Recursion

Introduction:

Recursive function is no different than a normal function. The motivation to use recursive functions vs non-recursive is that the recursive solutions are usually easier to read and comprehend. Certain applications, like **tree search, directory traversing** etc. are very well suited for recursion. 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. But there can be cases when recursion is the best way.

For example if you need to get the full tree of a directory and store it somewhere. You can write loops but it will be very complicated. And it will be very simple if you use recursion. You'll only get files of root directory, store them and call the same function for each of the subdirectories in root.

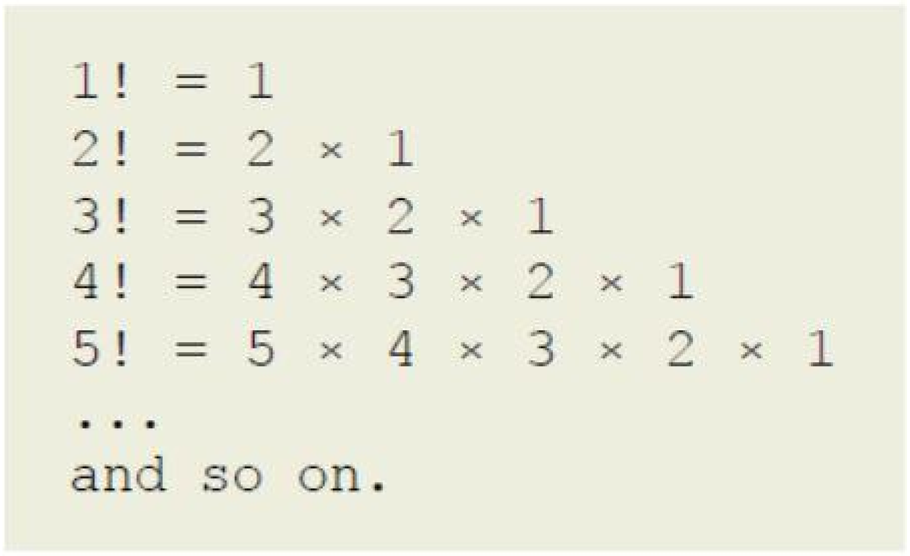
Recursion is a way of thinking about problems, and a method for solving problems. The basic idea behind recursion is the following: it's a method that solves a problem by solving smaller (in size) versions of the same problem by breaking it down into smaller subproblems. Recursion is very closely related to mathematical induction.
We'll start with recursive definitions, which will lay the groundwork for recursive programming. We'll then look at a few prototypical examples of using recursion to solve problems. We'll finish by looking at problems and issues with recursion.

Recursive Definitions:

We'll start thinking recursively with a few recursive definitions:
1. Factorial
2. Fibonacci numbers

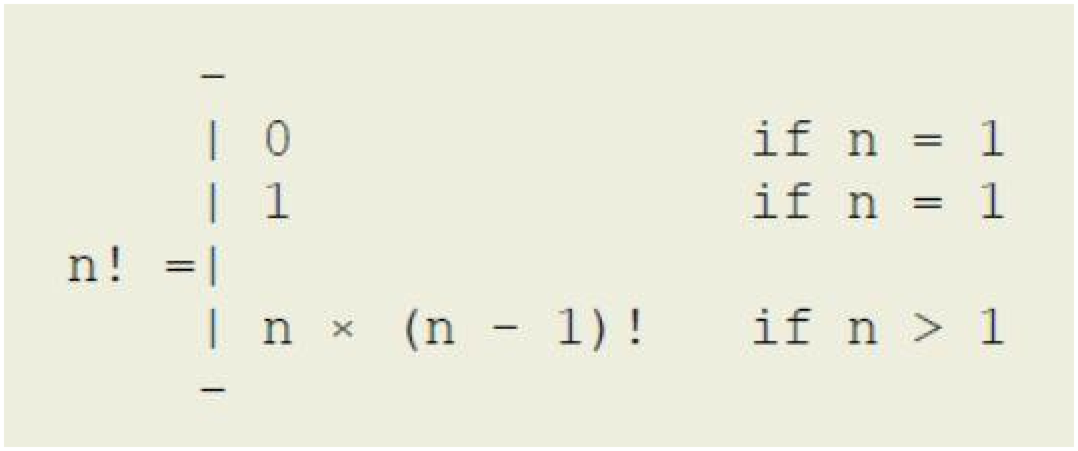
Factorial:

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.



