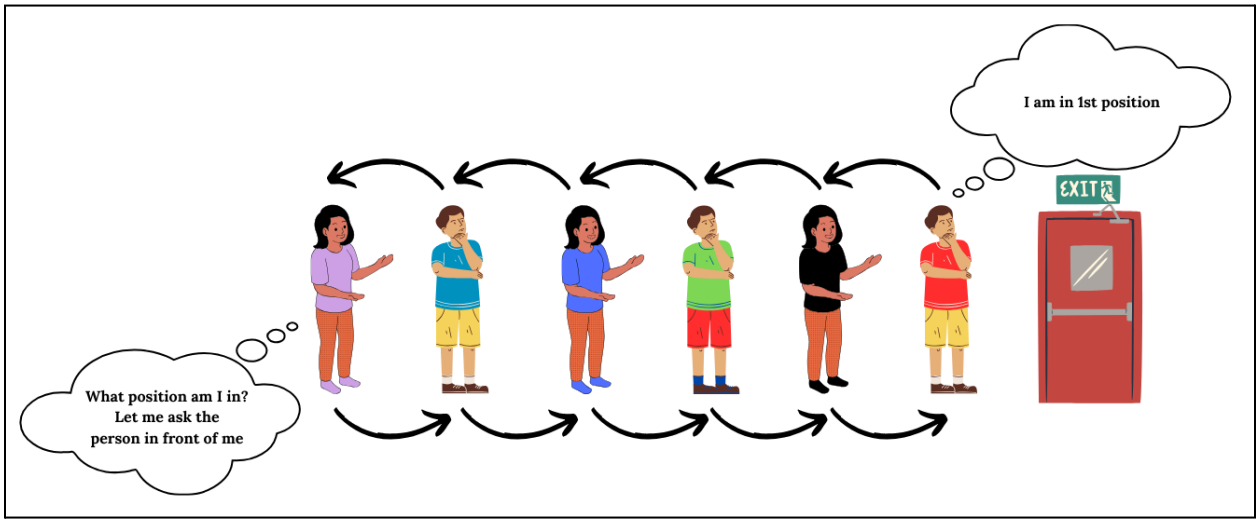
Recursion

A recursive function is no different than a normal function. The motivation for using recursive functions vs non-recursive is that the recursive solutions are usually easier to read and comprehend. The drawbacks are that you may need a little time to learn how to use it and you may need a little longer to debug errors. It takes more processing time and more memory. However, there can be cases when recursion is the best way.

Recursive Function - A recursive function has two parts: the base case, and the recursive case. The recursive case is when the function calls itself. The base case is when the function doesn't call itself again so it doesn't go into an infinite loop.



Recursion is a method that solves a problem by solving smaller (in size) versions of the same problem by breaking it down into smaller subproblems.

Factorial:

The factorial of a non-negative integer n is defined to be the product of all positive integers less than or equal to n. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.

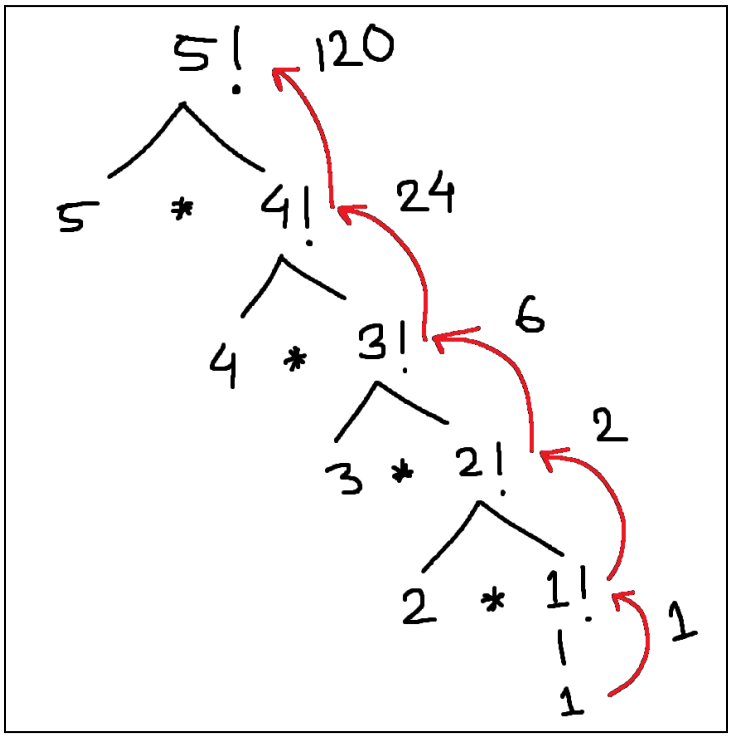
```
0! = 1
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
```

4! = 4 * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1
.....
..... And so on

We can write the recursion definition as

$$n! = \begin{cases} 1 & \text{if } n=1 \\ 1 & \text{if } n=0 \\ n*(n-1)! & \text{if } n>1 \end{cases}$$

Let's see the recursion tree for 5!



