Homework 4 Solution

Problem 1:

```
void removeOdd(list<int>& li)
{
    list<int>::iterator p = li.begin();
    while (p != li.end())
    {
        if (*p % 2 != 0)
            p = li.erase(p);
        else
            p++;
    }
}
void removeOdd(vector<int>& v)
{
    vector<int>::iterator p = v.begin();
    while (p != v.end())
    {
        if (*p % 2 != 0)
            p = v.erase(p);
        else
            p++;
    }
}
void removeBad(list<Movie*>& li)
{
    list<Movie*>::iterator p = li.begin();
    while (p != li.end())
    {
        if ((*p)->rating() < 50)
        {
            delete *p;  // *p is a Movie*; delete the Movie that *p points to
            p = li.erase(p);
        }
        else
            p++;
    }
}
void removeBad(vector<Movie*>& v)
{
    vector<Movie*>::iterator p = v.begin();
    while (p != v.end())
    {
        if ((*p)->rating() < 50)
        {
            delete *p;  // *p is a Movie*; delete the Movie that *p points to
            p = v.erase(p);
        }
```

```
        else
            p++;
    }
}
```

As we're traversing the vector using an iterator in test case 3, we insert items into the vector. If the vector is full to capacity when we ask to insert something, new storage is allocated so there's more room, the existing data is copied into the new storage, and the existing storage is then deleted, which invalidates the iterator, leaving it dangling, pointing into the old storage that's now gone. (This was not a problem in test case 1, since we were using integer subscripts; an int holding 2 to mark the position in the old storage remains valid, marking the position in the new storage. It was not a problem in test case 2, since for a list, inserting a node does not move the data in a node pointed to by an existing iterator.)