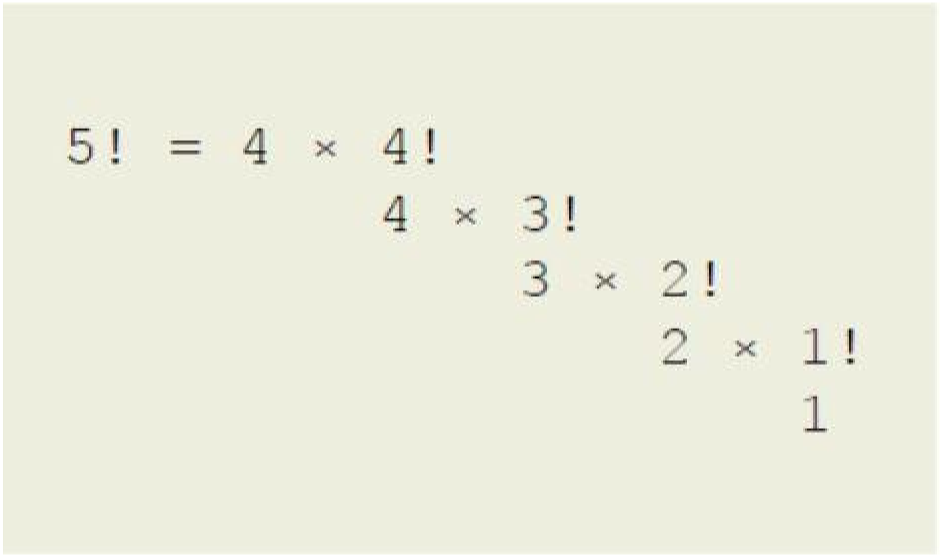
```
1! = 1
2! = 2 × 1
3! = 3 × 2 × 1
4! = 4 × 3 × 2 × 1
5! = 5 × 4 × 3 × 2 × 1
...
and so on.
```

We can easily evaluate n! for any valid value of n by multiplying the values iteratively. However, there is a much more interesting recursive definition quite easily seen from the factorial expressions: 5! is nothing other than 5 * 4!. If we know 4!, we can trivially compute 5!.
4! on the other hand is 4 * 3!, and so on until we have n! = n * (n - 1)!, with 1! = 1 as the base case. The mathematicians have however added the special case of 0! = 1 to make it easier (yes, it does, believe it or not). For the purposes of this discussion, we'll use both 0! = 1 and 1! = 1 as the two base cases.



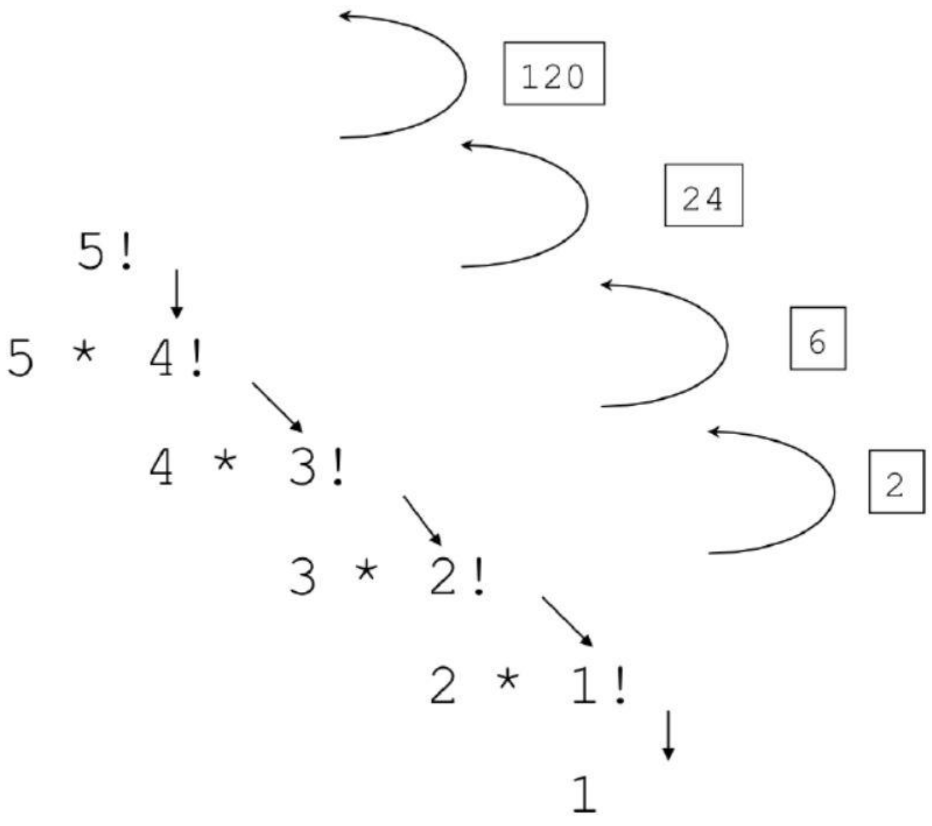
```
      -
      | 0               if n = 1
      | 1               if n = 1
n! =|
      | n × (n - 1)!   if n > 1
      -
```

Now we can expand 5! recursively, stopping at the base condition.



```
5! = 4 × 4!
         4 × 3!
             3 × 2!
                 2 × 1!
                     1
```

The recursion tree for 5! shows the values as well.