Now, let's see the recursive programming for this

```
def fact(n):
        if n==0 or n==1:
                return 1   #Base Case
        else:
                return n * fact(n-1)  #Recursive Part
```

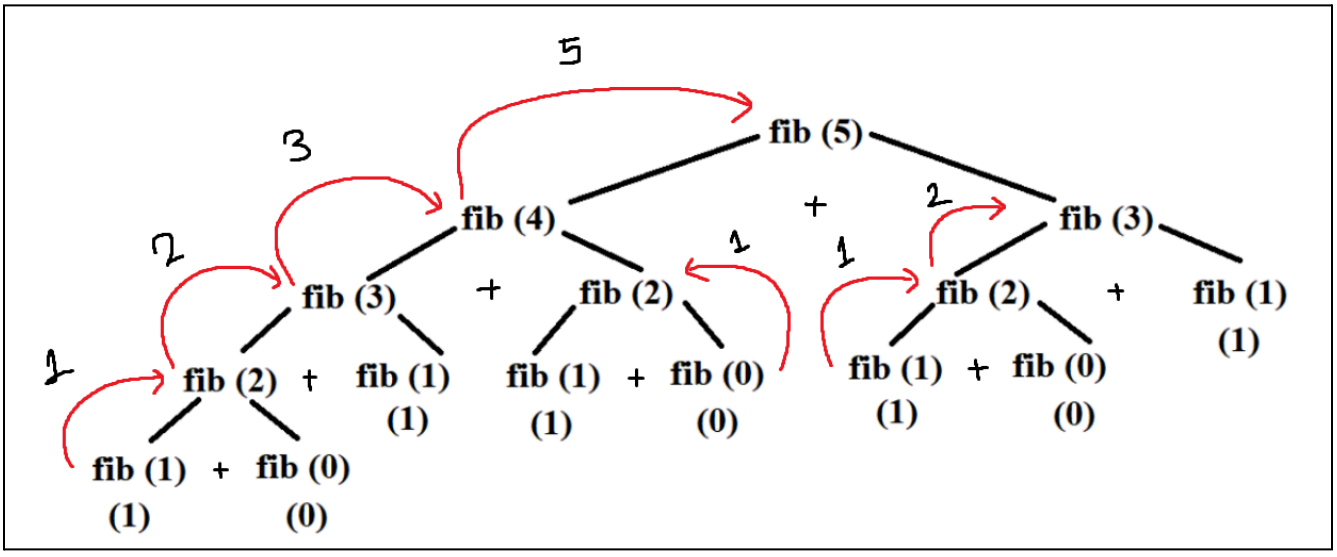
Fibonacci numbers:

The Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, ... We see that each number, except for the first two, is nothing but the sum of the previous two numbers.

We can write the recursion definition as

$$\text{fib}(n) = \begin{cases} 1 & \text{if } n=1 \\ 0 & \text{if } n=0 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n>1 \end{cases}$$

Let's see the recursion tree for fib(5)



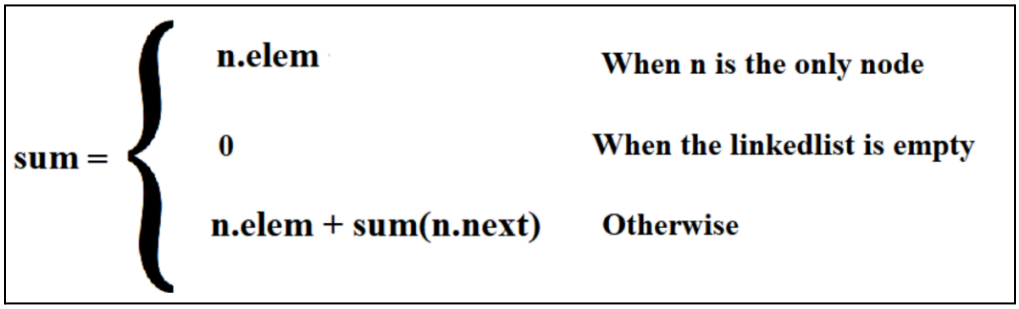
Now, let's see the recursive programming for this

```
def fib(n):
        if n==0:
                return 0  #Base Case
        if n==1:
                return 1   #Base Case
        else:
                return fib(n-1) + fib(n-2)  #Recursive Part
```

Sum of numbers in a Linked List:

The sum of the numbers in a list is nothing but the sum of the first number plus the sum of the rest of the list. The problem of summing the rest of the list is the same problem as the original, except that it's smaller by one element! Ah, recursion at play. The shortest list has a single number, which has a sum equal to the number itself. Now we have a base case as well. Note that if we allow our list of numbers to be empty, then the base case will need to be adjusted as well: the sum of an empty list is simply 0.

