1e)Make sure you understand why the code below passes the first two tests but fails the third. Draw pictures if necessary. Explain in a sentence or two what happens during the execution of test case 3 that eventually leads to test case 3 failing?

#include <iostream>

#include <vector>

#include <list>

using namespace std;

const int MAGIC = 11223344;

void test()

{

bool allValid = true;

vector<int> v1(5, MAGIC);

int k = 0;

for (; k != v1.size(); k++)

{

if (v1[k] != MAGIC)

{

cout << "v1[" << k << "] is " << v1[k] << ", not " << MAGIC << "!" << endl;

allValid = false;

}

if (k == 2)

{

for (int i = 0; i < 5; i++)

v1.push\_back(MAGIC);

}

}

if (allValid && k == 10)

cout << "Passed test 1" << endl;

else

cout << "Failed test 1" << endl;

allValid = true;

list<int> l1(5, MAGIC);

k = 0;

for (list<int>::iterator p = l1.begin(); p != l1.end(); p++, k++)

{

if (\*p != MAGIC)

{

cout << "Item# " << k << " is " << \*p << ", not " << MAGIC << "!" << endl;

allValid = false;

}

if (k == 2)

{

for (int i = 0; i < 5; i++)

l1.push\_back(MAGIC);

}

}

if (allValid && k == 10)

cout << "Passed test 2" << endl;

else

cout << "Failed test 2" << endl;

allValid = true;

vector<int> v2(5, MAGIC);

k = 0;

for (vector<int>::iterator p = v2.begin(); p != v2.end(); p++, k++)

{

if (k >= 20) // prevent infinite loop

break;

if (\*p != MAGIC)

{

cout << "Item# " << k << " is " << \*p << ", not " << MAGIC << "!" << endl;

allValid = false;

}

if (k == 2)

{

for (int i = 0; i < 5; i++)

v2.push\_back(MAGIC);

}

}

if (allValid && k == 10)

cout << "Passed test 3" << endl;

else

cout << "Failed test 3" << endl;

}

int main()

{

test();

}

**ANSWER TO 1E:**

Make sure you understand why the code below passes the first two tests but fails the third. Draw pictures if necessary

The second test passes because the iterator p never points outside the linked list. The execution of

for (int i = 0; i < 5; i++)

l1.push\_back(MAGIC);

increases the size of l1. l1.end() updates to the new end position which is

(l1.begin() + 10) instead of its previous value (l1.begin() + 5). Furthermore, we never dereference a garbage value or a nullptr value in the for loop below.

for (list<int>::iterator p = l1.begin(); p != l1.end(); p++, k++)

{

... ...

}

Thus, when we exit the for loop k == 10 and allValid == true. The program passes the test.

Behind the scenes a typical way (clang++, g++) a vector is implemented is with a node containing a pointer to a dynamically allocated array of that data type, an int that stores the size of the array, and an int that stores the capacity of the array. If we are adding elements to our vectors using v.push\_back(…), we might run out of storage in the dynamically allocated array. If this happens the STL vector creates a new dynamically allocated array about 1.6 times the size of the previous array and stores all the elements in the previous array into this new one while simultaneously updating the variables that store the size and capacity of the array. It then deletes the previous array.

The first test passes because the [] operator accounts for this mechanism and always returns a reference to the element at position n in the vector container. (v1[k] != MAGIC) always returns false so allValid stays true. after the execution of

if (k == 2)

{

for (int i = 0; i < 5; i++)

v1.push\_back(MAGIC);

}

v1.size() starts returning 10, resulting in k == 10, after the outer for loop is exited. Thus, the program passes.

Explain in a sentence or two what happens during the execution of test case 3 that eventually leads to test case 3 failing?

On the other hand, the third test fails because of the following:

Initially, v2 has 5 elements stored in it. Following the execution of

for (int i = 0; i < 5; i++)

v2.push\_back(MAGIC);

the mechanism described above takes place since 10 > (1.6)\*5. The previous array is destroyed. After the execution of

if (\*p != MAGIC)

{

... ...

... ...

}

the program crashes since p still points to memory where the old array used to be (p points to a garbage value after the execution of the for loop above and dereferencing that garbage value causes the program to crash).

3)Explain in a sentence or two why the call to the one-argument form of Sequence<Coord>::insert causes at least one compilation error. (Notice that the call to the one-argument form of Sequence<int>::insert is fine, as is the call to the two-argument form of Sequence<Coord>::insert.) Don't just transcribe a compiler error message; your answer must indicate you understand the ultimate root cause of the problem and why that is connected to the call to Sequence<Coord>::insert.

