CS32 HW4

705120273

Melody Chen

**Question 2:** This causes at least one compile error because our class of coordinates did not override the comparators for coordinates, so the compiler does not know how to compare the coordinates. In our insert function that takes only one parameter, it uses the > comparator, which we did not define in our coordinates class. It works for the other functions mentioned because for int, the > comparator is defined and for the other insert of coordinates, it did not use a comparator.

**Question 4B:** If we only had one parameter for this problem, we will not be able to come up with a solution because we will not have the ability to concatenate strings and pass them on to the next recursion. If string concatenation is not possible for this function, then it will be very difficult to make the parent of a node to be printed out for every children of the parent.

**Question 5:**

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]++;

}

}

}

The complexity for the algorithm above should be O(N3) because there’s three nested for-loops and their worst cases are O(N).

const int N = *some value*;

bool isFriend[N][N];

...

int numMutualFriends[N][N];

for (int i = 0; i < N; i++) O(N2)

{

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

// makes no sense in this case

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

{

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]++;

}

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

}

}

The complexity is O(N3) because there is three nested for loops. The first loop runs N times, and the 2nd nested for loop runs N-1 times, and the 3rd nested loop runs N times.

**Question 6:**

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++) //O(N2)

{

ItemType v;

seq1.get(k, v);

res.insert(resultPos, v);

resultPos++;

seq2.get(k, v);

res.insert(resultPos, v);

resultPos++;

}

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

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 complexity is O(N2) because the first loop runs N times and the functions it calls runs at most N times as well. So the first loop is O(N2). The copy constructor is O(N) because there is a single loop that runs N times. The second loop runs depends on the difference of the size of the two sequences. So overall, the complexity is O(N2).

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)

{

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)

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

// Swap \*this with res

swap(res);

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

}

The complexity is O(N) because the first loop runs to the max size of one of the sequences, so it is O(N). The second loop runs depending on the difference of the size. Thus, in general the complexity is O(N).