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. (In class I included declarations, but didn’t count the time for them.)

**CODE**

A1:  \[ n = \text{read()} \]
    \[
    \text{for } i = 1 \text{ to } n:\n    \quad A[i] = \text{read()}\n    \]

**OPERATIONS**

- \[ 2 \quad (1 \text{ I/O and 1 assignment}) \]
- \[ 2*n \quad (1 \text{ I/O and 1 assignment, repeated } n \text{ times}) \]

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:  \[ n = \text{read()} \]
    \[
    \text{for } i = 1 \text{ to } n:\n    \quad A[i] = \text{read()}\n    \]
    \[
    \text{for } i = 1 \text{ to } n:\n    \quad \text{for } j = 1 \text{ to } n:\n    \quad \quad \text{print } A[i] + A[j]\n    \]

**OPERATIONS**

- \[ 2 \quad (1 \text{ I/O and 1 assignment}) \]
- \[ 2*n \quad (1 \text{ I/O and 1 assignment, repeated } n \text{ times}) \]
- \[ 4*n^2 \quad (4 \text{ ops: 2 array references, 1 addition, 1 print, } n^2 \text{ times}) \]

So we would write the timing function for A2 as \[ T_{A2}(n) = 4n^2 + 2n + 2 \]
CODE
A3: \[ n = \text{read()} \]
\[ \text{for } i = 1 \text{ to } n:\]
\[ A[i] = \text{read()} \]
\[ B[i] = 2 \times A[i] \]

OPERATIONS
\[ 2 \]  (1 I/O and 1 assignment)
\[ 2n \]  (1 I/O and 1 assignment, repeated n times)
\[ 3n \]  (1 array reference, 1 multiplication, 1 assignment repeated n times)

So we would write the timing function for A3 as \( T_{A3}(n) = 5n + 2 \)

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 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>8</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>22</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>74</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>274</td>
<td>42</td>
</tr>
<tr>
<td>16</td>
<td>34</td>
<td>1058</td>
<td>82</td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How fast are these timing functions growing? Let’s look at the ratios for successive values in the columns. For A1, the ratios are \( \frac{6}{4}, \frac{10}{6}, \frac{18}{10}, \frac{34}{18} \) etc. We can see that these ratios are getting closer and closer to 2 ... can you see why they will never quite reach 2?

For A3, the sequence of ratios is similar: \( \frac{12}{7}, \frac{22}{12}, \frac{42}{22}, \frac{82}{42} \) etc. - again, the ratios approach 2.
For A2, the sequence of ratios is \( \frac{22}{8}, \frac{74}{22}, \frac{274}{74}, \frac{1058}{274} \) ... perhaps it’s a bit harder to see the pattern.

The ratios work out to (approximately) 2.75, 3.4, 3.7, 3.9 ... and if we went further, we would see that the ratios approach 4 but never quite reach it.

So when we double the size of the input (ie. the size of the input increases by a factor of 2), \( T_{A1} \) and \( T_{A3} \) also increase by a factor of (slightly less than) 2, but \( T_{A2} \) increases by a factor of (slightly less than) 4.

Experiment: What if we try increasing the size of the input by a factor of 3? That is, start with \( n = 1 \), then \( n = 3 \), then \( n = 9, 27, 81 \), etc. You can work it out, but I’ll jump to the results: \( T_{A1} \) and \( T_{A3} \) also increase by a factor of (slightly less than) 3, and \( T_{A2} \) increases by a factor of (slightly less than) 9.

In general, we find that if the input \( n \) increases by a factor of \( k \), \( T_{A1} \) and \( T_{A3} \) also increase by (slightly less than) a factor of \( k \). We can write this as

\[
\frac{T_{A1}(k \cdot n)}{T_{A1}(n)} \leq k \quad \text{and} \quad \frac{T_{A3}(k \cdot n)}{T_{A3}(n)} \leq k
\]

Similarly, we find that when \( n \) increases by a factor of \( k \), \( T_{A2} \) increases by (slightly less than) a factor of \( k^2 \). We can write this as

\[
\frac{T_{A2}(k \cdot n)}{T_{A2}(n)} \leq k^2
\]

We got to those conclusions by observation, but we can reach the same conclusion algebraically. For example, we can write

\[
T_{A2}(n) = 4n^2 + 2n + 2
\]

\[
T_{A2}(k \cdot n) = 4(k \cdot n)^2 + 2(k \cdot n) + 2
= k^2 \cdot 4 \cdot n^2 + k \cdot 2 \cdot n + 2
\leq k^2(4n^2 + 2n + 2)
\]
and we see that $\frac{T_{A2}(k * n)}{T_{A2}(n)}$ is always $\leq k^2$

Now I’m going to make claims about these three functions that might look arbitrary. But I will show how I obtained them, and why they are useful.

Claim: \[ \forall n \geq 2, T_{A1}(n) \leq 3n \]
\[ \forall n \geq 4, T_{A2}(n) \leq 5n^2 \]
\[ \forall n \geq 2, T_{A3}(n) \leq 6n \]

Let’s focus on $T_{A1}(n)$. We have seen that it grows linearly (ie at the same rate) as n grows. Can we use that information to give any information about the actual value of $T_{A1}(n)$?