Problem 2:

```
// Set.h

#ifndef SET_INCLUDED
#define SET_INCLUDED

template <typename ItemType>
class Set
{
  public:
    Set();               // Create an empty set (i.e., one whose size is 0).
    bool empty() const;  // Return true if the set is empty, otherwise false.
    int size() const;    // Return the number of items in the set.

    bool insert(const ItemType& value);
      // Insert value into the set if it is not already present.  Return
      // true if the value is actually inserted.  Leave the set unchanged
      // and return false if value is not inserted (perhaps because it
      // was already in the set or because the set has a fixed capacity and
      // is full).

    bool erase(const ItemType& value);
      // Remove the value from the set if it is present.  Return true if the
      // value was removed; otherwise, leave the set unchanged and
      // return false.

    bool contains(const ItemType& value) const;
      // Return true if the value is in the set, otherwise false.

    bool get(int i, ItemType& value) const;
      // If 0 <= i < size(), copy into value the item in the set that is
      // strictly greater than exactly i items in the set and return true.
      // Otherwise, leave value unchanged and return false.

    void swap(Set<ItemType>& other);
```

```
      // Exchange the contents of this set with the other one.

      // Housekeeping functions
    ~Set();
    Set(const Set<ItemType>& other);
    Set<ItemType>& operator=(const Set<ItemType>& rhs);

  private:
      // Representation:
      //   a circular doubly-linked list with a dummy node.
      //   m_head points to the dummy node.
      //   m_head->m_prev->m_next == m_head and m_head->m_next->m_prev == m_head
      //   m_size == 0  iff  m_head->m_next == m_head->m_prev == m_head
      //   If p and p->m_next point to nodes other than the dummy node,
      //      p->m_value > p->m_next->m_value

    struct Node
    {
        ItemType m_value;
        Node*    m_next;
        Node*    m_prev;
    };

    Node* m_head;
    int   m_size;

    void createEmpty();
      // Create an empty list.  (Will be called only by constructors.)

    void insertBefore(Node* p, const ItemType& value);
      // Insert value in a new Node before Node p, incrementing m_size.

    void doErase(Node* p);
      // Remove the Node p, decrementing m_size.

    Node* findFirstAtLeast(const ItemType& value) const;
      // Return pointer to first Node whose m_value >= value if present,
      // else m_head
};

// Declarations of non-member functions

template <typename ItemType>
void unite(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>&
result);
      // result = { x | (x in s1) OR (x in s2) }

template <typename ItemType>
void butNot(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>&
result);
      // result = { x | (x in s1) AND NOT (x in s2) }

// Inline implementations
```

```
template <typename ItemType>
inline
int Set<ItemType>::size() const
{
    return m_size;
}

template <typename ItemType>
inline
bool Set<ItemType>::empty() const
{
    return size() == 0;
}

template <typename ItemType>
inline
bool Set<ItemType>::contains(const ItemType& value) const
{
    Node* p = findFirstAtLeast(value);
    return p != m_head  &&  p->m_value == value;
}

// Non-inline implementations

template <typename ItemType>
Set<ItemType>::Set()
{
    createEmpty();
}

template <typename ItemType>
bool Set<ItemType>::insert(const ItemType& value)
{
      // Fail if value already present

    Node* p = findFirstAtLeast(value);
    if (p != m_head  &&  p->m_value == value)
        return false;

      // Insert new Node preserving ascending order and incrementing m_size

    insertBefore(p, value);
    return true;
}

template <typename ItemType>
bool Set<ItemType>::erase(const ItemType& value)
{
      // Find the Node with the value, failing if there is none.

    Node* p = findFirstAtLeast(value);
    if (p == m_head  ||  p->m_value != value)
        return false;
```

```
      // Erase the Node, decrementing m_size
    doErase(p);
    return true;
}

template <typename ItemType>
bool Set<ItemType>::get(int i, ItemType& value) const
{
    if (i < 0  ||  i >= m_size)
        return false;

      // Return the value at position i.  Since the values are stored in
      // ascending order, the value at position i will be greater than
      // exactly i items in the set, meeting get's specification.

      // If i is closer to the head of the list, go forward to reach that
      // position; otherwise, start from tail and go backward.

    Node* p;
    if (i < m_size / 2)  // closer to head
    {
        p = m_head->m_next;
        for (int k = 0; k != i; k++)
            p = p->m_next;
    }
    else  // closer to tail
    {
        p = m_head->m_prev;
        for (int k = m_size-1; k != i; k--)
            p = p->m_prev;
    }

    value = p->m_value;
    return true;
}

template <typename ItemType>
void Set<ItemType>::swap(Set<ItemType>& other)
{
      // Swap head pointers

    Node* p = other.m_head;
    other.m_head = m_head;
    m_head = p;

      // Swap sizes

    int s = other.m_size;
    other.m_size = m_size;
    m_size = s;
}

template <typename ItemType>
Set<ItemType>::~Set()
```

```
{
      // Delete all Nodes from first non-dummy up to but not including
      // the dummy

    while (m_head->m_next != m_head)
        doErase(m_head->m_next);

      // delete the dummy

    delete m_head;
}

template <typename ItemType>
Set<ItemType>::Set(const Set<ItemType>& other)
{
    createEmpty();

      // Copy all non-dummy other Nodes.  (This will set m_size.)
      // Inserting each new node before the dummy node that m_head points to
      // puts the new node at the end of the list.

    for (Node* p = other.m_head->m_next; p != other.m_head; p = p->m_next)
        insertBefore(m_head, p->m_value);
}

template <typename ItemType>
Set<ItemType>& Set<ItemType>::operator=(const Set<ItemType>& rhs)
{
    if (this != &rhs)
    {
          // Copy and swap idiom

        Set<ItemType> temp(rhs);
        swap(temp);
    }
    return *this;
}

template <typename ItemType>
void Set<ItemType>::createEmpty()
{
    m_size = 0;

      // Create dummy node

    m_head = new Node;
    m_head->m_next = m_head;
    m_head->m_prev = m_head;
}

template <typename ItemType>
void Set<ItemType>::insertBefore(Node* p, const ItemType& value)
{
      // Create a new node
```

```
    Node* newp = new Node;
    newp->m_value = value;

      // Insert new item before p

    newp->m_prev = p->m_prev;
    newp->m_next = p;
    newp->m_prev->m_next = newp;
    newp->m_next->m_prev = newp;

    m_size++;
}

template <typename ItemType>
void Set<ItemType>::doErase(Node* p)
{
      // Unlink p from the list and destroy it

    p->m_prev->m_next = p->m_next;
    p->m_next->m_prev = p->m_prev;
    delete p;

    m_size--;
}

template <typename ItemType>
typename Set<ItemType>::Node* Set<ItemType>::findFirstAtLeast(const ItemType&
value) const
{
      // Walk through the list looking for a match

    Node* p = m_head->m_next;
    for ( ; p != m_head  &&  p->m_value < value; p = p->m_next)
        ;
    return p;
}

template <typename ItemType>
void unite(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>&
result)
{
      // Check for aliasing to get correct behavior or better performance:
      // If result is s1 and s2, result already is the union.
      // If result is s1, insert s2's elements into result.
      // If result is s2, insert s1's elements into result.
      // If result is a distinct set, assign it s1's contents, then
      //   insert s2's elements in result, unless s2 is s1, in which
      //   case result now already is the union.

    const Set<ItemType>* sp = &s2;
    if (&result == &s1)
    {
        if (&result == &s2)
```

```
                return;
    }
    else if (&result == &s2)
        sp = &s1;
    else
    {
        result = s1;
        if (&s1 == &s2)
            return;
    }
    for (int k = 0; k < sp->size(); k++)
    {
        ItemType v;
        sp->get(k, v);
        result.insert(v);
    }
}

template <typename ItemType>
void butNot(const Set<ItemType>& s1, const Set<ItemType>& s2, Set<ItemType>&
result)
{
      // Guard against the case that result is an alias for s2 by copying
      // s2 to a local variable.  This implementation needs no precaution
      // against result being an alias for s1.

    Set<ItemType> s2copy(s2);
    result = s1;
    for (int k = 0; k < s2copy.size(); k++)
    {
        ItemType v;
        s2copy.get(k, v);
        result.erase(v);
    }
}

#endif // SET_INCLUDED
```