Fibonacci numbers:

The Fibonacci numbers are ‹0, 1, 1, 2, 3, 5, 8, 13, ...› (some define it without the leading 0, which is ok too). If we pay closer attention, we see that each number, except for the first two, is nothing but the sum of the previous two numbers. We can easily compute this iteratively, but let's stick with the recursive method. We already have the recursive part of the recursive definition, so all we need is the non-recursive part or the base case. The mathematicians have decided that the first two numbers in the sequence are 0 and 1, which give us the base cases (notice the two base cases).

Now we can write the recursive definition for any Fibonacci number n >= 0.

```
              _
              | 0                         if n = 0
              | 1                         if n = 1
  fib(n)  =   |
              | fib(n-1) + fib(n-2)       n ≥ 2
              _
```

We can now compute fib(5) using this definition.

```
                        fib(5)
                          +
                    /            \
                   /              \
               fib(4)             fib(3)
                 +                  +
            /         \          /       \
           /           \        /         \
         fib(3)      fib(2)   fib(2)   fib(1)
           +            +        +
```

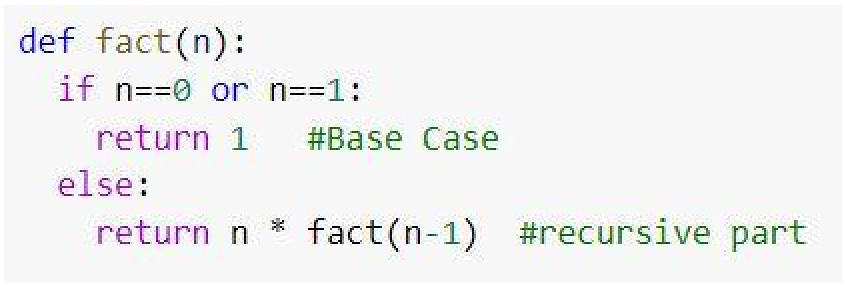
```
       /     \       /    \     /  \
      /       \     /      \   /    \
     f        f    f       f   f    f
     i        i    i       i   i    i
     b        b    b       b   b    b
    (2)      (1)  (1)     (0) (1)  (0)
     +
    / \
   ....
```

Before moving on, you should note how many times we're computing Fibonacci of 3 (fib(3) above), and Fibonacci of 2 (fib(2) above) and so on. This redundancy in computation leads to gross inefficiency, but something we can easily address is Memoization, which is the later topic we study.

Recursive programming:

A recursive function is one that calls itself, since it needs to solve the same problem, but on a smaller sized input. In essence, a recursive function is a function that is defined in terms of itself.

Let's start with computing the factorial. We already have the recursive definition, so the question is how do we convert that to a recursive method in Python that actually computes a factorial of a given non-negative integer. This is an example of functional recursion.

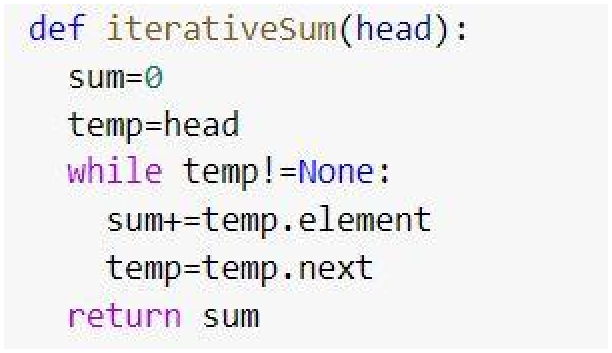


```python
def fact(n):
  if n==0 or n==1:
    return 1   #Base Case
  else:
    return n * fact(n-1)  #recursive part
```

Once you have formulated the recursive definition, the rest is usually quite trivial. This is indeed one of the greatest advantages of recursive programming.

Let's now look at an example of structural recursion, one that uses a recursive data structure. We want to compute the sum of the list of numbers. Linked list = 3-> 8-> 2-> 1-> 13-> 4->None.

The "easiest" way to find the sum of numbers of a linked list iteratively, as shown below.



```python
def iterativeSum(head):
  sum=0
  temp=head
  while temp!=None:
    sum+=temp.element
    temp=temp.next
  return sum
```

But a linked list is a recursive data structure. Hence, by thinking recursively, we note the following:

1. 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.
2. The shortest list has a single number, which has the 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.
Note the key difference between the iterative and recursive approaches: for each number in the list vs the rest of the list.

