Let's look at a simple example of determining the \( \Theta \) classification of a problem. The problem we will look at is evaluating a polynomial

\[
    f(x) = c_n \cdot x^n + c_{n-1} \cdot x^{n-1} + \cdots + c_2 \cdot x^2 + c_1 \cdot x + c_0
\]

First, we can observe that any algorithm that solves this must at the very least read or otherwise receive the values of \( x \) and the \( n+1 \) coefficients. Thus we can easily see that every algorithm for this problem must be in \( \Omega(n) \).

Consider the simple algorithm I will call BFI_Poly:

\[
    \text{BFI\_Poly}(x,c[n] \ldots c[0]):
    \begin{align*}
        &\text{value} = c[0] \\
        &\text{for } i = 1 \ldots n:
            \begin{align*}
                &\text{power} = 1 \\
                &\text{for } j = 1 \ldots i:
                    \begin{align*}
                        &\text{power} *= x \\
                        &\text{value} += c[i] \cdot \text{power}
                    \end{align*}
            \end{align*}
        \end{align*}
    \]

\text{BFI\_Poly()} clearly runs in \( O(n^2) \) time (you should verify this if it is not already familiar)

This is where algorithm (and data-structure) designers start to get excited. We have a problem with a lower bound of \( \Omega(n) \), and an algorithm that is in \( O(n^2) \)... can we either increase the lower bound, or decrease the upper bound?
It turns out that for this problem we can decrease the upper bound by finding a better algorithm - namely, Horner's rule:

```python
Horners_Poly(x, c[n] ... c[0]):
    value = c[n]
    for i = n-1 .. 0:
        value = value*x + c[i]
    return value
```

You should be able to verify that Horners_Poly correctly evaluates \( f(x) \) and that it runs in \( O(n) \) time.

(As a side-issue, can you find an easy way to modify BFI_Poly so that it also runs in \( O(n) \) time?)

Now we are in clover - the upper bound on our algorithm exactly matches the lower bound on the problem. We can now say that the problem is in \( \Theta(n) \). This really is very good news - it means we have found an algorithm for this problem that cannot be beat!

Well ... sort of.

It means our algorithm has the lowest possible complexity. There may be another algorithm with the same complexity and a lower value of \( c \), the constant multiplier. This is what we see when we compare mergesort and Quicksort: they have the same \( O(n \log n) \) complexity, but Quicksort is faster in general because it has a lower constant multiple. (Yes, of course I know that Quicksort has worst-case \( \Omega(n^2) \) complexity the way it is normally implemented. It is actually possible to modify Quicksort so that you can guarantee \( O(n \log n) \) performance but hardly anyone bothers because the pathological situations that give rise to the \( O(n^2) \) performance are very rare.)

******* The following information is really really interesting, but you can skip it now and read it later if you want. Look for another line like this one to find the point where you can skip to. *******

The study of \( \Theta \) classification has led to an incredibly important result in complexity theory with direct implications for algorithm design: **comparison-based sorting of a set of size \( \in \Theta(n \log n) \) where \( n \) is the size of the set.** In other words, there cannot be any sorting algorithm based on comparing elements of the set to each other that runs in less than \( O(n \log n) \) time.

A word about **comparison-based sorting**: most of the sorting algorithms we encounter are in this
category. Bubble-sort for example, (which we all know we would never use in most circumstances because it runs in $O(n^2)$ time) is based on repeatedly comparing two consecutive values in the array, and swapping them if required. Merge-sort boils down to a sequence of ever-larger merges, each of which consists of repeated comparisons between elements of the set. Quick Sort uses comparisons between values to partition the set into “small values” and “large values”, then sorts the two subsets recursively. Each of these can be expressed at the most abstract level as:

```plaintext
while (not sorted):
    compare two elements of the set
    based on the result of the comparison, do some stuff
```

So the question is: if we have a sorting algorithm that fits this pattern, can we put a lower bound on the number of comparisons we must do? It turns out that we can!

We can visualize the execution of such an algorithm as a binary tree (note that this does not mean that the algorithm involves building a tree ... in this analysis the tree is a **representation**al device for the execution of the algorithm). The root of the tree represents the first comparison. There are two possible outcomes, each leading to another comparison ... and each of those leads to two more, etc., until the set is sorted.
This tree has to include every possible sequence of comparisons that the algorithm might use to complete the sorting operation. Every possible initial permutation of the set of \( n \) values will follow a different sequence of comparisons to become sorted, so each leaf of this tree represents the termination of the algorithm for a different initial permutation. Since a set of \( n \) values has \( n! \) permutations, the execution tree must have \( n! \) leaves.

