Much of the material we covered this week was already posted in the notes for last week. These notes take up where those left off, and fill in some gaps.

We have discussed the $\Theta$ notation as applied to bounding the growth-rate of the time required for an algorithm. That’s a powerful tool, but the real strength of the $\Omega$ and $\Theta$ notations lies in applying them to problems.

What would it mean to say that a problem $P$ has a lower bound $g(n)$ on its complexity? It would mean that we can prove that every possible algorithm that solves $P$ is $\in \Omega(g(n))$.

How can we do that? We would have to prove the statement not only for all known algorithms that solve $P$, but also all algorithms that might be discovered in the future that solve this problem.

We can do this by making some simple assumptions about the computer architecture – basically that we are only considering sequential (non-parallel) machines, with constant-time arithmetic operations and a random-access memory. This rules out possible breakthroughs such as effective quantum computing, hyperspace or time-travel.

Within these constraints, we can see immediately that any problem that requires reading $n$ input values must be in $\Omega(n)$. This is kind of trivial but it is often the best we can do. It’s worth pointing out that sometimes we ignore the input phase of an algorithm – the best example is binary search, which we always describe as being in $O(\log n)$. Obviously this is only true if we ignore the time required to input (and sort) the set of values.

Sometimes we can do better. For example, suppose a certain problem requires multiplying all pairs of values in a set of size $n$. There are about $\frac{n^2}{2}$ such pairs so any sequential algorithm that computes the necessary products must be in $\Omega(n^2)$.

Knowing the $\Omega$ classification of a problem can help us in our quest to find an optimal algorithm for the problem. For example if we can show that a problem is in $\Omega(n^3)$ and the best algorithm we have is in $O(n^4)$, then there may be a more efficient algorithm still waiting to be discovered. But if we find an $O(n^3)$ algorithm for this problem then we can say the problem is in $\Theta(n^3)$ – all algorithms for this problem grow at least as fast as $n^3$ grows, and
we have found an algorithm that grows exactly that fast.

There is a famous and deeply studied problem that must be mentioned here: matrix multiplication. Given two \( n \times n \) matrices, we wish to compute their product. Since we have to input \( 2 \times n^2 \) values this problem is clearly in \( \Omega(n^2) \). The naive matrix multiplication algorithm is in \( O(n^3) \). For decades people have been trying to establish the \( \Theta \) classification on this problem by finding faster and faster algorithms.

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

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

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:

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

BFI_Poly() clearly runs in \( O(n^2) \) time (you should verify this if it is not already familiar)

So 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 using a better algorithm - namely, Horner's rule:

\[
\text{Horners\_Poly}(x,c[n] \ldots c[0]):
\]
\[
\begin{align*}
\text{value} & = c[n] \\
\text{for } i = n-1 \ldots 0: \\
\text{value} & = \text{value} \times x + c[i] \\
\text{return } value
\end{align*}
\]

You should be able to verify that \text{Horners\_Poly} correctly evaluates \( p(x) \) and that it runs in \( O(n) \) time.

(As a side-issue, can you find an easy way to modify \text{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 \( O(n \times \log n) \) complexity, but Quicksort is faster in general because it has a lower constant multiple. (Yes, I know that Quicksort has worst-case \( O(n^2) \) complexity the way it is normally implemented. It is actually possible to modify Quicksort so that you can guarantee \( O(n \times \log n) \) 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 and data structure design: comparison-based sorting of a set \( \in \Theta(n \times \log n) \) 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 \( O(n \times \log n) \) 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 representational 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.

![Binary tree diagram](image-url)
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 $\frac{X}{2}$ vertices, the one above that has $\frac{X}{4}$ vertices, and so on up to the root. In other words the number of levels is about $\log_2 X$.

So the execution tree for any possible comparison-based sorting algorithm must have about $\log_2 (n!)$ levels.