Suppose there is some particular value $n_0$ for which we can determine that $T_{A1}(n_0) \leq c * n_0$ for some positive constant $c$. Now consider $T_{A1}(k * n_0)$ where $k \geq 1$

From our previous discussion, we know $\frac{T_{A1}(k * n_0)}{T_{A1}(n_0)} \leq k$

and from there it is a simple step to $T_{A1}(k * n_0) \leq c * (k * n_0)$

Now if we replace “$k * n_0$” by a generic “$n$”, we get $T_{A1}(n) \leq c * n \ \forall n \geq n_0$

Are there such an $n_0$ and constant $c$? Yes! We can see that if we let $n_0 = 2$ and $c = 3$, the requirements are satisfied – this is exactly the claim I made above.

Now what about $T_{A3}$? You can work out that the same property holds (though you cannot use the same value for $c$)
But what about $T_{A2}$?

Rather than analyzing $T_{A2}$ specifically, let’s see if there is something that we can say about any timing function that is a polynomial function of the size of the input.

Suppose an algorithm $A$ has timing function

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

where the $a_i$ values are constants.

Claim 1: \( \exists \) \( n_0 \) such that \( \forall n \geq n_0, \ T_A(n) \leq (a_t + 1) \cdot n^t \)

To prove this claim, we will start with another, easier to prove claim:

Claim 2: \( \exists \) \( n_0 \) such that \( \forall n \geq n_0, n^t \geq a_{t-1} \cdot n^{t-1} + \cdots + a_1 \cdot n + a_0 \)

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

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

As $n$ increases, each 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.

This actually does most of the work of proving the original claim, which we repeat here:

Claim 1: \( \exists \) \( n_0 \) such that \( \forall n \geq n_0, \ T_A(n) \leq (a_t + 1) \cdot n^t \)

Proof: By Claim 2, \( \exists n_0 \) such that \( \forall n \geq n_0, n^t \geq a_{t-1} \cdot n^{t-1} + \cdots + a_0 \)

\[ \Rightarrow \forall n \geq n_0, \ T_A(n) \leq a_t \cdot n^t + n^t \]

ie \( \forall n \geq n_0, \ T_A(n) \leq (a_t + 1) \cdot n^t \)
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 $f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$ then we say $f(n) \in O(g(n))$

The significance of this is that as $n$ gets large, the growth-rate of $f(n)$ is no greater than the growth-rate of $g(n)$. In other words, the growth of $g(n)$ is an upper bound on the growth of $f(n)$.

We use this idea to group functions (specifically, timing functions for algorithms) into order classes.

Let's apply this to the three functions we saw before. It should be clear that $T_{A1}(n) \in O(n)$, $T_{A3}(n) \in O(n)$, and $T_{A2}(n) \in O(n^2)$

But wait! Are these order classes disjoint? Can a function be in more than one? YES!

For example, $T_{A1}(n) \leq 2n^2$ $\forall n \geq 2$, so $T_{A1}(n) \in O(n^2)$

In fact, every function in $O(n)$ is also in $O(n^2)$, and every function in $O(n^2)$ is also in $O(n^3)$ etc. We can write $O(1) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(n^4)$ etc
So any function is in infinitely many order classes. Which one do we use to describe it?

**We use the lowest (simplest) class we can for each function.** For example, we have shown that \( T_{A2}(n) \in O(n^2) \). Can we prove that \( T_{A2}(n) \notin O(n) \)?

Yes we can. Suppose \( T_{A2}(n) \in O(n) \). Then there must exist constants \( n_0 \) and \( c \) such that \( \forall n \geq n_0, T_{A2}(n) \leq c \cdot n \). But this means

\[
4n^2 + 2n + 2 \leq c \cdot n \quad \forall n
\]

\[
\Rightarrow 1 \leq \frac{c}{4n} \quad \forall n
\]

which clearly cannot be true as \( n \to \infty \). Thus \( T_{A2}(n) \notin O(n) \).

\( O(n^2) \) is the lowest complexity class that \( T_{A2}(n) \) belongs to.

And now, at long long last, we are able to say something definitive about \( T_{A1}(n), T_{A2}(n) \) and \( T_{A3}(n) \): \( T_{A1}(n) \) and \( T_{A3}(n) \) both belong to \( O(n) \) and \( T_{A2}(n) \) does not.
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</td>
</tr>
<tr>
<td>( c \times n^3 )</td>
<td>( O(n^3) )</td>
<td>cubic time</td>
</tr>
<tr>
<td>( c \times n^k ) Where ( k ) is a constant</td>
<td>( O(n^k) )</td>
<td>polynomial time</td>
</tr>
<tr>
<td>( c \times k^n ) where ( k ) 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 \( f_2(n) \in O(g_2(n)) \) and \( f_1(n) + f_2(n) \in O(max(g_1(n), g_2(n))) \) 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

\[
f(n) \geq c \cdot g(n) \quad \forall n \geq n_0
\]

then \( f(n) \in \Omega(g(n)) \) \((\Omega \text{ is the Greek letter "Omega"})\).

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

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 \( f(n) \in \Omega(g(n)) \).
Here’s an example:

Let $f(n) = 0.0001 \cdot n^2 + (10^6) \cdot n + 3$

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 $f(n) \geq c \cdot n^2 \quad \forall n \geq 0$.

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

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

i.e.

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

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

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 $c \cdot n^2 - 0.0001 \cdot n - 10^6$ is $\geq 1$ so

$$n \cdot (c \cdot n^2 - 0.0001 \cdot n - 10^6) \geq n,$$

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 it turns out that we can show an algorithm's complexity is in \( O(g(n)) \) and in \( \Omega(g(n)) \), 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 \( f(n) \in \Theta(g(n)) \).