Recursion

All code in this section will be in Python since that was the language of instruction in CIT591 (or 590). Even if you do not know Python, the syntax is easy enough to understand. You will not be asked to code in exams. A choice of language is just needed so that it is completely unambiguous as to what my program is doing.

Recursion

The idea behind recursion is the same as that being used in proofs by induction. Can we show the fact that if a problem is solved (if a theorem is true) for smaller instances (for n-1, n-2 etc) then the problem is solvable (then the theorem is true) for the current instance (for n).

The canonical example used to introduce recursion in most CS classes is factorial. Factorial has this nice property that $factorial(n) = n \times factorial(n-1)$. So when you write the function recursively as follows

```
def factorial(n):
    if n == 0: return 1
    return n * factorial(n-1)
```

there is level of trust that goes on that should not make you really question something along the lines of 'why should I believe factorial(n-1) will get computed correctly'. The correct way of thinking is to say let me assume factorial(n-1) is computed correctly. Can I do something with that?

Proving a program works!

It is sometimes important to know if a program really works. And by this we do not mean, is there a syntax error or a logical flaw. We are asking the question that in a world where the idea that you have (CS calls those algorithms) gets perfectly translated into code (in some non crazy programming language), can you prove that your program will work regardless of the input provided to it.

Now the current best practice in the industry to do this proof is to write a bunch of unit tests. But does unit testing equate to a proof of correctness?

Unfortunately, math will just laugh at that notion. Unit testing just amounts to proof by example. Wouldn't it be better if you could say it works for everything. That turns out to be a universal statement.

So for factorial this is what you want to prove

This program will compute factorial correctly regardless of input!

How do you actually write a proof? You rely on the fact that induction is recursion!

Here's a sketch of the proof.

Base case - observe that when n=0 then the program returns 1 and that happens to be the same as 0!

Induction hypothesis - we assume that factorial works just fine for 0,1,2,...,n-1

Now for computing factorial of n, see that the program computes it as $n \times factorial(n-1)$. But we know by induction that we actually have $factorial(n-1) = (n-1)!$.

So the program is just computing $n \times (n-1)!$. But that is just $n!$.

How long does it take? aka Big - O

As noted before, in most cases in the industry, you do not care about proving the correctness of your code (depends on the industry of course). But you do care about how much time your program is going to take!

While there are several ways of measuring time, the theoretical computer science methodology is to basically focus on the particular aspect of scalability.

Take for instance the case of searching through an array of size n.

The way a CS person expresses the time taken for an algorithm is to say the algorithm is O(f(n)). For instance the search problem can easily be shown to be what is called $O(n)$ (once you know the definition of O). So you will say

Searching in an unsorted array is an O of n operation.

or

Searching is a linear operation.

Definition of big-O:

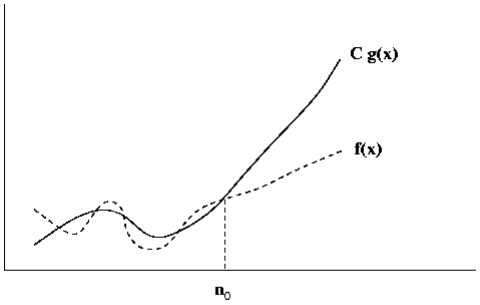
Let f and g be two functions defined on some subset of the real numbers. One writes

$$f(x) = O(g(x)) \text{ as } x \to \infty$$

if and only if there is a positive constant C such that for all sufficiently large values of x, $f(x)$ is at most C multiplied by the absolute value of $g(x)$. That is, $f(x) = O(g(x))$ if and only if there exists a positive real number C and a real number n_0 such that

$$|f(x)| \leq C|g(x)| \text{ for all } x \geq n_0.$$

While the formal definition is needed for the math's sake, most CS people think of it in terms of this graph



f(x) is O(g(x))

Beyond a certain point, a constant multiple of one of the functions dominates the other.

Example
Show that $n^2 + 100n$ is $O(n^2)$.

Show in class why in this case you can set C to be a number like 200 and x_0 to be 2. Obviously other values are possible but the thing about the O analysis is that you can be sloppy

Sloppiness is ok?

Algorithmic complexity is interesting in that it deals with inequalities more than equalities. So you will often look at the expression for time taken and draw conclusions very quickly by dropping terms. For instance if you compute the relationship between search time and the size of the array and say that it is $100n$, that is actually generally expressed as $0(n)$.

The constants do matter, but first it is important to make an efficient algorithm in the big - O sense. If you have two competing algorithms that are both linear, then you can worry about the constants. But if you have one which is linear and one which is quadratic, it should be clear which one is to be picked.

Therefore, remember that you are allowed to do some sloppy math when you are analyzing algorithms. The key is to begin the sloppiness only after you have written a correct expression down and done some level of evaluation.