Now we can write the recursive definition of this problem:

```
        _
       | k                k is the only element
sum =  |
       | k + sum(n.next)  otherwise
        _
```

and using recursive definition, we can write the recursive method in python as shown below:

```
def recursiveSum(head):
  if head.next==None:
    return head.element  #Base case: Linked list has only one element which is head
  else:
    return head.element+recursiveSum(head.next)  #Recursive Part: first value+sum of the rest of the list
```

Examples:

Now we will look at the following examples which can be solved by recursive programming:

1. Length of a String
2. Length of a linked list
3. Sequential search in a sequence
4. Binary search in a sorted array
5. Finding the maximum in a sequence (linear version)
6. Finding the maximum in an array (binary version)
7. Selection sort
8. Insertion sort
9. Exponentiation – a^n

Length of a String:

A character string is a recursive structure: a string is either empty, or a character followed by the rest of the string. This implies that recursion is a natural way to solve string related problems. Let's take the case of computing the length of a string; the length of "abc" is 1 longer than the length of the rest of the string, which is "bc"; the length of "bc" is 1 longer than the length of "c";

and the length of "c" is 1 longer than the length of "", which is the empty string (and hence has a length of 0). We now have our recursive case and a base case! An example of what is known as structural recursion.
So, the length of a String has the following recursive formulation: the length is 1 plus the length of the rest of the String. In Python, the String provides a slicing method to extract the rest of the String: string[1:] produces the substring from index 1 onwards. The recursion stops when the String is empty, which gives us the base case.

Recursive definition is given below:

```
          _
          | 0               if string s is empty
len(s) =  |
          | 1 + len(rest)   otherwise
          _
```

Recursive method is shown below:

```python
def strLength(str):
  if len(str)==0:  #length of the string is zero that means it is empty string
    return 0       #Base case
  else:
    return 1+strLength(str[1: ])   #Recursive part
```

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.

Recursive definition is given below:

```
           _
          | 0               if l is an empty list
 len(l) = |
          | 1 + len(rest)   otherwise
           _
```

Recursive method is shown below:

```python
def listLength(head):
  if head==None:  #empty linked list
    return 0      #base case
  else:
    return 1+listLength(head.next)  #Recursive part
```

Sequential search in a sequence:

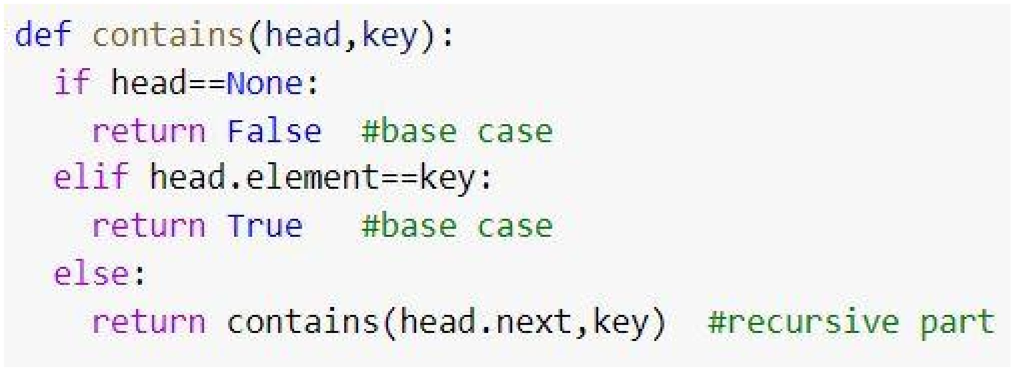
How would you find something in a linked list? Well, look at the first node and check if the key is in that node. If so, done. Otherwise, check the rest of the linked list for the given key. If you search an empty list for any key, the answer is false, so that's our base case.

This is almost exactly the same, at least in form, as finding the length of a linked list, and also an example of structural recursion.

Recursive definition is given below:

```
                  _
                 | false              if list l is empty
                 | true               if l.item = k
 contains(l,k) = |
                 | contains(l.next,k) otherwise
                  _
```

Recursive method is shown below:

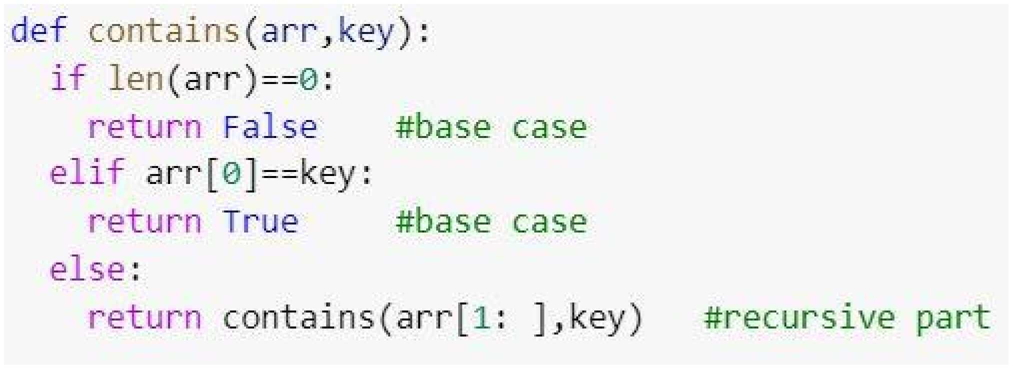


```
def contains(head,key):
  if head==None:
    return False  #base case
  elif head.element==key:
    return True   #base case
  else:
    return contains(head.next,key)  #recursive part
```

What if the sequence is an array? How do we deal with the rest of the array part then? We can handle it in two ways:

1. We can create a new array that contains a copy of the rest of the array, and pass that instead to the next level of recursion. This is of course very inefficient and to be avoided at all costs. In programming languages such as Python which provide efficient array slicing, this would be the way to go.

Recursive method:



```
def contains(arr,key):
  if len(arr)==0:
    return False    #base case
  elif arr[0]==key:
    return True     #base case
  else:
    return contains(arr[1: ],key)   #recursive part
```

2. Without array slicing, we can solve this problem. We can maintain a left index, along with a reference to the array, that is used to indicate where the beginning of the array is.

Initially, left = 0, meaning that the array begins at the expected index of 0. Eventually, a value of left = len(array) - 1 means that the rest of the array is simply the last element, and then left = len(array) means that it's an 0-sized array.

Recursive definition:

```
                     -
                     | false                        if l ≥ a.length
                     | true                         if a[l] = k
contains(a,l,k)  =   |
                     | return contains(a,l+1,k)  otherwise
                     -
```

Recursive method:

```
def contains(arr,left,key):
  if left>=len(arr):
    return False    #base case
  elif arr[left]==key:
    return True     #base case
  else:
    return contains(arr,left+1,key)   #recursive part
```

We start the search with **contains(arr, 0, key)**, and then at each step, the rest of the array is given by advancing the left boundary(index).

Instead of just a yes/no answer, what if we wanted the position of the key in the array? We can simply **return left** instead of true as the position if found, or use a sentinel -1 instead of false if not.

Binary search in a sorted array:

Given the abysmal performance of sequential search, we obviously want to use binary search whenever possible. Of course, the pre-conditions must be met first:

1. The sequence must support random access (an array that is)
2. The data must be sorted

Now we can formulate a recursive version of binary search of a key in an array data between the positions left(l) and right(r) (inclusive):

```
if the array is empty (if l > r that is):
  return false
else:
  Find the position of the middle element: mid = (l + r)/2
  If key == data[mid], then return true
  If key > data[mid], the search the right half data[mid+1..r]
  If key < data[mid], the search the left half data[l..mid-1]
```

Now we can write the recursive definition, and translate that to Python.

```
                        -
                        | false                   if l > r
                        | true                    if k = a[mid]
 contains(a,l,r,k)  =   |
                        | contains(a,mid+1,r,k)   if k > a[mid]
                        | contains(a,l,mid-1,k)   if k < a[mid]
                        -
```

Recursive method:

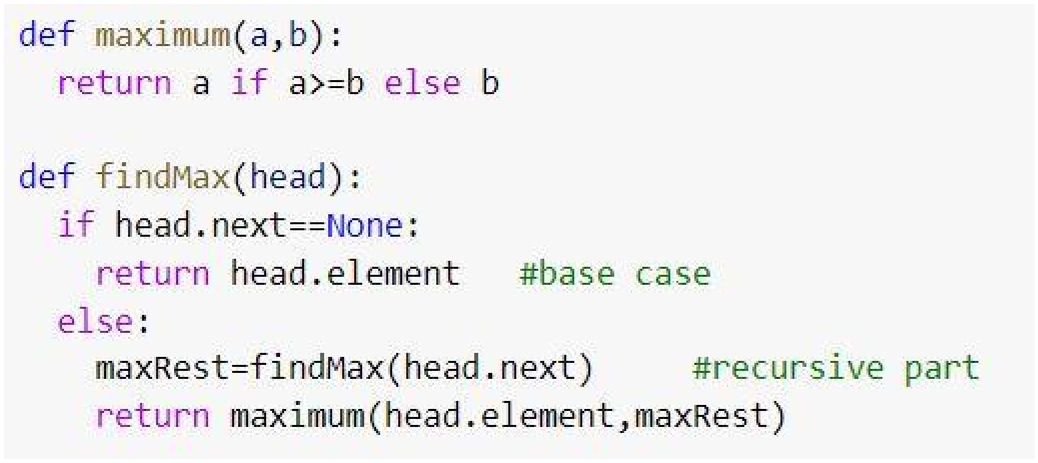
```python
def contains(arr, left, right, key):
  if left > right:
    return False    #base case
  else:
    mid=(left+right)//2
    if key==arr[mid]:
      return True   #base case
    elif key > arr[mid]:
      return contains(arr, mid+1, right, key) #recursive part
    else:
      return contains(arr, left, mid-1, key)  #recursive part
```