You could alternatively have implemented the member functions inside the class declaration:

```
template <typename ItemType>
class Set
{
    ...
    int size() const
    {
        return m_size;
    }
    ...
};
```

Since for this problem you were already starting with a lot of code written with the implementations outside the class declaration, it was less work to leave their implementations outside.

Problem 3:

The instantiation of Set::insert calls Set::findFirstAtLeast, which contains the expression p->m_value < value, where both operands are Coord. Also, Set::insert contains the expression p->m_value == value, where both operands are Coord. We never defined operator< or operator== for Coord operands (nor should we for operator<, since coordinates don't have a natural ordering).

Problem 4:

```
void listAll(const File* f, string path)
{
    if (f->files() == nullptr)
        cout << path << endl;
    else
    {
        path += '/';
        cout << path << endl;
        const vector<File*>& files = *(f->files());
        for (size_t k = 0; k < files.size(); k++)
            listAll(path + files[k]->name(), files[k]);
    }
}
```

Other ways to write the for loop are

```
    for (vector<File*>::const_iterator p = f->files()->begin();
                                           p != f->files()->end(); p++)
        listAll(path + (*p)->name(), *p);
```

or (in C++11 or later)

```
    for (File* subfile : *(f->files()))  // or  for (auto subfile : *(f->files()))
        listAll(path + subfile->name(), subfile);
```

Without any static or global variables or any additional containers, there would be no way to keep track of the path from the root to each node of the tree.

Problem 5:

Consider the code in the k loop:

```
        for (int k = 0; k < N; k++)
        {
            if (k == i  ||  k == j)
                continue;
            int d = dist[i][k] + dist[k][j];
            if (d < minDist)
            {
                minDist = d;
                bestMidPoint[i][j] = k;
            }
        }
```

This involves one initialization (int k = 0), which we can ignore, since it's dominated by the N repetitions of everything else. The most work that each of the N iterations of the loop might do is a comparison (k < N), an increment (k++), two equality tests (k == i and k == j), an addition, a less than test, 2 integer assignments, and 3 double subscriptings. These basic operations are each constant time, so the number of operations performed during one execution of the innermost loop body is bounded by some constant m.

For the rest of this analysis, we won't be so meticulous: we'll drop low order terms where it won't affect the result.