#include "Sequence.h" // class template from problem 1

class Coord

{

public:

Coord(int rr, int cc) : m\_row(rr), m\_col(cc) {}

Coord() : m\_row(0), m\_col(0) {}

double r() const { return m\_row; }

double c() const { return m\_col; }

private:

double m\_row;

double m\_col;

};

int main()

{

Sequence<int> si;

si.insert(50); // OK

Sequence<Coord> sc;

sc.insert(0, Coord(50,20)); // OK

sc.insert(Coord(40,10)); // error!

}

**ANSWER TO 3:**

The reason that sc.insert(Coord(40,10)) fails is that the > operator (Op1 > Op2 , where Op1 is an ItemType and Op2 is an int) is not defined for the Coord class. The reason that sc.insert(0, Coord(50,20)) does not result in a compilation error is that insert(int pos, const ItemType& value) does not use any operators that are undefined for the Coord class. si.insert(50) compiles because the > operator (Op1 > Op2 , where both Op1, and Op2 are ints) is defined for the built-in int class.

4)b) We introduced the two-parameter overload of listAll. Why could you not solve this problem given the constraints in part a if we had only a one-parameter listAll, and you had to implement *it* as the recursive function?

**ANSWER TO 4B:**

We could not solve the problem in part a if we only had a one parameter listAll and had to implement it as the recursive function. This is because the decision as to whether or not we should cout dots after the label name depends on the recursion depth. If we had a on-parameter listAll, we would have the current label of the domain but we would not have enough information to construct the proper path string and cout everything in the correct order.

For example, we would have no way of telling if we are in the domain before the root domain without keeping track of the recursion depth which would result in an incorrect cout for the one parameter listAll all together. Hence, we need to keep track of recursion depth. In part a, this issue was taken care of by introducing the string argument called path which allows us to store subdomain names as we go and account for egde cases like couting a single domain with no subdomains (no dots), etc...

5)a What is the time complexity of this algorithm, in terms of the number of basic operations (e.g., additions, assignments, comparisons) performed: Is it O(N), O(N log N), or what? Why? (Note: In this homework, whenever we ask for the time complexity, we care only about the high order term, so don't give us answers like O(N2+4N).)

**ANSWER TO 5A:**

Overall, the time complexity of the algorithm below is O((3/2)\*N^3) approx. **O(N^3).**

**Overall: N\*(O(3\*N^3)+ O(3)) approx. O(N^3) time complexity**

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

bool isFriend[N][N]; **===> O(1)**

...

int numMutualFriends[N][N]; **===> O(1)**

for (int i = 0; i < N; i++)**===>** **N\*(O(3\*N^2)+ O(1)) approx. O(3\*N^3)**

{

numMutualFriends[i][i] = -1; **===> O(1)**

for (int j = 0; j < N; j++)**===>** **N\*(O(3\*N)+ O(2)) approx. O(3\*N^2)**

{

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

continue;

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

for (int k = 0; k < N; k++) **===> O(3\*N)**

{

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

continue;

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

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

}

}

}

5)b) The algorithm in part a doesn't take advantage of the symmetry of friendship: for every pair of users i and j, isFriend[i][j] == isFriend[j][i]. We can skip a lot of operations and compute the number of mutual friends more quickly with this algorithm. What is the time complexity of this algorithm? Why?

**ANSWER TO 5B:**

Overall, the time complexity of the algorithm below is O((3/2)\*N^3) approx. **O(N^3).** This algorithm will be twice as fast as the algorithm in part 5a even though they are of the same time complexity. The reason this is faster is because of the for loop that only iterates through i elements instead of N elements.

Overall: N\*(O((3/2)\*N^3)+ O(3)) **approx. O(N^3) time complexity**

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

bool isFriend[N][N]; **===> O(1)**

...

int numMutualFriends[N][N]; **===> O(1)**

for (int i = 0; i < N; i++)**=>(O(3\*(1+…+(N-1))\*N)+O(1))=> O(N^3\*(3/2))**

{

numMutualFriends[i][i] = -1; **===> O(1)**

// makes no sense in this case

for (int j = 0; j < i; j++)**==> i\*(O(3\*N)+ O(2))approx. O(3\*i\*N)**

{

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

for (int k = 0; k < N; k++)**===> N\*O(3) ==> O(3\*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)**

}

}

6. Here again is the non-member interleave function for Sequences from [Sequence.cpp](http://web.cs.ucla.edu/classes/spring22/cs32/Homeworks/4/Sequence.cpp):

Assume that seq1, seq2, and the old value of result each have N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its time complexity? Why?