We start the search with **contains(arr, 0, len(arr) - 1, key)**, and then at each step, the rest of the array is given by one half of the array – left or right, depending on the comparison of the key with the middle element.
Instead of just a yes/no answer, what if we wanted the position of the key in the array? We can simply **return mid** as the position instead of true if found, or use a sentinel -1 instead of false if not.

Finding the maximum in a sequence (linear version):

Given a sequence of keys, our task is to find the maximum key in the sequence. This is of course trivially done iteratively (for a non-empty sequence): take the 1st one as maximum, and then iterate from the 2nd to the end, exchanging the current with the maximum if the current is larger than the maximum.
Formulating this recursively: the maximum key in a sequence is the larger of the following two:
1. the 1st key in the sequence
2. the maximum key in the rest of the sequence

Once we have (recursively) computed the maximum key in the rest of the sequence, we just have to compare the 1st key with that, and we have our answer! The base case is also trivial (for a non-empty sequence): the maximum key in a single-element sequence is the element itself.
Since the rest of the sequence does not need random access, we can easily do this for a linked list or an array. Let's write it for a linked list first.



```python
def maximum(a,b):
  return a if a>=b else b

def findMax(head):
  if head.next==None:
    return head.element   #base case
  else:
    maxRest=findMax(head.next)     #recursive part
    return maximum(head.element,maxRest)
```

We start to find the maximum with **findMax(head)** (where head is the reference to the first node of the list), and then at each step, the rest of the array is given by advancing the head reference.

What if the sequence is an array? Well, then we use the same technique we've used before — use a left (and optionally right) boundary to window into the array.

Recursive method:

```
def maximum(a,b):
  return a if a>=b else b

def findMax(arr, left):
  if left == len(arr)-1:
    return arr[left]    #base case
  else:
    maxRest=findMax(arr, left+1)    #recursive part
    return maximum(arr[left], maxRest)
```

We start to find the maximum with **findMax(arr, 0)**, and then at each step, the rest of the array is given by advancing the left boundary(index).

Finding the maximum in an array (binary version):

If our sequence is an array, we can also find the maximum by formulating the following recursive definition: the maximum key in an array is the larger of the following two:

1. the maximum key in the left half of the array
2. the maximum key in the right half of the array

```
def maximum(a,b):
  return a if a>=b else b

def findMax(arr, left, right):
  if left == right:
    return arr[left]    #base case
  else:
    mid = (left+right)//2
    maxLeftHalf=findMax(arr, left, mid)     #recursive part
    maxRightHalf=findMax(arr, mid+1, right) #recursive part
    return maximum(maxLeftHalf, maxRightHalf)
```

We start to find the maximum with **findMax(arr, 0, len(arr)-1)**, and then at each step, the array is divided into two halves.

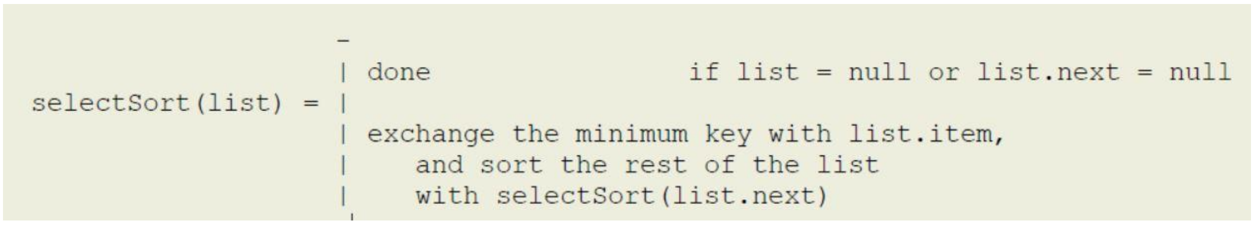
Advance Recursion Part 1

Selection sort:

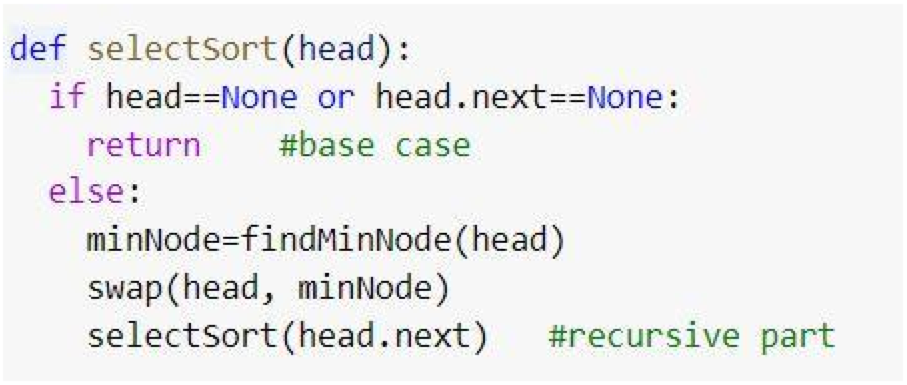
How about sorting a sequence recursively? Since it does not require random access, we'll look at recursive versions for both linked lists and arrays.
The basic idea behind selection sort is the following: put the 1st minimum in the 1st position, the 2nd minimum in the 2nd position, the 3rd minimum in the 3rd position, and so on until each key is placed in its position according to its rank. To come up with a recursive formulation, the following observation is the key:

Once the 1st minimum in the 1st position, it will never change its position. Now all we have to do is to sort the rest of the sequence (from 2nd position onwards), and we'll have a sorted sequence.

Now we can write the recursive definition for a linked list, and translate it to Python.



```
                       -
                       | done                 if list = null or list.next = null
 selectSort(list)  =   |
                       | exchange the minimum key with list.item,
                       |    and sort the rest of the list
                       |    with selectSort(list.next)
```



```python
def selectSort(head):
  if head==None or head.next==None:
    return    #base case
  else:
    minNode=findMinNode(head)
    swap(head, minNode)
    selectSort(head.next)   #recursive part
```

Here, be careful about one thing: the swap method means we are not exchanging the nodes rather than we are exchanging the element of nodes.

```
def swap(a,b):
  temp=a.element
  a.element=b.element
  b.element=temp
```

We sort a list headed by head reference by calling **selectSort(head)**. Note that we're not finding the minimum key, but rather the node that contains the minimum key since we need to exchange the left key with the minimum one. We can write that iterative of course, but a recursive one is simply more fun. This is of course almost identical to finding the maximum in a sequence, with two differences: we find the minimum, and we return the node that contains the minimum, not the actual minimum key.

```
                    -
                    | list.item                    if list.next = null
findMinNode(list) = |
                    | get the node that contains the minimum in
                    |   the rest with findMinNode(list.next)), and
                    |   return either list.item or the other one
                    |   depending on which one has the smaller key
                    -
```

Recursive method of findMinNode:

```
def findMinNode(head):
  if head.next==None:
    return head       #base case
  else:
    minNode=head
    minNodeRest=findMinNode(head.next)    #recursive part
    if minNodeRest.element < head.element:
      minNode=minNodeRest
    return minNode
```

If the sequence is an array, then we have to use the left (and optionally right) boundary to window into the array.

```
def selectSort(arr, left):
  if left==len(arr)-1:
    return        #base case
  else:
    minIdx=findMinIdx(arr, left, len(arr)-1)
    swap(arr, left, minIdx)
    selectSort(arr, left+1)     #recursive part
```

Here, be careful about one thing: the swap method means we are not exchanging the indices rather than we are exchanging the element of these indices in the array.

```
def swap(arr, a, b):
  temp=arr[a]
  arr[a]=arr[b]
  arr[b]=temp
```

We sort an array(arr) by calling **selectSort(arr, 0)**. And we can write findMinIndex recursively of course, which is again almost exactly the same as either finding the maximum in a sequence, or finding the maximum in an array — we'll use the binary method just for illustration.

Recursive method of findMinIdx:

```
def findMinIdx(arr, left, right):
  if left==right:
    return left       #base case
  else:
    mid=(left+right)//2
    minIdxLeft=findMinIdx(arr, left, mid)     #recursive part
    minIdxRight=findMinIdx(arr, mid+1, right) #recursive part
    minIdx=minIdxLeft
    if arr[minIdxRight] < arr[minIdxLeft]:
      minIdx=minIdxRight
    return minIdx
```

Insertion Sort:

Insertion works by inserting each new key in a sorted array so that it is placed in its rightful position, which now extends the sorted array by the new key. In the beginning, there is a single key in the array, which by definition is sorted. Then the second key arrives, which is then inserted into the already sorted array (of one key at this point), and now the sorted array has two keys. Then the third key arrives, which is inserted into the already sorted array, creating a sorted array of 3 keys. And so on. Iteratively, it's a fairly simple operation. The question is how can we formulate this recursively. The following observation is the key to this recursive formulation:

Given an array of n keys, sort the first n-1 keys and then insert the nth key in the sorted array such that all n keys are now sorted.

