in the worst case(i = N), the order of the second loop is still O(N). So the time complexity is the same as 5a: O(N^3). Although this algorithm could be a bit more efficient than 5a.

6a

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

{

Sequence res;

int n1 = seq1.size();

int n2 = seq2.size();

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

int resultPos = 0;

for (int k = 0; k < nmin; k++) //since seq1 and seq2 both have size N, nmin = N

// the loop is O(N)

{

ItemType v;

seq1.get(k, v); //the get() function calls nodeAtPos() function

//this function searches each node from the end or the head to the

//middle until it reaches the value matches, therefore it is O(N/2)

res.insert(resultPos, v); //v is always insert to the end of the res sequence, so it is

//O(1)

resultPos++;

seq2.get(k, v); //O(N/2)

res.insert(resultPos, v); // O(1)

resultPos++;

}

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

//since seq1 and seq2 have the same size, this loop will be skipped

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

for (int k = nmin ; k < n; k++)

{

ItemType v;

s.get(k, v);

res.insert(resultPos, v);

resultPos++;

}

result.swap(res); // the swap function only swaps the head pointer, O(1)

// result will now be destroyed, the destructor calls doErase() function, which visit each node, link the node right before and after the node, then destroy the node. Each deletion visits 3 nodes, total 2N delete is executed, so order O(2N);

}

The structure of the loop is therefore an O(N) loop contains two O(N/2) steps. The sum of two O(N/2) steps is O(N). The constant actually doesn’t matter here. And the final result is O(N\*N) + O(2N) = O(N^2)

6b

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

{

Sequence res;

//this is a circularly linked list with a dummy node

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)

//this loop go through every node in seq1 and seq2, thus O(N)

{

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

//this function creates a new node, link it in between the head and the node right before

//the head, so only two nodes are visited each time, O(2)

res.insertBefore(res.m\_head, p2->m\_value); //O(2) as well

}

Node\* p = (p1 != seq1.m\_head ? p1 : p2); //again, seq1,2 have the same size, skip

//this part

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

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

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

// Swap \*this with res

swap(res); //only swap the head and the size,

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

// result will now be destroyed, the destructor calls doErase() function, which visit each node, link the node right before and after the node, then destroy the node. Each deletion visits 3 nodes, total N delete is executed, so order O(N);

}

The total time complexity is therefore O(1) + O(1) + O(N\*4) + O(N) = O(N).