We can write the recursion definition as



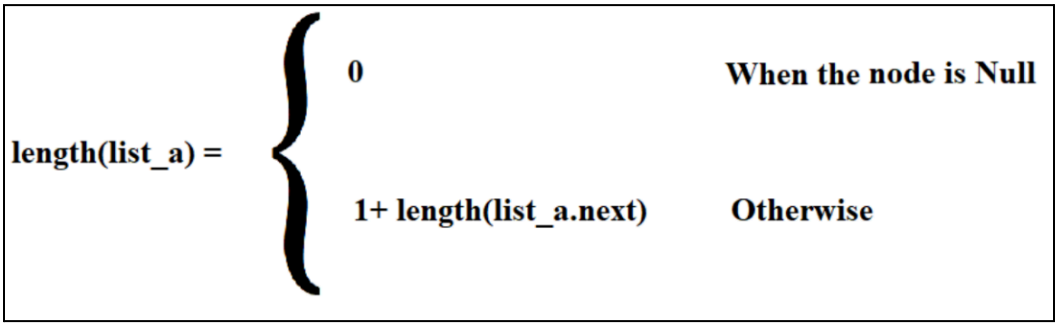
Now, let's see the recursive programming for this

```
def sumList(head):
        if head == None:
                return 0
        else
                return head.elem + sumList(head.next)
```

Length of a Linked List:

A linked list is also a recursive structure: a linked list is either empty or a node followed by the rest of the list. We can also compute the length of a list recursively as follows: the length of a linked list is 1 longer than the rest of the list! The empty list has a length of 0, which is our base case.

We can write the recursion definition as



Now, let's see the recursive programming for this

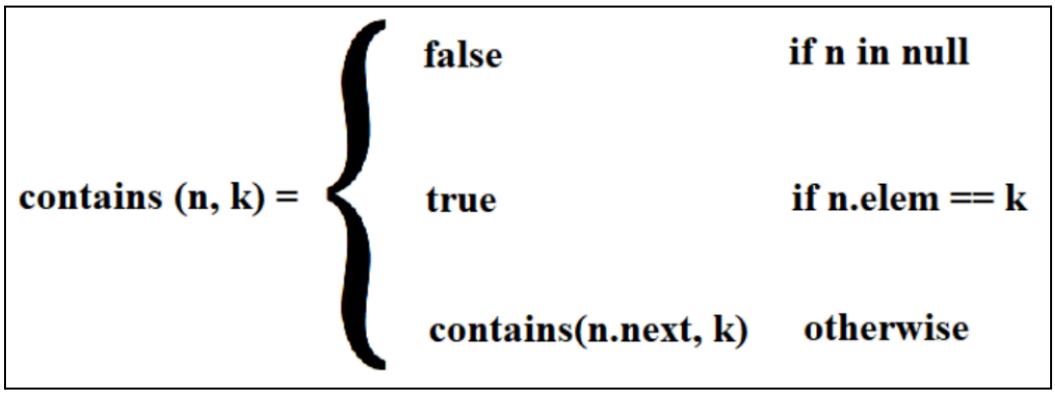
```
def countList(head):
        if head == None:
                return 0
        else
                return 1 + countList(head.next)
```

Sequential search in a sequence:

- Using Linked List:

We look at the first node and check if the key is in that node. If so, done. Otherwise, check the rest of the linked list for the given key. If you search an empty list for any key, the answer is false, so that's our base case.

We can write the recursion definition as

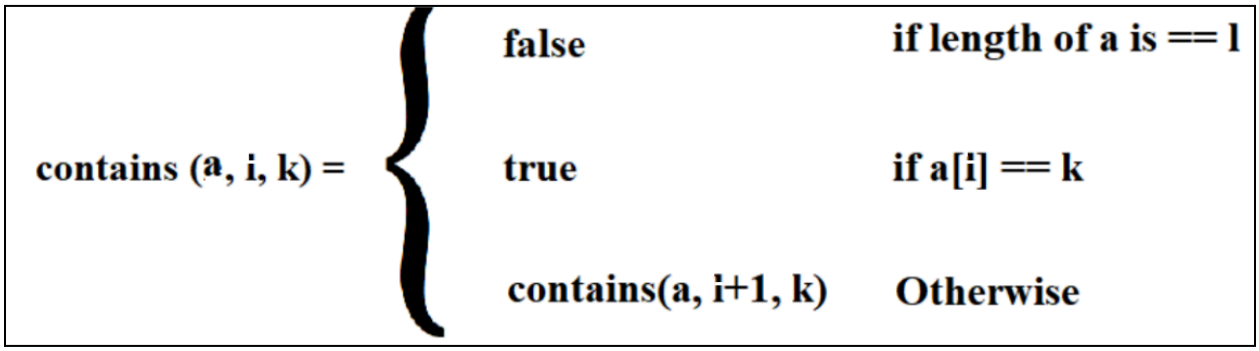


Now, let's see the recursive programming for this

```
def sequentialSearch(head, key):
        if head == None:
                return False
        eliff head.elem == key:
                return True
        else:
                return sequentialSearch(head.next, key)
```

