In most of our discussions of data structures, the values being stored will simply be integers. It is important to recognize that in actual applications, data items usually consist of a collection of attributes of an object (for example, all the information regarding a book in a library, or all the information about a particular pharmaceutical product). Normally one attribute is recognized as being an unique identifier of the object (for example, the ISBN of a book) – we usually call this the key. This is what is represented by the simple integers that we will store in our structures. We will not often concern ourselves with the other attributes of the data objects, but we should keep their existence in the back of our minds.

Since we now recognize that in order to decide on the best representation for our data we need to know what operations we will be performing, it behooves us to consider some of the common operations on a set of values or items. These can be grouped into 2 categories:

Operations related to a single item:

- add an item to the set
- remove an item from the set (these first two operations do not necessarily go together: some data sets never grow, and some never shrink)
- attach extra data to an item (such as adding a new email address to a person in your contact set)
- find the successor of an item (that is, find the one that would come next in sorted order)
- find the predecessor of an item
- search for a particular item (this has two forms: "Is x in the set?" and "What is x's location in the set?")
Operations related to the entire set:

- combine two (or more) sets into a single set (A brief excursion into this topic, to illustrate how the choice of data structure can affect the amount of work required to perform an operation. If the sets are stored in arrays, combining two sets requires copying the items in one array to empty spaces in the other array, if possible. If there is insufficient empty space, a new array must be allocated and all the items must be copied into the new array. On the other hand if the sets are stored in linked lists, the combined set is created by appending one list to the end of the other, which is accomplished in a single step.)

- sort the set

- find the max and/or min element in the set

- perform a range query on the set (find all elements $x$ such that $a \leq x \leq b$, for some specified $a$ and $b$ values)

- find a subset of the set based on one or more attributes (such as "find all red sports cars in the "Vehicles for sale" set)

One goal of this course is to give you an understanding of many of the most important data structures and their strengths and weaknesses, so that when implementing an algorithm you can choose a data structure that is well-suited to the problem.

What criterion should we use to choose an appropriate data structure for an application?

How about "I already understand data structure A, and I don't understand data structure B"? .... umm, no.

Perhaps "There is a built-in module for data structure A, but I would have to code data structure B myself"? .... fail!

Or “I can code A in 5 minutes, but B would take an hour” ... nope, that’s not a good reason.

What could be left? What could be right?

The answer is computational complexity. We will prefer structure A to structure B if A has a
lower order of complexity for the operations we need in our particular application.

Before we discuss computational complexity, we need to clarify which operations can be completed in constant time.

We assume that all fundamental operations:
- +, -, *, / and comparisons for integers and floating point numbers
- comparisons on Booleans
- comparisons and type conversions on characters
- execution control
- accessing a memory address
- assigning a value to a variable

take constant time

It is important to note that this model implies an upper limit on the number of digits in any number. This is true of virtually all programming languages.

This model does not assume constant time operations on strings. A string is considered to be a data structure consisting of a sequence of characters.

I expect we have all seen "big O" complexity classification (since it has been covered in other courses), but we will review the ideas anyway.

To determine the “timing function” for an algorithm we count the fundamental operations as a function of the size of the input. But when we do this, we usually just count the operations that involve the actual data. In other words we ignore things like index variables and execution control operations. As we will see, we don’t even need to be completely precise in our counting.
Consider this algorithm, which is written in pseudo-code that I just made up. Notice that I’m leaving out all declarations.

**CODE**

A1:  

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = read()</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>(1 I/O and 1 assignment)</td>
</tr>
</tbody>
</table>

for i = 1 to n:

| A[i] = read() |
| 2*n           |
| (1 I/O and 1 assignment, repeated n times) |

We don’t count any of the operations relating to the loop management because they don’t involve the data.

So we would write the timing function for A1 as $T_{A1}(n) = 2n + 2$

(Note for purists: the size of the input here is actually $n + 1$ since that is the total number of read actions we execute. For our purposes here, calling it $n$ is fine.)

