2) The call to the one-argument form of Sequence<Coord>::insert causes a compilation error because the one-argument form of the insert function inserts at a position found by comparing the values of the items in the sequence with the > operator. While this works for primitive types like integers for which the > operator has been defined, because we haven’t overloaded a definition of the > operator for Coords, the one-argument insert function doesn’t know what it means to compare two Coords with > (are we comparing m\_row? m\_col? the sum of both?).

4b) We wouldn’t be able to implement just the one-parameter version of listAll as a recursive function to solve the problem because the one-parameter version doesn’t return a string “updating” its location in the pathname, so we wouldn’t have access to the prepended paths and wouldn’t be able to print the full pathname.

5a) **O(N^3),** because there are 3 for loops that all loop N times, and all the other operations are proportional to a constant (constant time). My work step-by-step is shown below, starting from the innermost loop and working my way out from there.

O(N^3):

const int N = *some value*; -----> O(1)

assert(N > 2); // algorithm fails if N <= 2 -----> O(1)

double dist[N][N]; -----> O(1)

...

int bestMidPoint[N][N]; ----> O(1)

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

{ ------> O(N\*N) = O(N^2)

bestMidPoint[i][i] = -1; // one-stop trip to self is silly ----> O(1)

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

{ -------> O(N)

if (i == j) ------> O(1)

continue; -----> O(1)

int minDist = *maximum possible integer*; ------> O(1)

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

{ -----------> O(1)

if (k == i || k == j) -------> O(1)

continue; ---------> O(1)

int d = dist[i][k] + dist[k][j]; -----> O(1)

if (d < minDist) -----> O(1)

{ ----> O(1)

minDist = d; ---> O(1)

bestMidPoint[i][j] = k; ----> O(1)

}

}

}

}

5b) **O(N^3),** because there are 3 for loops, 1 of which loops N times, and the other two of which equal looping i times for each N, where i = N. All the other operations are proportional to a constant (constant time). The two that equal looping i times for each N is equivalent to O(0+1+2+3+...+(N-1)), which equals (N-1)\*N/2, or .5N^2 - .5N, which for our purposes looking at only large N (ie. only higher order terms and ignoring constants), is equivalent to O(N^2). Multiplying this by the last for loop that loops N times gives a time complexity of O(N^3).

My work step-by-step is shown below, starting from the innermost loop and working my way out from there.

O(N^3):

const int N = *some value*; -----> O(1)

assert(N > 2); // algorithm fails if N <= 2 -----> O(1)

double dist[N][N]; -----> O(1)

...

int bestMidPoint[N][N]; ----> O(1)

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

{ ------> O(0+1+2+....+(N-1)) = (N-1)\*N/2 = O(N^2)

bestMidPoint[i][i] = -1; // one-stop trip to self is silly ----> O(1)

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

{ -------> O(N)

int minDist = *maximum possible integer*; ------> O(1)

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

{ -----------> O(1)

if (k == i || k == j) -------> O(1)

continue; ---------> O(1)

int d = dist[i][k] + dist[k][j]; -----> O(1)

if (d < minDist) -----> O(1)

{ ----> O(1)

minDist = d; ---> O(1)

bestMidPoint[i][j] = k; ----> O(1)

**bestMidPoint[j][i] = k;** ----> O(1)

}

}

}

}

6a) **O(N^2)** is the time complexity in terms of how many linked list nodes are visited in this function because the first for loop calls get() for each sequence nmin (N) times. get() is called an average of O(N) times, so within the first for loop it’s O(2\*N^2), one O(N\*N) for each sequence. Because we’re ignoring constants (and the second for loop doesn’t even get called (n1 == n2)) and also ignoring lower order terms (+O(N)), this gives us a time complexity of O(N^2) for the purposes of this homework. My work is shown below.

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

{ -----> O(2\*N^2) + O(N) = **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(2\*N^2)

{ -----> O(N+N) = O(2\*N)

ItemType v; -----> O(1)

seq1.get(k, v); -----> O(N)

res.insert(resultPos, v); -----> O(1)

resultPos++; -----> O(1)

seq2.get(k, v); -----> O(N)

res.insert(resultPos, v); -----> O(1)

resultPos++; -----> O(1)

}

const Sequence& s = (n1 > nmin ? seq1 : seq2); -----> O(1)

int n = (n1 > nmin ? n1 : n2); ----> O(1)

for (int k = nmin ; k < n; k++) -----> O(N^2) \*doesn’t get executed\*

{ -----> O(N)

ItemType v; -----> O(1)

s.get(k, v); -----> O(N)

res.insert(resultPos, v); -----> O(1)

resultPos++; ----> O(1)

}

result.swap(res); -----> O(N)

}

6b) **O(N)**, because for this version of the interleave function, insertBefore doesn’t have to traverse the linked list as it inserts at the “end” of the list (right before the head) for every call. This means it has a constant time, and this is called up to N times in each for loop. Disregarding constants of proportionality (\*3), this gives us a linear time complexity of O(N), which is better than the implementation in part a) (O(N^2)). Work is shown below.

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

{ -----> O(N + N + N) = O(3\*N) = **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)

{ -----> O(1)

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(N)

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

}