Mergesort

Here is one version of the mergesort code in Python.

```
def mergeSort(a):
    if len(a)==0 or len(a) == 1:
        return a
    else:
        firstHalf = mergeSort(a[:len(a)/2])
        secondHalf = mergeSort(a[len(a)/2:])
```

```
        return merge(firstHalf, secondHalf)

def merge(a1, a2):
    a3 = []
    len1 = len(a1)
    len2 = len(a2)
    i=j=0
    while i < len1 and j < len2:
        if a1[i] < a2[j]:
            a3.append(a1[i])
            i += 1
        else:
            a3.append(a2[j])
            j += 1
    # we would either have gone through
    # all of a1 or all of a2 by this time
    # just tack on the remainging stuff at the end
    if i < len1:
        a3.extend(a1[i:])
    if j < len2:
        a3.extend(a2[j:])
    return a3
```

Recurrences

A recurrence relation in the most basic sense is an equation which is trying to define a sequence recursively.

So things like $f(n) = n*f(n-1)$ (factorial) or $T(n) = 2T(n/2)+n$ (merge sort recurrence).

The analysis of algorithms that are recursive in nature boils down to solving recurrences and we will cover some theorems

First order Linear recurrences with constant coefficients

Theorem 1. *The closed form solution for*

$$T(n) = \begin{cases} rT(n-1) + g(n) & \text{if } n > 0 \\ a & \text{if } n = 0 \end{cases}$$

is given by
$T(n) = r^n a + \sum_{i=1}^{n} r^{n-i} g(i)$

Proof. The proof of most any property related to recursion and recurrences is best done with induction.

Base case: When $n = 0$, we have $T(0) = a$ as per the definition and as per the formula we have $r^0 a + \sum_{i=1}^{0} r^{0-i} g(i)$. But the summation just returns 0 if we are going from a higher number to a lower number. So this just reduces to $r^0 a = a$.

Assume the formula is correct for $T(n-1)$.

We know $T(n) = rT(n-1) + g(n)$.

This is where the induction kicks in. Just substitute the formula for $T(n-1)$ since we have assumed that to be true.

$$\begin{aligned} T(n) =& r(r^{n-1}a + \sum_{i=1}^{n-1} r^{n-1-i} g(i)) + g(n) \\ &= r^n a + \sum_{i=1}^{n-1} r^{n-i} g(i) + g(n) \\ &= r^n a + \sum_{i=1}^{n-1} r^{n-i} g(i) + r^{n-n} g(n) \\ &= r^n a + \sum_{i=1}^{n} r^{n-i} g(i) \end{aligned}$$

and this matches the claim being made for $T(n)$.

Combine this with the base case and we have a proof by induction.

□

Application of the theorem.

This theorem can readily be applied for a whole number of 'real world' problems.

For instance, here is a recursive way of finding the maximum element in an array

```
def maximo(arr):
    if len(arr) == 1:
        return arr[0]
    return max(arr[0], maximo(arr[1:]))
```

How long does this function take? What is the running time of this algorithm?

Both of these questions are generally meant to be answered in the big-O sense. In particular, we need to express $T(n)$ (the time taken on an input of size n) as $O(g(n))$ where $g(n)$ is one of the commonly found functions.

As we can see from the algorithm, to solve the n sized version of the problem, we make one call to the function with an $n-1$ sized version of the problem and do a single computation after that.

$$T(n) = T(n-1) + 1$$

and it is very easy to use the above theorem to see that the closed form solution for this recurrence is going to be $T(n) = n$. So the running time is $O(n)$.

Another example from the HW

The Towers of Hanoi recursion worked out to $Moves(n) = 2Moves(n-1) + 1$. Also in this case we have $Moves(0) = 0$.

By direct application of the theorem we have the result that

$$\begin{aligned} Moves(n) &= 2^n \cdot 0 + \sum_{i=1}^{n} 2^{n-i} \cdot 1 \\ &= 2^{n-1} + 2^{n-2} + \ldots + 1 \\ &= 2^n - 1 \end{aligned}$$

Mergesort recurrence

For mergesort we can write the recurrence as the following

$$T(n) = 2T(n/2) + T_{merge}$$

Also as a base case, assume that sorting an array of size 1 takes some constant time a. $T(1) = a$.

since we are dividing the problem into roughly equal halves and then doing the merging. Merging involves going through the arrays in a loop but each of the arrays that are being merged is just looped over once. So we conclude that merging is a linear time operation. And we can write T_{merge} as some kn for a positive integer constant k.

$$\begin{aligned} T(n) &= 2T(n/2) + kn \\ &= 4T(n/4) + kn + kn \\ &= 8T(n/8) + kn + kn + kn \\ &= 2^i T(n/2^i) + ikn \\ &= an + kn \log_2 n \end{aligned}$$

$n \log_2 n$ is a term that dominates over n (beyond a certain value of n).

which means that mergesort is O(n log n).