Now we are almost done. We can use the number of levels of the tree to put a lower bound on the running time of the algorithm. (For example, if the tree has 12 levels then there is some leaf that is only reached after 11 comparisons.) If we actually built this tree for bubble-sort we would see that it has about \( C \times n^2 \) levels for some constant \( C \), and if we built the execution trees for merge-sort or Quicksort we would see that those trees have about \( C \times n \log n \) levels for some constant \( C \).

But can we say anything about the minimum height of a binary tree with \( n! \) leaves? If we think about this for a moment, we can see that if a binary tree has \( X \) leaves at the bottom level, then the level above this has \( X/2 \) vertices, the one above that has \( X/4 \) vertices, and so on up to the root. In other words the number of levels is about \( \log_2 X \). Our execution tree for our unspecified comparison-based sorting algorithm has \( n! \) leaves, so it must have about \( \log_2 (n!) \) levels.

Because of the way logs work, we get

\[
\log_2 (n!) = \log_2 (1 \times 2 \times \cdots \times \frac{n}{2} \times (\frac{n}{2} + 1) \times \cdots \times n)
\]

\[
= \log_2 (1) + \log_2 (2) + \cdots + \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2} + 1) + \cdots + \log_2 (n)
\]

\[
\geq \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2} + 1) + \cdots + \log_2 (n)
\]

\[
\geq \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2}) + \cdots + \log_2 (\frac{n}{2})
\]

\[
\geq \frac{n}{2} \log_2 (\frac{n}{2})
\]

\[
= \frac{n}{2} (\log_2 (n) - \log_2 (2))
\]

\[
= \frac{n}{2} (\log_2 (n) - 1)
\]

[For those interested in LaTeX typesetting, the source for the above is:

\begin{align*}
\log_2 (n!) &= \log_2 (1 \times 2 \times \cdots \times \frac{n}{2} \times (\frac{n}{2} + 1) \times \cdots \times n) \\
&= \log_2 (1) + \log_2 (2) + \cdots + \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2} + 1) + \cdots + \log_2 (n) \\
&\geq \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2} + 1) + \cdots + \log_2 (n) \\
&\geq \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2}) + \cdots + \log_2 (\frac{n}{2}) \\
&\geq \frac{n}{2} \log_2 (\frac{n}{2}) \\
&= \frac{n}{2} (\log_2 (n) - \log_2 (2)) \\
&= \frac{n}{2} (\log_2 (n) - 1)
\end{align*}
\[
\log_2 \left( \frac{n}{2} \right) + \log_2 \left( \frac{n}{2} + 1 \right) + \cdots + \log_2(n) \\
\ge \log_2 \left( \frac{n}{2} \right) + \log_2 \left( \frac{n}{2} \right) + \cdots + \log_2 \left( \frac{n}{2} \right) \\
\ge \frac{n}{2} \log_2 \left( \frac{n}{2} \right) \\
\ge \frac{n}{2} \left( \log_2(n) - \log_2(2) \right) \\
\ge \frac{n}{2} \left( \log_2(n) - 1 \right)
\]

which we now know we can write as \( \log_2(n!) \in \Omega(n \log n) \)

And there it is! The execution tree for a comparison-based sort algorithm must have at least \( c \times n \log n \) levels, for some constant \( c \), and so every comparison-based sorting algorithm that can successfully sort all possible initial permutations is in \( \Omega(n \log n) \).

End of story? Well not quite. If we place restrictions on the initial permutation (so that not all \( n! \) initial permutations are possible) then we may be able to get a lower complexity (the execution tree does not need as many leaves). Also, there do exist sorting algorithms that are not comparison-based – under some circumstances these can run faster than \( n \log n \) time. But for general purpose, no-restrictions sorting, the result holds.

******* Ok, you can start reading again here. But you skipped over some amazing stuff – one previous student said this was their favourite thing they learned in CISC-235 – you should go back and read it sometime. *******

At this point we closed our introduction to the \( \Omega \) and \( \Theta \) complexity classifications, and finally turned our attention to a data structure: the **stack**
Stacks

Consider the problem of evaluating an arithmetic expression such as $3 + 4 \times 7 + 8 \times (3 - 1)$

Most people in North America have been taught that parentheses have highest precedence, followed by exponentiation, then multiplication and division, then addition and subtraction, so the expression above evaluates to $3 + 28 + 16 = 47$

But these precedence rules are completely arbitrary. For example, we could keep the rule about parentheses but do everything else in simple left-to-right order ... which would give 114 ... or right-to-left order ... which would give 103 ... or give addition higher precedence than multiplication ... which would give 210 (assuming I have done the calculations properly)

The notation we have used here to write down the expression is called infix notation because the operators (*, +, etc) are placed between the operands (3, 4, 7, etc)

In order to evaluate an infix expression correctly we need to know exactly what rules of precedence were assumed by the person who created the expression. Alternatively we can put parentheses all through the expression. For example $((3 + (4 \times 7)) + (8 \times (3 - 1)))$ corresponds to the standard evaluation, whereas $(((3 + 4) \times 7) + 8) \times (3 - 1))$ gives another result ... both are completely unambiguous.

