Data Structure Scaling, Reading

"Scaling" refers to how well a solution or approach performs when the data set it uses becomes large. We ran into issues of scaling in Module 5, where we applied our programming skills to a problem involving "big data". If that DVC scheduling history database had consisted of only 100 records (instead of 70,000+), we would never have noticed that **our original solution was inefficient**. But using the full set and waiting long minutes for our first lab assignment solution to complete, led us to a clever solution to cut down on the runtime. There are even more efficient solutions than this, and we'll get to that in later modules of this course.

What led us to the solution was a calculation of how many comparisons were taking place in the solution. That calculation involved averaging, estimations, and **the number of records**. That was a *deliberate* attempt to improve runtime.

In Module 6 on stacks and queues -- in the linked list implementations -- we actually had two *fortunate* decisions that improved timing performance, without realizing what we did. It just sort of happened naturally. But think back -- why did we add a "siz" data member? We could have simply let the **Stack::size** and **Queue::size** functions traverse the list and *count* the number of nodes instead.

```
template <class V>
int Stack<V>::size() const
{
  int siz = 0;
  for (Node* p = firstNode; p; p = p->next, siz++);
  return siz;
}
```

And we really did *not* need a private data member for the tail pointer in **Queue::push**. All we had to do to find the last node was to traverse the list. Like this:

```
template <typename V>
void Queue<V>::push(const V& value)
{
  Node* p, *prev; // prev will end up pointing to the last node
  for (p = firstNode, prev = 0; p; prev = p, p = p->next);

  Node* temp = new Node;
  temp->value = value;
  temp->next = 0;
  if (prev) prev->next = temp;
  else firstNode = temp;
  ++siz;
}
```

Maybe we did not give this much conscious thought, but more likely it just naturally made sense to track instead of count. And in terms of big data, look at the number of cycles these loops would have to do for data structures of 70,000 values, just to report back the size or to push one new value.

If that was *not* a case of naturally choosing the more efficient way of writing these functions, and you really *did* wonder why we didn't just traverse and thereby avoid adding the overhead of more private data members, then it's time to start taking "*scaling*" into consideration. Develop and test with small data sets, but think about how your solutions would work given large data sets that's what *scaling* is.

Scaling And Collegiate Programming Competitions

DVC's programming students annually participate in competitions with other colleges and universities in our region of the country. Teams of students are judged on how many problems they solve, and how fast they solve them.

These competitions are *famous* for posing certain types of problems and providing tiny data sets for use in developing and testing a solution, but in the fine print they'll say something like "for data sets of up to 100,000 records" or something like that. They lure unsuspecting teams into writing solutions that scale poorly. Teams submitting their solutions end up disappointed and confused when the judges return their work with the feedback: "runtime limit exceeded". Does it sound like what happened to us in Module 5?

Doing The Math

We'll learn to write more efficient solutions that scale well, using two different techniques. The first is to be more aware of ways to avoid loops so that the decisions on **Stack::size** and **Queue::push** really are not just fortunate, but reasoned. The second is to study techniques for counting the numbers of operations -- comparisons, assignments, etc -- and expressing them in terms of the size of our data set, in a way that expresses how it scales with size. That's what this module is about.