Read for next time: TCS 11

More on Vectors:

To use vectors, you have to #include <vector>
because <vector> is one of the Standard Template Libraries (STLs).

Then you can declare vectors of any base type or element type (e.g., vectors of doubles, vectors of ints, vectors of Bools, vectors of strings, vectors of Times [or any user-defined struct], vectors of vectors of doubles [i.e., 2D matrices]). This means that when you declare a vector, you have to state its base type or element type:

```
vector<int> count; // makes count a vector of ints
vector<double> doubleVector; // a vector of doubles
```

The preceding defines these vectors to have 0 length (no elements), which is not very useful. However you can use push_back to add things to the end (back) of a vector and make it bigger in this way.

If you know how big the vector needs to be, then you can declare it to have that size (or dimension):

```
vector<int> count (4); // makes count a 4-element vector
// which is indexed count[0], … , count[3]
```

The initial size is 4, but you can use push_back to make it bigger.

You can give a vector initial values just like you can give initial values to other

variables.

vector<int> count (4, 0); // initializes the 4 elements to 0

You can declare a vector to be a copy of another vector:

vector<int> fred (count); // this makes fred a copy of count
vector<int> alice = count; // this makes alice a copy of count

The above are all different way of constructing a vector. We will find that when we define our own data types (classes), we can define constructors for them (ways of creating and initializing objects belonging to that type or class).

Bottom-up Development
We put together simple components to make more complex and more useful bigger components. And we can put these together to make even bigger ones, and so forth.

For both top-down and bottom-up programming: you cannot apply them blindly; you have to have some understanding of where you are going.

This chapter gives you an example of bottom-up programming: doing some interesting things with vectors of random numbers (in particular to see how good the pseudo-random number generator is).

Analyzing the program:
Consider the outer loop (in the main program):

```
for (int i = 0; i<upperBound; i++) {
  cout << i << '\t' << howMany (Vector, i) << endl;
```

```
  }
```

We execute the above loop upperBound times.
But what does this loop involve? It contains a call to howMany. How much time does that take?

```
int howMany (const vector<int>& vec, int value) {
  int count = 0;
  for (int i=0; i< vec.size(); i++) {
    if (vec[i] == value) count++;
  }
  return count;
}
```

The amount of time is proportional to the length of the vector.
So howMany (Vector, i) takes time proportional to the length of Vector, which was declared to be of length numValues.

The time to execute this program is proportional to:
  upperBound * numValues

Letting *M* = upperBound and *N* = numValues (shorter names),
then the running time is *O*(*MN*) = "order *MN*," which means that in the long run (big *M*, *N*), the time is proportional to *MN*. This is called *asymptotic analysis*.

You can analyze time, space, or any other resource in this way, to find out how much the algorithm will use when things get big.

However, if you think about it, you will see that if your goal is to count the frequency of all the numbers, you only have to make one pass through the vector. But you have to keep a histogram as you do it.