Because of the way logs work, we get

\[
\log_2 (n!) = \log_2 \left( 1 \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \cdots \cdot \frac{n}{2} \cdot (\frac{n}{2} + 1) \cdot \cdots \cdot n \right) \\
= \log_2 (1) + \log_2 (2) + \cdots + \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2} + 1) + \cdots + \log_2 (n) \\
\geq \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2} + 1) + \cdots + \log_2 (n) \\
\geq \log_2 (\frac{n}{2}) + \log_2 (\frac{n}{2}) + \cdots + \log_2 (\frac{n}{2}) \\
\geq \frac{n}{2} \log_2 (\frac{n}{2}) \\
= \frac{n}{2} (\log_2 (n) - \log_2 (2)) \\
= \frac{n}{2} (\log_2 (n) - 1)
\]

which we now know means that we can write $\log_2 (n!) \in \Omega(n \log n)$.
And there it is! The execution tree for *any* 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 sorting story? Not quite (stories never end). 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 \( O(n \log n) \) 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 of the Omega and Theta complexity classifications, and finally turned our attention to a data structure: **the stack**

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. Wouldn't it be wonderful if there were a universal way to represent an expression so that no matter what rules of precedence are in use, the method of evaluating the expression is always the same?

There is! It is called **postfix notation**, and it was invented in 1924 by Jan Lukaseiwicz (I spelled his name wrong in class – oops!)... 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

\(31 8 3 1 - * +\)

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 \[ 3 4 7 * 8 3 1 - * + + \]

In class I mentioned that there is a well-known algorithm to translate expressions from infix notation to postfix notation. I’m including this algorithm in the notes here – it’s worth looking at. I’m explaining it in detail in these notes but you can treat this as “enrichment” material. If you wish to skip over it, look for the line that starts ********

If you examine the two postfix expressions just given above, you may notice that the operands (the numbers) are in exactly the same order as they were in the original infix expression. This gives a clue to how we might design an algorithm to do the translation:

- leave the operands in the same order
- working in decreasing order of precedence, push each operator to the right until it is just to the right of the operands that it applies to

The problem with this is that it requires multiple passes over the infix expression. To avoid this we take a different approach: we step through the infix expression from left to right, passing the operands straight through, but keeping track of the operators as we go and "holding them in reserve". When we encounter an operator with higher precedence than the previous one, we add it to the ones we are holding. When we encounter an operator with equal or lower precedence than the previous one, we "bring back" (ie “output”) the high precedence operator(s) we are holding on to, then we "hold on to" the new operator.

The key concept is that we defer each operator until deferring it any longer would create an error.
As an example, consider the infix expression $6 + 8 \times 4 / 9 - 5 \ldots$ we process it from left to right

- $6$ is an operand, so we simply output it
- $+ \;$ is an operator, so we hold onto it
- $8$ is an operand so we output it
- $\times \;$ is an operator, its precedence is higher than the previous one ($+$) so we hold onto it
- $4$ is an operand so we output it
- $/$ is an operator with equal precedence to the previous one, so we output the previous one, and hold onto the $/$
- $\times \;$
- $9$ is an operand so we output it
- $- \;$ is an operator with lower precedence than the previous one ($/$) so we bring back the $/$
- $/$
- Now we compare the $-$ to the other operator we are holding onto ($+$). This also has precedence $\geq$ the new operator so we output it too
- $+ \;$
- $5$ is an operand so we output it

We are left holding onto the $-$ and there are no more numbers so we output the $-$

Thus we get $6\ 8\ 4\ \times\ 9\ /\ +\ 5\ -$ which is a valid postfix form of the original infix expression

The question is, how are we going to hold onto those operators, and get them back in reverse order 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
- **isEmpty**() - 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

\[ S = \text{new Stack()} \]

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:

\[
S = \text{new Stack()}
S.\text{push}(1)
S.\text{push}(3)
S.\text{push}(7)
S.\text{push}(9)
\]

print \( S.\text{pop()}, S.\text{pop()}, S.\text{pop()}, S.\text{pop()} \)

will print 9 7 3 1

Holding onto things and giving them back in reverse order is exactly what we need for our infix-to-postfix algorithm. The algorithm looks like this:

```python
InfixToPostfix(e):
# e is an expression in infix notation, in which we can
# identify the individual tokens (operands, operators and
# parentheses)
# We will assume e is well-formed
S = \text{new Stack()}
postFix = \text{empty list}
for t in e:
    if t is an operand:
        append t to postFix  # we don't add operands to the stack, we just
    else if t is a left parenthesis ":("
        S.push(t)
    else if t is a right parenthesis ":)"
        # pop off all stored operators, back to the
        # matching left parenthesis
        x = S.pop()
        while x != "(" :
            append x to postFix
            x = S.pop()
    else:
        # The only remaining possibility is that t is an operator. We will add it to the
        # stack, but first we must pop off any operators that need to be added to the
        # Postfix expression now.
```
if not S.isEmpty():
    # pop stored operators until we find one with lower precedence
    # than t
    x = S.pop()
    while precedence(x) >= precedence(t):
        # assume precedence(x) returns the precedence level of x
        # for example, precedence("*") would be > precedence("+")