Wouldn't it be nice if there were a universal way to represent an expression so that we don't need to know any rules of precedence, or use any parentheses to determine the order of evaluation.

There is! It is called postfix notation, and it was invented in 1924 by Jan Łukaseiwicz ... because he was Polish, this is sometimes called Polish Postfix notation. In postfix notation, operators come after their operands, so "3 * 4" (infix) becomes "3 4 *" (postfix)

The expression we started with : $3 + 4 \times 7 + 8 \times (3 - 1)$ can be written as $3 4 7 * + 8 3 1 - * +$

We can evaluate this in simple left-to-right order ... we keep going until we hit an operator (the *) and then we apply it to the two numbers just before it: $4 \times 7 = 28$, and we put the result in place where the $4 7 *$ was, so now the expression is $3 28 + 8 3 1 - * +$

The next thing we see is the +, which we apply to the numbers just in front of it (3 and 28) and put the result back in the expression, giving

$3 28 8 3 1 - * +$

The next operator we find is -, which we apply to 3 1. The expression is now
31 8 2 * +

The next operator is *, applied to the 8 2. This gives

31 16 +

We apply the + to the numbers before it, giving a final result of 47 ... which is exactly the result we expect from the original expression using standard rules of precedence ... but note that we did not need to know those precedence rules. Once we have the expression in postfix form we just evaluate from left to right.

But something magic happened there - I just pulled the postfix version of the expression out of thin air. Can we find a way to convert any infix expression to an equivalent postfix expression?

We can build it up one piece at a time ... for example, we can look at \( 3 + 4 \times 7 + 8 \times (3 - 1) \) and see the parenthesized part "(3 - 1)" which we can immediately write as "3 1 -" and now that this is taken care of, we can work on the next level of precedence: multiplication and division. We can see that "4 \times 7" will become "4 7 *", and that the "8 *" combines with the "3 1 -" to give "8 3 1 - *". Now all that we have left to deal with are the additions. We can resolve these in different ways, but perhaps the simplest is to put the first "+" right after the things it applies to ( the "3" and the "4 7 *") and the second "+" after the "8 3 1 - *".

The original expression could also be translated into postfix notation as \( 3 \ 4 \ 7 \ * \ 8 \ 3 \ 1 \ - \ * \ + \ + \)

You should make sure that you see that this gives the same result.

So the precedence rules have not gone completely – we need to use them when we convert from infix to postfix. There is an efficient algorithm to convert infix to postfix but it is a bit messy and we won’t address it here.

Instead, we focus on the much simpler problem of evaluating an expression in postfix notation.

Consider the postfix expression \( 3 \ 4 \ 2 \ * \ 8 \ 5 \ - \ * \). To evaluate this we need to scan across to find the first operator (the *) then apply it to the two most recent values. Next we find the +, and apply it to the two values preceding it, and so on.

The question is, how are we going to hold onto those values, and get them back in reverse order (ie most recent one first) when we need them? The answer is ... a stack.

A stack is our first example of an Abstract Data Type: we specify the operations we need to be able to perform on the data we will store, but we do not specify the details of the implementation. Of course when we actually write code we do need to choose a specific
implementation, and the choice we make will often have significant impact on the efficiency of our program.

A stack must provide (at least) three operations:

- **push(x)** - add the value x to the stack
- **pop()** - remove the most recently added value, and return it
- **is_Empty()** - return True if there are no values in the stack, and False otherwise

In keeping with popular practice, we will imagine that we have implemented a **Stack** class and that we can create a stack with a statement such as

```python
S = new Stack()
```

Then the operations listed above become methods attached to the stack we create.