That innermost loop body obviously accounts for most of the operations of the whole algorithm, since it's executed the most and the rest of the algorithm has only basic operations. In all, that body accounts for sum(i from 0 to N-1) of sum(j from 0 to N-1) of sum(k from 0 to N-1) of m operations.

sum(i from 0 to N-1) of {sum(j from 0 to N-1) of [sum(k from 0 to N-1) of m]} ~ m * sum(i from 0 to N-1) of {sum(j from 0 to N-1) of [sum(k from 0 to N-1) of 1]} ~ m * sum(i from 0 to N-1) of {sum(j from 0 to N-1) of N} ~ m * sum(i from 0 to N-1) of N*N* ~ *m* * *N*N*N = O(N3)

Again, the innermost statement accounts for the most operations performed, and it accounts for sum(i from 0 to N-1) of sum(j from 0 to i-1) of sum(k from 0 to N-1) of m operations. We'll retain the constant of proportionality just to get a feel for how much faster this can be. (We assume m is about the same as Part a.)

sum(i from 0 to N-1) of {sum(j from 0 to i-1) of [sum(k from 0 to N-1) of m]} ~ m * sum(i from 0 to N-1) of {sum(j from 0 to i-1) of [sum(k from 0 to N-1) of 1]} ~ m * sum(i from 0 to N-1) of {sum(j from 0 to i-1) of N} ~ m * sum(i from 0 to N-1) of i*N} ~ m * N * sum(i from 0 to N-1) of i} ~ m * N * ((N-1) * N / 2)

Since ((N-1) * N / 2) ~ N*N/2, the full sum is about (m/2)* *N*N*N = O(N3). For large N, this algorithm is about twice as fast as the one in Part a, but it's still order N3; doubling the size of a problem increases the running time about eightfold.

Problem 6:

First, notice that our implementation of get takes a length of time proportional to the distance between the desired position and the nearest end of the list. For position k in a list of size N, this is min(k, N-k).

The statements in the k loop are executed N times. Each time, the get function visits min(k,N-k) nodes (which is bounded by N/2, which is O(N)), and insert visits O(N) nodes (since it checks for duplicates), so on each iteration, the body of the loop visits O(N) nodes. N * O(N) = O(N2). Possibly assigning set1 to result in the else branch before the loop visits only O(N) nodes; the rest of that if statement is constant time. Thus, the running time is dominated by the loop: unite is O(N2).

Copying the items into v is O(N). Sorting v is O(N log N). Deleting the original result nodes is O(N). (Notice that each call to doErase is O(1).) Copying the nonduplicate items from v is O(N). (Notice that each call to insertBefore is O(1).) Destroying v is O(N). Since O(N log N) dominates O(N), this version of unite is O(N log N).

The else branch in the initial if statement assigns set1 to *this, which is O(N); everything else in that if statement is constant time. The while loop visits the N items in set1 and the N items in set2; the loop body is constant time, so the loop is O(N). The for loop is O(N). Thus, this version of unite is O(N).

Problem 7:

Changes to the program as given are bold.

```
…
inline
bool compareStudentPtr(const Student* lhs, const Student* rhs)
{
    return compareStudent(*lhs, *rhs);
}
…
void insertion_sort(vector<Student>& s, bool comp(const Student&, const Student&))
{
    for (size_t k = 1; k < s.size(); k++)
    {
        Student currentStudent(s[k]);
        size_t m = k;
        for ( ; m > 0  &&  comp(currentStudent, s[m-1]); m--)
            s[m] = s[m-1];
        s[m] = currentStudent;
    }
}
…
      // Create a auxiliary copy of students, to faciliate the later reordering.
    vector<Student> auxStudents(students.begin(), students.end());

      // Create a vector of Student pointers, and set each pointer
      // to point to the corresponding Student in auxStudents.
    vector<Student*> studentPtrs;
    for (size_t k = 0; k < auxStudents.size(); k++)
        studentPtrs.push_back(&auxStudents[k]);

      // Sort the vector of pointers using the STL sort algorithm
      // with the comp parameter as the ordering relationship.
    sort(studentPtrs.begin(), studentPtrs.end(), comp);
```

```
      // Using the now-sorted vector of pointers, replace each Student
      // in students with the Students from auxStudents in the correct order.
    for (size_t k = 0; k < studentPtrs.size(); k++)
        students[k] = *studentPtrs[k];

      // auxStudents will be destroyed upon return from the function
…
```