Now two more simple algorithms:

**CODE**

A2:  

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = read()</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>(1 I/O and 1 assignment)</td>
</tr>
</tbody>
</table>

for i = 1 to n:

| A[i] = read() |
| 2*n           |
| (1 I/O and 1 assignment, repeated n times) |

for i = 1 to n:

| for j = 1 to n: |
| 2*n^2         |
| (2 ops, n^2 times) |

So we would write the timing function for A2 as $T_{A2}(n) = 2n^2 + 2n + 2$
CODE
A3: n = read()
    for i = 1 to n:
        A[i] = read()
        B[i] = 2*A[i]

OPERATIONS
2 (1 I/O and 1 assignment)
2*n (1 I/O and 1 assignment, repeated n times)
2*n (1 I/O and 1 assignment, repeated n times)

So we would write the timing function for A3 as $T_{A3}(n) = 4n + 2$

Our goal is to use the timing functions as a way of comparing the efficiency of algorithms. But as we have already seen, they are somewhat approximate because they don’t count every single operation. So instead of comparing the explicit timing functions for different algorithms, we use the timing functions to collect algorithms into groups. Then to compare two algorithms, we compare the groups they are assigned to.

We group algorithms together based on the growth-rate of their timing functions. To illustrate this we can look at the three algorithms above and see what happens when we repeatedly double the value of n (i.e. double the size of the input).

<table>
<thead>
<tr>
<th>n</th>
<th>$T_{A1}(n)$</th>
<th>$T_{A2}(n)$</th>
<th>$T_{A3}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>42</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>146</td>
<td>34</td>
</tr>
<tr>
<td>16</td>
<td>34</td>
<td>546</td>
<td>66</td>
</tr>
</tbody>
</table>

Etc.

If we look at the growth-rates of these, we can see that when we double n, $T_{A1}(n)$ and $T_{A3}(n)$ roughly double, while $T_{A2}(n)$ goes up by a factor of about 4. It seems like A1 and A3 have similar behaviour and A2 grows faster ... but can we quantify that?

I’m going make some claims that may seem to come out of nowhere. A bit further along, we will see how I derived them:

Claim: $\forall n \geq 2, T_{A1}(n) \leq 3n$
Claim: $\forall n \geq 4, T_{A2}(n) \leq 3n^2$
Claim: $\forall n \geq 2, T_{A3}(n) \leq 5n$
We can see that these claims are true for the values of n in the table ... it turns out they are true for all other values that meet the conditions as well (we’ll see why that’s true in a minute).

Let’s focus on A1 and A3. The fact that \( T_{A1}(n) \leq 3n \) means that \( T_{A1} \) can’t grow any faster than \( 3n \) grows. Similarly, \( T_{A3} \) can’t grow any faster than \( 5n \) grows. And here’s the kicker: \( 3n \) and \( 5n \) grow at exactly the same rate. They both have the property that if \( n \) increases by a factor of \( k \), then the value of \( 3n \) and \( 5n \) both also increase by a factor of \( k \).

But what about A2? The claim I made above tells us that \( T_{A2} \) can’t grow any faster than \( 3n^2 \) grows. Suppose we had some other algorithm A4 where \( T_{A4}(n) \leq 14n^2 \). We see the same thing as in the last paragraph: \( 3n^2 \) and \( 14n^2 \) both grow at exactly the same rate: if \( n \) increases by a factor of \( k \), then the values of \( 3n^2 \) and \( 14n^2 \) both increase by a factor of \( k^2 \).

It looks like we are on firm ground if we say that \( T_{A1} \) and \( T_{A3} \) grow at the same rate, and \( T_{A2} \) grows at a faster rate. But we need to be careful – can we be sure that there are no values \( n_0 \) and \( c \) such that \( \forall n \geq n_0, T_{A2}(n) \leq c \times n \) ? If such values exist then we would have to group A2 together with A1 and A3.

Suppose such values do exist. Then we would have
\[
\forall n \geq n_0, 2n^2 + 2n + 2 \leq c \times n
\]
\[
\Rightarrow \quad \forall n \geq n_0, n^2 \leq \frac{(c - 2)n}{2} - 1
\]
\[
\Rightarrow \quad \forall n \geq n_0, 1 \leq \frac{c - 2}{2n} - \frac{1}{n^2}
\]
But when \( n \) gets sufficiently large, both terms on the right hand side will be < 1, giving a contradiction. Therefore these \( n_0 \) and \( c \) values do not exist. This proves that \( T_{A2} \) grows faster than either of the others.

**** This is actually as far as we got on 20180112 ... but I’m including the rest of the topic. We will cover the following material in class on 20180115 ****
Was there anything particular about the timing functions that we used? Not really.

