1.What is the base case, and can it be solved?
2.What is the general case?
3.Does the recursive call make the problem smaller and does it approach the base case?

```
template <class T> Node<T>* FindSumStart(Node<T>* n) {
  if (n == NULL) {
    return NULL;
  }
  int total = 0;
  Node<T>* tmp = n;
  while (tmp != NULL) {
    if (total == tmp->value) {
      return n;
    }
    total += tmp->value;
    tmp = tmp->next;
  }
  return FindSumStart(n->next);
}
```

```
35 30 28
5 7 30 28
5 7 2 15 28
5 7 2 3 5 28
5 7 2 3 5 2 14
5 7 2 3 5 2 2 7
5 7 2 3 5 2 2 7
```

Solution:

```
bool Factor(Node* &head, Node* &tail, Node* n) {
  // base case
  if (n == NULL) return false;
  // see if this element has any factors
  for (int i = 2; i < n->value; i++) {
    if (n->value % i == 0) {
      // create a new node in front of this one
      Node* tmp = new Node(i);
      // change all of the links
      tmp->prev = n->prev;
      if (n->prev != NULL) {
        tmp->prev->next = tmp;
      }
      tmp->next = n;
      n->prev = tmp;
      n->value = n->value / i;
      // handle the special case of the first node
      if (n == head) head = tmp;
      return true;
} }
  // recurse if we couldn't split this element
  return Factor(head,tail,n->next);
}
// driver function
bool Factor(Node* &head, Node* &tail) {
  return Factor(head,tail,head);
}
```

Now write the constructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

Solution:

```
template <class T> Stairs<T>::Stairs(int s, const T& val) {
  size = s;
  data = new T*[s];
  for (int i = 0; i < s; i++) {
    data[i] = new T[i+1];
    for (int j = 0; j <= i; j++) {
      data[i][j] = val;
    }
  }
}
```

```
template<>
MyArray<T>::operator=( const MyArray& rhs ) {
    if( this != &rhs ) {
        delete [] pElements;
        pElements = new T[ rhs.numElements ];
        for( size_t i = 0; i < rhs.numElements; ++i )
            pElements[ i ] = rhs.pElements[ i ];
        numElements = rhs.numElements;
    }
    return *this;
}
```

```
void Student::copy(const Student& s) {
  courses_per_term = s.courses_per_term;
  num_terms = s.num_terms;
  initialize();
  for (int i = 0; i < num_terms; i++) {
    for (int j = 0; j < courses_per_term; j++) {
      data[i][j] = s.data[i][j];
    }
  }
}
```

Now write the destructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

Solution:

```
template <class T> Stairs<T>::~Stairs() {
  for (int i = 0; i < size; i++) {
    delete [] data[i];
  }
  delete [] data;
}
```

Lists:

Insert – adds a value to the position before the iterator:

mylist = 1, 2, 3, 4, 5 and itr points to 2

mylist.insert (itr, 10)

mylist = 1, 10, 2, 3, 4, 5

mylist.insert (itr, 2, 20)

mylist = 1, 10, 20, 20, 2, 3, 4, 5

Erase – Erases a value pointed to by an iterator and returns a pointer to the next element:

mylist = 1, 2, 3, 4, 5 and itr points to 2

itr = mylist.erase (itr, 2)

mylist = 1, 3, 4, 5 and itr = 3

Iterators:

vector<data type>::iterator <iterator name> = vec.begin()

list<data type>::iterator <iterator name> = lst.begin()

string::iterator <iterator name> = str.begin()

- If the list/vector/string is a constant, then use a const_iterator.
- If you want to go backwards, use reverse_iterator and rbegin() and rend().

Orders of Magnitude

$O(1)$ – *CONSTANT* – number of operations is independent of size. (Computations, inserting/erasing in a list)

$O(\log n)$ - *LOGARITHMIC* – dictionary lookup or binary search.

$O(n)$ – *LINEAR* – searching through a list, erasing in a vector. (*for* statements)

$O(n \log n)$ – sorting a vector or list.

$O(n^2)$, $O(n^3)$, $O(n^k)$ – *POLYNOMIAL* – finding the closest pair of points in a list. Nested for statements.

$O(2^n)$, $O(k^n)$ – EXPONENTIAL – Fibonacci, chess, Recursion.

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Assignment / Copy / Destructor

Assignment operator –

<class> <class>::operator=(const <class>& <class object>) {}

Uses the = to assign one class object to another. Uses a copy function.

For example: *MyClass c1, c2;*

c1 = c2; // assigns c2 to c1

Copy constructor - MyClass(*const* MyClass& other)

Is similar to a regular class constructor, but you are creating a new class object from an existing one. Will call the copy function.

Destructor - ~MyClass(); Will use delete commands to erase all data associated with the class object. Called automatically once the object goes out of scope.

Recursion: examine frames of the stack

Dr. Memory for leaks (takes a long time to run

Backtrace to find crash

Gdb variable values

Watchpoint to see semantic errors.

List<int>::const_iterator it1 = lst.begin()

List<int>::iterator it2 = lst.begin()

Rank these 6 order notation formula from fastest(1) to slowest(6).

Solution: 1 $O(8 \cdot s \cdot w \cdot h)$

Solution: 4 $O((s \cdot w \cdot h)^8)$

Solution: 6 $O((8 \cdot w \cdot h)^s)$

Solution: 5 $O(w \cdot h \cdot 8^s)$

Solution: 2 or 3 $O((s + w \cdot h)^8)$

Solution: 2 or 3 $O(w \cdot h \cdot s^8)$

NOTE: The ordering of the '2' vs. '3' depends on the relative size of the variables h, w, and s.

If $w = h = s$ **:** $(w + w \cdot w)^8 = w^{16} > w \cdot w \cdot w^8 = w^{10}$.

If $w = h$ & $s = w^2$ **:** $(w^2 + w \cdot w)^8 = w^{16} < w \cdot w \cdot (w^2)^8 = w^{18}$.