        # for the purpose of this algorithm, precedence("(") - ie. the
        # precedence of a left parenthesis – must be 0
        append x to postFix

        # it's time for this operator to join the postFix
        # expression
        if not S.isEmpty():
            x = S.pop()
            # get the next one from the stack
        else:
            break
        # exit the while loop because the stack is empty
    if precedence(x) < precedence(t):
        # if the last thing we removed from the stack has lower
        # precedence than t, it needs to go back on the stack
        S.push(x)
    S.push(t)
    # the new operator always goes on the stack, waiting until
    # its operands are ready

while not S.isEmpty():
    # add any operators still in S to the postfix expression
    x = S.pop()
    append x to postFix

return postFix

That's a long-winded, complex-looking algorithm, and you should work through it by hand for a couple of examples to see how it works. But you don’t need to memorize it.

******

We’ll pick up the thread by showing that an expression in postfix form can be evaluated with a simple algorithm ... also using a stack. Don’t worry – this algorithm is a lot simpler than the one that creates the postfix form.
Let \( E \) be a string that represents an expression in postfix form.
Assume \( E \) has a “next” method that returns the next token in \( E \).
Let \( S \) be a stack.

while we haven’t processed all of \( E \):
    \( x = E.\text{next()} \)
    if \( x \) is a value:
        \( S.\text{push}(x) \)
    else:
        # \( x \) is an operator
        let \( n \) be the number of values required for \( x \) (usually 2)
        pop the top \( n \) values from \( S \)
        \( y \) = the result of applying operator \( x \) to the values just popped off \( S \)
        \( S.\text{push}(y) \)
return \( S.\text{pop}() \)

This algorithm will fail if \( E \) is not well-formed. As an exercise, improve the algorithm by using the \( \text{isEmpty()} \) method to avoid problems.

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.

Now, what can we say about this postfix evaluation problem in terms of its complexity? It should be clear that this problem is in \( \Omega(n) \)- where \( n \) is the length of the postfix 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 (remember: all arithmetic operations take constant time – this is part of our model of computation). Thus we have an algorithm with \( O \) 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 \( O \) classification.

Wait a minute … there’s a big \textit{unstated} 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 isEmpty():
        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.
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__(self):
        self.top = NULL

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

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

    def isEmpty():
        return self.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"). 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.
Stack exercises:

1. Write an algorithm that will move the top value on one stack to the top of another stack.

2. Write an algorithm that starts with a stack containing $n$ integers and finishes with the same integers in the same stack, but with the value that was on the bottom of the stack moved to the top, and all other values moved down one position. For example if the stack initially looks like this:

   4 ← top
   17
   9
   23

   then it should finish like this:

   23
   4
   17
   9

   You may use another temporary stack in your algorithm.

3. Write an algorithm that takes as input the integers \{1, 2, ..., n\} in a randomly determined arrangement on two stacks, and a target arrangement of the same integers on the same two stacks. Using only the methods created in exercises 1 and 2, rearrange the integers to match the target arrangement.

   For example suppose $n = 3$,
   
   start arrangement is $3 \ 1 \ 2$ on the first stack and $2 \ 3$ on the second stack,
   
   target arrangement is $2 \ 3 \ 1$ on the first stack and nothing on the second stack

   One solution is
   
   - move the top of Stack 1 to Stack 2 (as in Exercise 1)
   - move the bottom of Stack 2 to the top of Stack 2 (as in Exercise 2)
   - move the top of Stack 2 to the top of Stack 1
   - move the top of Stack 2 to the top of Stack 1
It’s not hard to create a generic algorithm that will transform any initial arrangement to any target arrangement ... but creating an algorithm that performs the transformation in the smallest number of steps is much more challenging.