Suppose an algorithm A has timing function

$$T_A(n) = a_t \times n^t + a_{t-1} \times n^{t-1} + \cdots + a_1 \times n + a_0$$

where the $a_i$ values are constants.

Claim: $\exists n_0$ such that $\forall n \geq n_0, n^t \geq a_{t-1} \times n^{t-1} + \cdots + a_1 \times n + a_0$

Proof: Suppose not. Then $\forall n, n^t < a_{t-1} \times n^{t-1} + \cdots + a_1 \times n + a_0$

$$\Rightarrow 1 < \frac{a_{t-1}}{n} + \cdots + \frac{a_1}{n^{t-1}} + \frac{a_0}{n^t}$$

As n increases, each positive term in the sum on the right gets smaller, and in fact gets arbitrarily close to 0. Thus there is a value of $n$ for which each term in the sum is $< \frac{1}{t}$. For this value of $n$ the sum on the right hand side is $< 1$... which is a contradiction. Therefore such an $n_0$ exists.

$$\Rightarrow \forall n \geq n_0, \quad T_A(n) \leq a_t \times n^t + n^t$$

ie $\forall n \geq n_0, \quad T_A(n) \leq (a_t + 1) \times n^t$

This proves the existence of appropriate values of $n_0$ and c. For our analysis, existence is all we need.

**Big O Classification**

Let $f(n)$ and $g(n)$ be non-negative valued functions on the set of non-negative numbers. If there are constants $c$ and $n_0$ such that $f(n) \leq c \times g(n)$ $\forall n \geq n_0$ then we say $f(n) \in O(g(n))$.

In other words, the growth-rate of $g(n)$ is an upper bound on the growth-rate of $f(n)$.

$O(g(n))$ represents the set of all functions whose growth-rate is $\leq$ the growth-rate of $g(n)$. 
There are several complexity classes that we encounter frequently. Here is a table listing the most common ones.

<table>
<thead>
<tr>
<th>Dominant Term</th>
<th>Big-O class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$ (a constant)</td>
<td>$O(1)$</td>
<td>constant time</td>
</tr>
<tr>
<td>$c \times \log n$</td>
<td>$O(\log n)$</td>
<td>logarithmic  time</td>
</tr>
<tr>
<td>$c \times n$</td>
<td>$O(n)$</td>
<td>linear time</td>
</tr>
<tr>
<td>$c \times n \times \log n$</td>
<td>$O(n \times \log n)$</td>
<td>$n \log n$ time</td>
</tr>
<tr>
<td>$c \times n^2$</td>
<td>$O(n^2)$</td>
<td>quadratic time or $n^2$ time</td>
</tr>
<tr>
<td>$c \times n^3$</td>
<td>$O(n^3)$</td>
<td>cubic time or $n^3$ time</td>
</tr>
<tr>
<td>$c \times n^k \text{ Where } k \text{ is a constant}$</td>
<td>$O(n^k)$</td>
<td>polynomial time</td>
</tr>
<tr>
<td>$c \times k^n \text{ where } k \text{ is a constant }&gt; 1$</td>
<td>$O(k^n)$</td>
<td>exponential time</td>
</tr>
<tr>
<td>$c \times n!$</td>
<td>$O(n!)$</td>
<td>factorial time</td>
</tr>
</tbody>
</table>

**Combinations of Functions**

If $f_1(n) \in O(g_1(n))$, and $f_2(n) \in O(g_2(n))$ ....

then $f_1(n) + f_2(n) \in O(max(g_1(n), g_2(n)))$

and $f_1(n) \times f_2(n) \in O(g_1(n) \times g_2(n))$

So far this should all be very familiar. But O classification is just the small first step in the field of computational complexity. There are many other ways of grouping functions together based on the resources (time and/or space) they require. We will consider two more: **Omega** classification and **Theta** classification.
Omega Classification

Big O classification gives us an upper bound on the growth-rate of a function (that is, \( f(n) \in O(g(n)) \)) tells us that \( f(n) \) grows no faster than \( g(n) \) grows, but it doesn't tell us anything about a lower bound on the growth-rate of \( f(n) \).

Your first reaction to this observation might well be "why would we care about a lower bound on the growth-rate? We use this computational complexity stuff to measure the worst-case running time of an algorithm ... and for worst-case analysis, all we need is an upper bound."

Before we explain why lower-bound analysis is important, we will define exactly what we mean by it and how it works.

