Com Sci 32

Homework 2

Problem 2

The one argument form of Sequence<ItemType>::insert() involves in its implementation a comparison between the instance of ItemType to be inserted and those already within the Sequence so that the new item can be inserted at an appropriate position. Of course, for a call to Sequence<ItemType>::insert() to succeed, it should be legal to use the comparison operators for objects of the ItemType class/ pre-defined datatypes. As is visible in the implementation of the Coord class, the comparison operators have not been overloaded, thus a call to the one-argument form of Sequence<Coord>::insert() results in a compilation error since it tries to compare two Coord objects, which is not legal. The two argument form of Sequence<Coord>::insert() works fine because it does not involve any comparisons, so there are no attempts to compare to Coord objects that could potentially lead to compile errors. In the case of Sequence<int>::insert(), both the one-argument and two-argument forms work fine since it is alright to compare two integers.

Problem 4b

If the recursive function listAll() had only one parameter, one might propose the following solution: recursively call the function for all subdomains of a particular domain, print a dot and then the label of the domain, handling the base cases and root case appropriately. Although the following solution would gather the labels of each node along each path from the root to leaves in the tree, and so even in the right order, the output wouldn’t be in the format we desire (which is that each unique path from the root to a leaf should have a unique line to itself). This is one of the reasons why the single parameter overloading of listAll() won’t do the job we would desire it to.

To ensure that we get output in an appropriate manner, we should only be performing print actions at the leaves, and the strings we print should contain in them the path that was traversed to reach them from the root. This will ensure that we obtain all paths from the root to leaves in separate lines. In the single parameter overloading of listAll(), there wouldn’t be a string to append the current label to such that this modified string becomes available to the next (deeper) recursive call. The problem could also not be resolved by declaring a string variable locally as it wouldn’t be available to deeper recursive calls, neither would it have any information from the recursive calls above it. This is why a string parameter is essential so that every time in a recursive call, we can separately append the current Domain’s individual subdomains’ labels onto them and call the function recursively for the current Domain’s subdomains (if any) with these modified string copies, so that the subdomains can further modify the string copies passed onto them. Of course, the root node will be handled appropriately since the one-parameter overload calls within it the two-parameter overload with root’s label, so that is also included in the path (if the root’s label is non-empty). Only when the recursive calls hit their base cases (i.e. when the function is called for a pointer to a Domain that does not have any sub-Domains) do we print the string parameter passed onto them, which would have gathered information successively through each recursive call above them and would ensure that the paths from the root to each leaf have a unique line for themselves, thereby printing all paths from the root to leaves in separate lines.

Problem 5a

The time complexity of the following algorithm is **O(N^3)** as by ignoring any other constant time statements, it essentially uses three nested for loops, and is explained as follows using the inside-out approach:

const int N = *some value*;

bool isFriend[N][N];

...

int numMutualFriends[N][N];

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

{

numMutualFriends[i][i] = -1; // the concept of mutual friend **<===== O(1)**

// makes no sense in this case

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

{

if (i == j) **<===== O(1)**

continue; **<===== O(1)**

numMutualFriends[i][j] = 0;

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

{

if (k == i || k == j) **<===== O(1)**

continue; **<===== O(1)**

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

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

}

}

}

Problem 5b

The time complexity of this algorithm is **O(N^3)** as well. To do so, we can trace the function inside-out by replacing any non-constant loop bounds with their maximum values –

const int N = *some value*;

bool isFriend[N][N];

...

int numMutualFriends[N][N];

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

{

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

// makes no sense in this case

for (int j = 0; j < **N**; j++) **// although the loop limit is now i, we consider it to be N for the algorithm’s big-O analysis. <===== O(N^2)**

{

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];**

}

}

Thus for purposes of big-O analysis, the algorithm is identical to its predecessor and therefore also has the time complexity O(N^3). Even though utilizing the symmetry relation isFriend[i][j] = isFriend[j][i] reduces the number of operations (by half exactly), essentially, in the worst case, each for loop runs N times, and since these for loops are nested inside each other, the time complexity is O(N3) (although the present algorithm has a better coefficient for N^3 than the previous algorithm – ½ for the present algorithm versus 1 for the first algorithm - because of which it performs better than the first).

Problem 6a

Using the inside-out approach, the time complexity of this algorithm is **O(N^2)** –

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result) **<====== O(N2)**

{

Sequence res;

int n1 = seq1.size(); **<====== O(1)**

int n2 = seq2.size(); **<====== O(1)**

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

int resultPos = 0;

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;

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

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

resultPos++;

}

result.swap(res); **<====== O(1)**

} // destruction of res **<====== O(N)**

Thus, we can infer that the time complexity for the execution of the interleave() function would be O(N^2). Note that for deciding the time complexity of the Sequence<ItemType>::insert() function, we referred to its implementation which in turn calls sequentially the Sequence<ItemType>::nodeAtPos() and Sequence<ItemType>::insertBefore() functions, which are respectively of time complexities O(N) and O(1), thus making the time complexity for the Sequence<ItemType>::insert() function O(N). Similarly, Sequence<ItemType>::get() calls the Sequence<ItemType>::nodeAtPos() function, which makes its time complexity O(N). Sequence<ItemType>::size() just returns a private integer data member, and Sequence<ItemType>::swap() performs a simple pointer swap, so the time complexities of both these functions is O(1). However, a call to the destructor involves use of Sequence<ItemType>::doErase() (O(1)) in a while loop (thus making the destructor’s time complexity O(N)). Essentially, calling nodeAtPost, which has a time complexity O(N) within a loop causes the the time complexity of the function’s execution to be O(N^2).

Problem 6b

Using the inside-out approach once again, the time complexity of this algorithm turns out to be **O(N)** -

void Sequence::interleave(const Sequence& seq1, const Sequence& seq2) **<====== O(N)**

{

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) **<====== 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);

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

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.**<====== O(N)**

}

For reasons stated in part a of problem 6, the time complexities of individual components of this function can easily be determined, and as is shown above, the execution of the whole function turns out to have a time complexity of O(N). This is better than the non-member implementation in part a which was of time complexity O(N2). The difference here is made by avoiding the use of looping through integer indices and calling the Sequence<ItemType>::nodeAtPos() function (which had a complexity O(N) and hence made the complexity of such loops O(N2)) at each iteration, and directly using Node pointers to traverse the linked list.