Cameron Maiden

Professor David Smallberg

COM SCI 32

May 28, 2024

Homework 4 Report

**Problem 1e)**

| #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(); } |
| --- |

The third test case does not pass because when the k == 2 condition is met, more values are being pushed onto the back of the vector. Since the vector has reached capacity, a new array of larger size must be dynamically allocated. This means that the iterator p (hilighted in yellow) becomes invalidated because it was never reassigned to the new dynamically allocated array.

**Problem 3)**

| **#include "Set.h" // class template from problem 2 #include <string> using namespace std;**  **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() {  Set<int> si;  si.insert(21); // OK  Set<string> ss;  ss.insert("21 Savage"); // OK  Set<Coord> sc;  sc.insert(Coord(21, -21)); // error! }** |
| --- |

The reason why sc.insert results in an error is because there is no guarantee that the -> operator does not alter the data members of the Set class, which violates the const Coord guarantee we promised in the findFirstAtMost function. In addition, what does it mean to check if one Coord is greater than another? Is it based on the values of X and Y? The > symbol is ambiguous when comparing two Coords

**Problem 4b)** The reason we are not able to get the correct output for the listAll one parameter overload is because if we try to call the function recursively, the only string the path can output is the name of the current directory, it has no way of appending the path to the previous string, the desired output could only be solved iteratively.

**Problem 5a)**

| **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 of this algorithm is O(N\*N\*N) = O(N^3). This is due to the fact that each for loop runs N times in the worst case scenario, and since the for loops are nested we multiply the complexity of each for loop together.

**Problem 5b)**

| **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 < i; j++) ======> O(N) // 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];  } }** |
| --- |

In this case, the complexity of the algorithm is O(N\*i\*N) = **O(N^3)**. Even though the loop limit for the second for loop is now i instead of N, we only consider the worst case scenario in our algorithm analysis. In this case, the worst case scenario is when i = N. Therefore, the complexity remains the same as in part 5a.

**Problem 6a)**

| **void unite(const Set& set1, const Set& set2, Set& result) {  const Set\* sp = &set2;  if (&result == &set1)  {  if (&result == &set2)  return;  }  else if (&result == &set2)  sp = &set1;  else  {  result = set1;  if (&set1 == &set2)  return;  }  for (int k = 0; k < sp->size(); k++) =====> O(N)  {  ItemType v;  sp->get(k, v); =======> O(N/2)  result.insert(v); =======> O(N)  } }** |
| --- |

In the case of the unite function and assuming that set1, set2 and the old result have N elements, the complexity of the algorithm is O(N \* N) = **O(N^2)**. This is because the for loop iterates N times. And when using the get function, based on the way get is implemented traverses the linked list at most N/2 times. In addition, the insert function in the worst case would have to traverse N elements in the list. Therefore, the total complexity of the unite algorithm is O(N^2)

**Problem 6b)**

| **void Set::unite(const Set& set1, const Set& set2) {  vector<ItemType> v;   // copy all items into v;  for (Node\* p1 = set1.m\_head->m\_next; p1 != set1.m\_head; p1 = p1->m\_next) =======> O(N)  v.push\_back(p1->m\_value);  for (Node\* p2 = set2.m\_head->m\_next; p2 != set2.m\_head; p2 = p2->m\_next) =======> O(N)  v.push\_back(p2->m\_value);   // sort v using an O(N log N) algorithm  sort(v.begin(), v.end()); ========> O(N log N)   // delete result nodes (other than the dummy node)  while (m\_head->m\_next != m\_head) =======> O(N)  doErase(m\_head->m\_next);   // copy unique items from v into result  for (size\_t k = 0; k < v.size(); k++) =======> O(2N)  {  if (k == 0 || v[k] != v[k-1]) // add non-duplicates  insertBefore(m\_head->m\_next, v[k]);**  **}   // v is destroyed when function returns }** |
| --- |

The complexity of this algorithm is **O(N log N)** due to the fact that the sorting algorithm has the biggest time complexity. All of the other operations shown are some degree of O(N)

**Problem 6c)**

| **void Set::unite(const Set& set1, const Set& set2) {  const Set\* sp = &set2;  if (this == &set1)  {  if (this == &set2)  return;  }  else if (this == &set2)  sp = &set1;  else  {  \*this = set1;  if (&set1 == &set2)  return;  }  Node\* p1 = m\_head->m\_next;  Node\* p2 = sp->m\_head->m\_next;  while (p1 != m\_head && p2 != sp->m\_head)  {  if (p1->m\_value > p2->m\_value)  p1 = p1->m\_next;  else  {  if (p1->m\_value < p2->m\_value)  insertBefore(p1, p2->m\_value); =======> O(N)  else  p1 = p1->m\_next;  p2 = p2->m\_next;  }  }  for ( ; p2 != sp->m\_head; p2 = p2->m\_next) =======> O(N)  insertBefore(m\_head, p2->m\_value); }** |
| --- |

In this algorithm, the complexity is **O(N).** The biggest term is both the insertBefore function and for loop at the end of the function. Although those function have the same complexity, we do not consider constants of proportionality in this case.