**Definition:** Let \( f(n) \) and \( g(n) \) be functions. If there exist constants \( c \) and \( n_0 \) with \( c > 0 \) such that
\[
   f(n) \geq c \times g(n) \quad \forall n \geq n_0
\]
then \( f(n) \in \Omega(g(n)) \)

(\( \Omega \) is the Greek letter "Omega")

Note that this is almost exactly the same as the definition of Big O except that the "\( \leq c \times g(n) \)" has become "\( \geq c \times g(n) \)"

As with Big O classification, we can see that \( \Omega(g(n)) \) is actually a class of functions, all of which grow at least as fast as \( g(n) \) grows. We can also see that there is a hierarchy of Omega classes, just as there is a hierarchy of Big O classes. For example, suppose \( f(n) \in \Omega(n^3) \). This means "growth-rate of \( f(n) \) \( \geq \) growth-rate of \( n^3 \)." But since "growth-rate of \( n^3 \) \( \geq \) growth rate of \( n^2 \), we can conclude that "growth rate of \( f(n) \) \( \geq \) growth rate of \( n^2 \)," which is equivalent to saying that \( f(n) \in \Omega(n^2) \).

In fact, if \( f(n) \in \Omega(n^k) \), then \( f(n) \in \Omega(n^i) \quad \forall i < k \).

(Note the parallel to Big O: if \( f(n) \in O(n^k) \), then \( f(n) \in O(n^i) \quad \forall i > k \))

When determining the Big O classification for \( f(n) \) we try to find the smallest function \( g(n) \) such that \( f(n) \in O(g(n)) \). Conversely, when determining the \( \Omega \) classification for \( f(n) \) we try to find the largest function \( g(n) \) such that \( f(n) \in \Omega(g(n)) \).

In class we did a couple of examples. Here's another:
Let $f(n) = 0.0001 \times n^2 + (10^6) \times n + 3$

We know that $f(n) \in O(n^2)$. It’s also very easy to see that $f(n) \in \Omega(n^2)$... we can let $c = 0.0001$ and it is immediately clear that $f(n) \geq c \times n^2 \quad \forall n \geq 0$.

Now is it possible that $f(n) \in \Omega(n^3)$?

If this were the case, then there would exist a positive constant $c$ such that

$$f(n) \geq c \times n^3 \quad \forall n \geq n_0$$

i.e.

$$0.0001 \times n^2 + (10^6) \times n + 3 \geq c \times n^3$$

$$3 \geq n \times (c \times n^2 - 0.0001 \times n - 10^6)$$

but we can easily see that this is impossible: even if $c$ is very small, as $n$ gets large there will come a point beyond which $c \times n^2 - 0.0001 \times n - 10^6$ is $\geq 1$ so $n \times (c \times n^2 - 0.0001 \times n - 10^6) \geq n$, which would give $3 \geq n \quad \forall n \geq n_0$... which is not possible.

Thus $f(n) \notin \Omega(n^3)$

This example illustrates a useful fact: if $f(n)$ is a polynomial, then the Big O class and the $\Omega$ class for $f(n)$ are identical.

But this is not always the case. For example, consider this function:

```python
A(n):
    if n % 2 == 0:
        for i = 1..n:
            print '*'
    else:
        for i = 1..n^2:
            print '*'
```

Let $f(n)$ be the time required to execute $A(n)$. If you plot $f(n)$ for $n = 1, 2, 3, ...$ you will see that it has a zig-zag shape. The tops of the zigs occur when $n$ is odd, and they grow at the
same rate as $n^2$. It is easy to see that $f(n) \in O(n^2)$. However, the bottoms of the zags, which occur when $n$ is even, do not show this behaviour - they grow at the same rate as $n$.

Referring back to our previous definitions, we are now able to say that $f(n) \in O(n^2)$ and also $f(n) \in \Omega(n)$... and neither of these can be improved: there is no lower $O$ class for $f(n)$, and no higher $\Omega$ class for $f(n)$.

This example demonstrates that an algorithm's Big O class may be different from its $\Omega$ class.

If we can show an algorithm's complexity is in $O(g(n)) \text{ and } \Omega(g(n))$, then we get very excited - it means that $g(n)$ gives both an upper and a lower bound on the growth-rate of the time required by the algorithm. Basically it means we know exactly how fast the algorithm's time requirement grows. This is so amazingly wonderful that we give it a special name:

**Theta Classification**

If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, we say $f(n) \in \Theta(g(n))$. 