- Using Array:

We can write the recursion definition as



Now, let's see the recursive programming for this

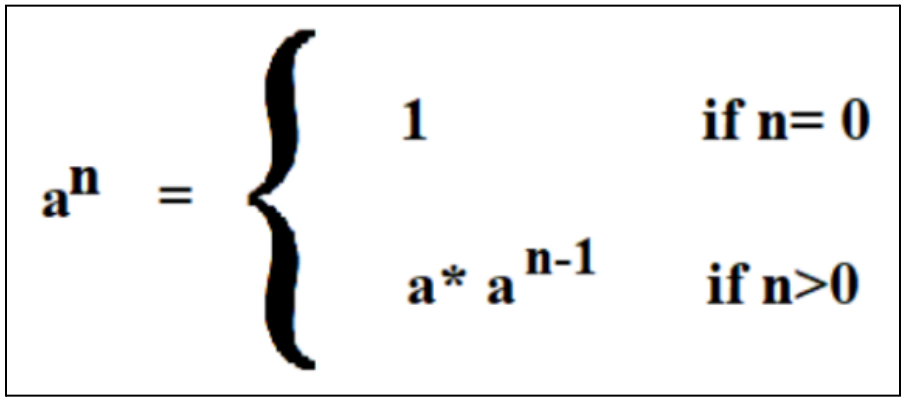
```
def sequentialSearch(arr, index, key):
        if index >= len(arr):
                return False
        elif arr[index] == key:
                return True
        else
                return sequentialSearch(arr, index + 1, key)
```

Exponentiation – a^n:

To compute a^n, we can iteratively multiply a n times, and that's thinking recursively, $a^n = a * a^{n-1}$ and $a^{n-1} = a * a^{n-2}$ and so on. The recursion stops when the exponent n = 0, since by definition $a^0 = 1$.

- Solution 1:

We can write the recursion definition as



Now, let's see the recursive programming for this

```
def power(base, exponent):
        if exponent == 0:
                return 1
        else:
                return base * power(base, exponent - 1)
```

- Solution 2:

As it turns out, there is a much more efficient recursive formulation for the exponentiation of a number. We start by noting that $2^8 = 2^4 * 2^4$, and that $2^7 = 2^3 * 2^3 * 2$.

We can write the recursion definition as

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a^{n/2} * a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} * a^{(n-1)/2} * a & \text{if } n \text{ is odd} \end{cases}$$

Now, let's see the recursive programming for this

```
def power(base, exponent):
        if exponent == 0:
                return 1
        elif  exponent%2==0:
                return power(base, exponent/2) * power(base, exponent/2)
        else
                return power(base, (exponent-1)/2) * power(base,
                        (exponent-1)/2) * a
```

- Solution 3:

Notice how we're computing the following expressions twice in the previous:
1. power(base, exponent/2)
2. power(base, (exponent - 1)/2)

Why not compute it once, and then use the result twice (or as many times as needed)? This will give us a huge boost when we compute the running time of this algorithm. This technique of removing the redundancy in computations by saving the intermediate results in temporary variables is called Memoization.

Now, let's see the recursive programming for this

```
def power(base, exponent):
        if exponent == 0:
                return 1
        elif  exponent%2==0:
                temp = power(base, exponent/2)
                return temp * temp
        else
                temp = power(base, (exponent-1)/2)
                return temp * temp * base
```

Issues/problems to watch out for:

- Inefficient recursion:

The recursive solution for Fibonacci numbers shows massive redundancy, leading to very inefficient computation. The first recursive solution for exponentiation also shows how redundancy shows up in recursive programs. There are ways to avoid computing the same value more than once by caching the intermediate results, either using Memoization or Dynamic Programming (topic for CSE221)

- Space for activation frames:

Each recursive method call requires that its activation record be put on the system call stack. As the depth of recursion gets larger and larger, it puts pressure on the system stack, and the stack may potentially run out of space.

- Infinite recursion:

Ever forgotten a base case? Or miss one of the base cases? You end up with infinite recursion since nothing is stopping your recursion! Whenever you see a Stack Overflow error, check your recursion!