Section Handout #7

Problem 1: Client-side vs. Implementation-side
Consider a basic Stack class built on top of a Vector:

```
template <typename Type>
class Stack {
      public:
            Stack();
            ~Stack();

            void push(Type elem);
            Type pop();

            bool isEmpty();

      private:
            Vector<Type> elems;
}
```

a) Say we as the implementers want to add a method that will reverse the elements that are currently in the Stack. How can we do it?

b) Now write another Reverse function, this time as a client of the stack class.

Problem 2: Template Class Conversion

A Mob is a special kind of queue; instead of dequeing the element that has been in the queue the longest, it dequeues a randomly selected element from anywhere in the queue. Suppose Mob is defined (partially) as follows:

(Mob.h):

```
class Mob {
      public:
            Mob();
            void enqueue(int newElem);
            int dequeue();
            int size();
      private:
            Vector<int> elems;
};
```

(Mob.cpp):

```
void Mob::enqueue(int newElem)
{
      elems.add(newElem);
}
```

```
int Mob::dequeue()
{
	int elemNum = RandomInteger(0, elems.size()-1);
	int value = elems[elemNum];
	elems.removeAt(elemNum);
	return value;
}
```

What would you need to do to change this version of Mob (which stores ints) into a templatized version? Rewrite the above code to match this version, and describe anything else you would need to do to get it running.

Problem 3: Big-O Detective

In this problem, you have a templatized Set class, but you don't know anything about its implementation. You would like to determine the Big-O for the Set's contains method using some kind of testing procedure.

a) You've seen how to use execution timing to infer a function's Big-O. If you can determine the amount of time it takes a contains call to complete, how could you use this information to determine the Set's contains function's Big-O? Will it make a difference if you ask about elements that you know to be contained in the set versus ones that you know aren't and, if it makes a difference, would one be better for this purpose?

b) Now you'd like to determine the Big-O a different way: you'd like to count the number of comparisons the contains function needs to use to determine if an element is contained. How could you accomplish this? (Hint: this may require using something which is normally considered bad style.)

Problem 4: Deques

A deque (pronounced “deck”) is a double-ended queue where items can be inserted and removed from both ends of the deque. This structure can be used to simulate both a stack and a queue, and is useful for array operations where we only want to insert at the beginning or end of a list of items. How would we implement a deque efficiently with linked lists? Would it be preferable to use a singly-linked or doubly-linked list? What functions would it need to implement? How could it be used to simulate a stack or a queue? Note that you should not write any code for this problem. Instead, answer the questions above and sketch out the general form of the implementation, but do not implement it.

The lists in problems 5 and 6 use the structure definition:

```
struct Node
{
	int value;
	Node* next;
```

```
};
```
These linked list problems will hopefully get you thinking about how to manipulate this data structure; you will need these skills for your current assignment.

Problem 5: Stutter

Write a function `Stutter` that given a linked list will duplicate every cell in the list. If the incoming list contains the elements `(11 5 21 7 7)`, the function will destructively modify the list to contain `(11 11 5 5 21 21 7 7 7 7)`.

Problem 6: RemoveDuplicates (a.k.a. Unstutter)

Write a function `RemoveDuplicates` that given a linked list will remove any **neighboring** duplicates found in the list. If the incoming list is `(5 5 22 37 89 89 15 15 22)` the function will destructively modify the list to contain `(5 22 37 89 15 22)`. Don't worry about handling duplicate sequences longer than 2 or duplicates that aren't right next to each other in the list.