Stacks are often described as a LIFO (Last In First Out) data structure: the most recently added (pushed) value is the first one removed (popped). The first thing to notice about a stack is that it automatically reverses the order of a sequence:

```python
S = new Stack()
S.push(1)
S.push(3)
S.push(7)
S.push(9)
print S.pop(), S.pop(), S.pop(), S.pop()
```

will print **9 7 3 1**

Holding onto things and giving them back in reverse order is exactly what we need for our evaluate-postfix algorithm. The algorithm looks like this:
Evaluate_Postfix(e):
    # e is an expression in postfix notation, in which we can
    # identify the individual tokens (operands, operators and
    # parentheses)
    # We will assume e is well-formed
    S = new Stack()
    for x in e:
        if x is a value:
            S.push(x)
        else:
            # x is an operation
            values = []       # an empty list
            while len(values) < the number of parameters for x:
                values.append(S.pop())
            S.push(x(values))
    return S.pop()

As noted, this assumes that e is well-formed – if it is not (too many or too few operators) then
we need to use the is_Empty() method to avoid errors. A revised version is shown on the
next page.
Evaluate_Postfix(e):
    # e is an expression in postfix notation, in which we can
    # identify the individual tokens (operands, operators and
    # parentheses)
    # We will assume e is well-formed
    S = new Stack()
    for x in e:
        if x is a value:
            S.push(x)
        else:
            # x is an operation
            values = []  # an empty list
            while len(values) < the number of parameters for x:
                if S.is_Empty():
                    ERROR("Stack is empty")
                    # not enough values in e
                else:
                    values.append(S.pop())
            S.push(x(values))
    result = S.pop()
    if S.is_Empty():
        return result
    else:
        ERROR("Stack is not empty")
        # too many values in e

It is worth noting that postfix notation is very important in computer science because it gives a good model how arithmetic is actually carried out in a computer. When we write a high-level statement like
    C = A + B
it gets translated into assembly language sort of like this:
    load the contents of address A into a CPU register
    load the contents of address B into another CPU register
    add the contents of those two registers and store the result in another CPU register
    copy that register to address C

In other words the addition is really carried out in a postfix way: we identify the operands, then execute the operation on them.

What can we say about this evaluate-postfix problem in terms of its complexity? It should be
clear that this problem is in $\Omega(n)$ - where $n$ is the length of the expression - since we must at least look at every token in the expression.

Furthermore, you can see that the algorithm given is in $O(n)$ since the amount of work done for each token is bounded by a constant (assuming that all our arithmetic operations take constant time). Thus we have an algorithm that solves the problem with Big O classification equal to the $\Omega$ classification of the problem. Thus this problem is in $\Theta(n)$ and we know that no algorithm for this problem can have a lower Big O classification.

*Wait a minute ... there's a big assumption in that last paragraph ...*

The claim that the algorithm is in $O(n)$ is only true if each of the stack operations is in $O(1)$ (i.e. takes constant time) ... and that may not be true!

So now we need to look at the actual implementation of a stack.

There are two simple solutions: **store the stack in a one-dimensional array** or **store the stack in a linked list**

**array:** we can use an array with indices in the range $[0..k]$ for some $k$. We store the stack in locations 0, 1, 2 etc, with location 0 holding the first item pushed onto the stack, etc. We can use a variable called top to keep track of the top of the stack. Our Stack class might look something like this:

```python
class Stack():
    def init():
        this.array1 = new array[0..k]
        this.top = -1  # the stack is empty

    def push(x):
        if this.top == k:
            ERROR("Stack overflow")
        else:
            this.top += 1
            this.array1[top] = x

    def pop():
        if this.top == -1:
            ERROR("Can’t pop from empty stack")
        else:
            x = this.array1[top]
            this.top -= 1
            return x

    def is_Empty():
        return this.top == -1
```
This is very fast and simple - but of course the maximum size of the stack is limited. This can be handled by allocating a new, larger array when needed.

A linked list: we need to create a Node object, containing two fields:

- **value**: the value being stored
- **next**: a pointer to another Node object

Now our Stack class might look like:

```python
class Stack():
    def init():
        this.top = NULL

    def push(x):
        newNode = new Node()
        newNode.value = x
        newNode.next = this.top
        this.top = newNode

    def pop():
        if this.top == NULL:
            ERROR("Can’t pop from empty stack")
        else:
            x = this.top.value
            this.top = this.top.next
        return x

    def is_Empty():
        return this.top == NULL
```

This involves more operations per push and pop than the array version and so will be a bit slower in practice. However it has the benefit that there is no upper limit on the size of the stack.

Now we can verify that with either of these implementations, all stack operations take O(1) time ... so the $\Theta(n)$ classification of the problem is correct.
Stacks are widely used - most compilers and programming environments use a stack to handle nested function calls (sometimes called the "execution stack" or the "call stack"). Browsers and text-editors use stacks to implement the “back” and “undo” operations. Adobe Postscript is heavily stack-based. IBM, Apple and NASA use a language called Forth which is completely stack-based. One of the appeals of the stack data structure is that it is very simple and can be implemented in limited memory space, yet it is very versatile.