20200116

We have discussed the Θ 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 Ω and Θ 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 Ω 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 x n matrices, we wish to compute their product. Since we have to input $2 * n^2$ 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 Θ classification on this problem by finding faster and faster algorithms.

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

$$p(x) = c_k * x^k + c_{k-1} * x^{k-1} + \cdots + c_2 * x^2 + c_1 * x^1 + c_0$$

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 $k + 1$ coefficients. Thus we can easily see that every algorithm for this problem must be in $\Omega(k)$.

Consider the simple algorithm I will call BFI_Poly:

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

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

So we have a problem with a lower bound of $\Omega(k)$, and an algorithm that is in $O(k^2)$... 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:

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

You should be able to verify that Horners_Poly correctly evaluates $p(x)$ and that it runs in $O(k)$ time.

(As a side-issue, can you find an easy way to modify BFI_Poly so that it also runs in $O(k)$ 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** $\Theta(k)$. 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, 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 $O(n * \log n)$ 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.

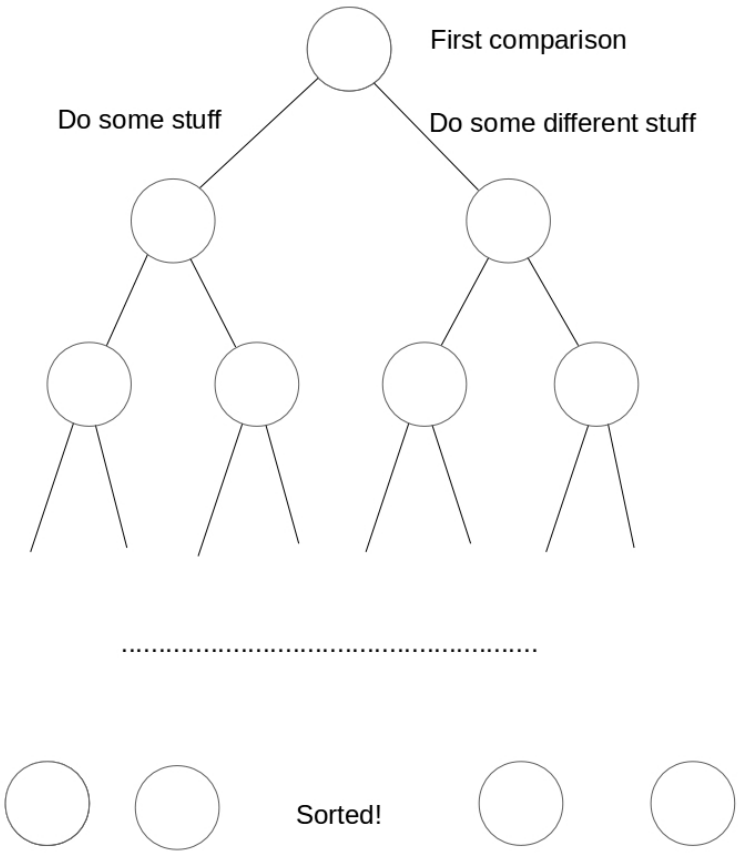
The study of Θ classification has led to an incredibly important result in complexity theory with direct implications for algorithm and data structure design: **comparison-based sorting of a set** $\in \Theta(n * \log n)$ **where** n **is the size of the set.** In other words, there **cannot** be any 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:

```
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.



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 $c * n^2$ 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 $c * n \log n$ 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

$$\begin{aligned}
\log_2(n!) &= \log_2(1 * 2 * \cdots * \frac{n}{2} * (\frac{n}{2}+1) * \cdots * n) \\
&= \log_2(1) + \log_2(2) + \cdots + \log_2(\frac{n}{2}) + \log_2(\frac{n}{2}+1) + \cdots + \log_2(n) \\
&\geq \log_2(\frac{n}{2}) + \log_2(\frac{n}{2}+1) + \cdots + \log_2(n) \\
&\geq \log_2(\frac{n}{2}) + \log_2(\frac{n}{2}) + \cdots + \log_2(\frac{n}{2}) \\
&\geq \frac{n}{2}\log_2(\frac{n}{2}) \\
&= \frac{n}{2}(\log_2(n) - \log_2(2)) \\
&= \frac{n}{2}(\log_2(n) - 1)
\end{aligned}$$

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*** $c * n \log n$ 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 – in a previous year one student said this was their favourite thing they learned in CISC-235 – you should go back and read it sometime.

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:

```
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 isEmpty():
        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"). 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 1and 2, rearrange the integers to match the target arrangement.

For example suppose

n = 3,

start arrangement is $\begin{matrix}3\\1\end{matrix}$ on the first stack and $\underline{2}$ on the second stack,

target arrangement is $\begin{matrix}3\\2\\1\end{matrix}$ 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.