**ANSWER TO 6a:**

**Overall, the time complexity of the algorithm below is aprroximately O(N\*log(N)).**

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result) **===> O(2\*N\*log(N)) approx. O(N\*log(N))**

{

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((log(1)+…+log(4\*(N-1)))) using stirling’s approximation we have O(4\*N\*log(4\*N)) equals (4\*N\*(2+log(N)). Thus, approx. O(N\*log(N))**

{

ItemType v; **===> O(1)**

seq1.get(k, v); **===> O(log(k))**

res.insert(resultPos, v); **===> O(log(resultPos))**

resultPos++;**===> O(1)**

seq2.get(k, v); **===> O(log(k))**

res.insert(resultPos, v); **===> O(log(resultPos))**

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(2\*(log(1)+…+log(N-1))) equals O( 2N\*log(N)) using stirling’s approximation we have O(2N\*log(N)), Thus, approx. O(N\*log(N))**

{

ItemType v; **===> O(1)**

s.get(k, v); **===> O(log(k))**

res.insert(resultPos, v); **===> O(log(resultPos))**

resultPos++; **===> O(1)**

}

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

}

-----------------------------------------------------------------------------

void Sequence::swap(Sequence& other) **===> O(5) approx. O(1)**

{

// Swap head pointers

Node\* p = other.m\_head; **===> O(1)**

other.m\_head = m\_head; **===> O(1)**

m\_head = p; **===> O(1)**

// Swap sizes

int s = other.m\_size; **===> O(1)**

other.m\_size = m\_size; **===> O(1)**

m\_size = s; **===> O(1)**

}

int Sequence::insert(int pos, const ItemType& value)**==>O(log(pos) +11) approx.O(log(pos))**

{

if (pos < 0 || pos > m\_size) **===> O(1)**

return -1;

Node\* p = nodeAtPos(pos); **===> O(log(pos) + 3)**

insertBefore(p, value); **===> O(7)**

return pos;

}

bool Sequence::get(int pos, ItemType& value) const **==>O(log(pos) + 5) approx.O(log(pos))**

{

if (pos < 0 || pos >= m\_size) **===> O(1)**

return false;

Node\* p = nodeAtPos(pos); **===> O(log(pos))**

value = p->m\_value; **===> O(1)**

return true;

}

void Sequence::insertBefore(Node\* p, const ItemType& value)**==>O(7)approx.O(1)**

{

// Create a new node

Node\* newp = new Node; **===> O(1)**

newp->m\_value = value; **===> O(1)**

// Insert new item before p

newp->m\_prev = p->m\_prev; **===> O(1)**

newp->m\_next = p; **===> O(1)**

newp->m\_prev->m\_next = newp; **===> O(1)**

newp->m\_next->m\_prev = newp; **===> O(1)**

m\_size++; **===> O(1)**

}

Sequence::Node\* Sequence::nodeAtPos(int pos) const **==>O(log(pos) + 3) approx.O(log(pos))**

{

Node\* p; **===> O(1)**

// If pos is closer to the head of the list, go forward to find it.

// Otherwise, start from tail and go backward.

if (pos <= m\_size / 2) **===> O(log(pos))**

{

p = m\_head->m\_next; **===> O(1)**

for (int k = 0; k != pos; k++) **===> O(pos)**

p = p->m\_next; **===> O(1)**

}

else // closer to tail

{

p = m\_head; **===> O(1)**

for (int k = m\_size; k != pos; k--) **===> O(pos)**

p = p->m\_prev; **===> O(1)**

}

return p;

}

6. Assume that seq1, seq2, and the old value of \*this each have about N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its time complexity? Why? Is it the same, better, or worse, than the implementation in part a?

**ANSWER TO 6b:**

**Overall, the time complexity of the algorithm below is aprroximately O(N).** This function is better than the function implementation in part a because it doesn’t use helper functions such as seq2.get(k, v) and res.insert(resultPos, v) in for loops. These functions have a time complexity of O(log(k)) and O(log(resultPos)) respectively. Using these in a for loop increases the time complexity of the function significantly.

void Sequence::interleave(const Sequence& seq1, const Sequence& seq2) **===> O(2\*N) approx.. 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.

}