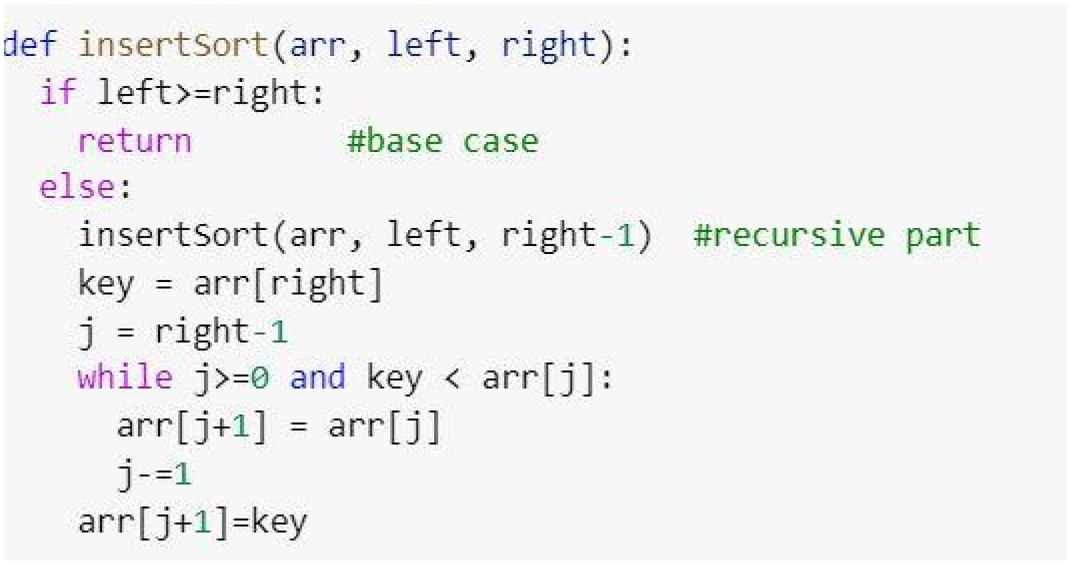
Note how the recursive part comes first, and then the nth key is inserted (iteratively) into the sorted array.

Unlike recursive selection sort, we're going from right to left in the recursive version of insertion sort. Note that we don't need random access, but need to be able to iterate in both directions (reverse direction to insert the new key in the sorted partition). So, if we're sorting a linked list using the insertion sort algorithm, the list must be doubly-linked.

If sorting an array, we can formulate the solution as follows:

```
                      -
                      | done                if l >= r
insertSort(a,l,r) =   |
                      | recursively sort the first n-1 keys using
                      |    insertSort(a,l,r-1), and then insert the
                      |    nth key (index r in
                      |    case) such that the result is sorted.
                      -
```

Recursive method of insertion sort:



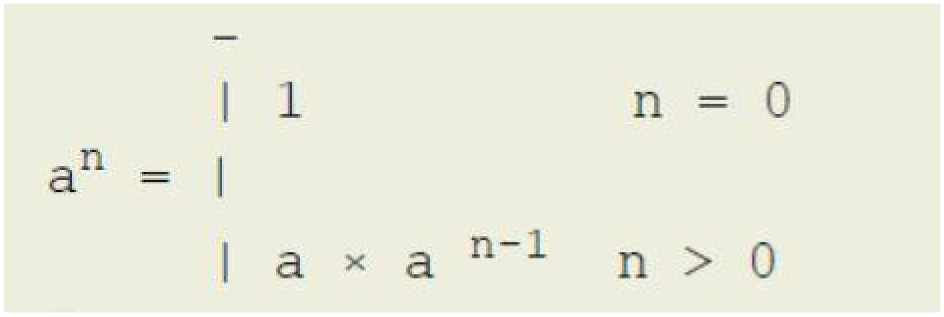
```
def insertSort(arr, left, right):
  if left>=right:
    return        #base case
  else:
    insertSort(arr, left, right-1)  #recursive part
    key = arr[right]
    j = right-1
    while j>=0 and key < arr[j]:
      arr[j+1] = arr[j]
      j-=1
    arr[j+1]=key
```

We sort an array(arr) by calling **insertSort(arr, 0, len(arr)- 1)**.

Exponentiation – a^n:

This is another example of functional recursion. To compute a^n, we can iteratively multiply a n times, and that's that. 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.

Recursive definition:



$$a^n = \begin{cases} 1 & n = 0 \\ a \times a^{n-1} & n > 0 \end{cases}$$

Recursive method:

```
def exp(a, n):
  if n==0:
    return 1    #base case
  else:
    return a * exp(a, n-1)    #recursive part
```

As it turns out, there is actually 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 generalize that with the following recursive definition, and its implementation.

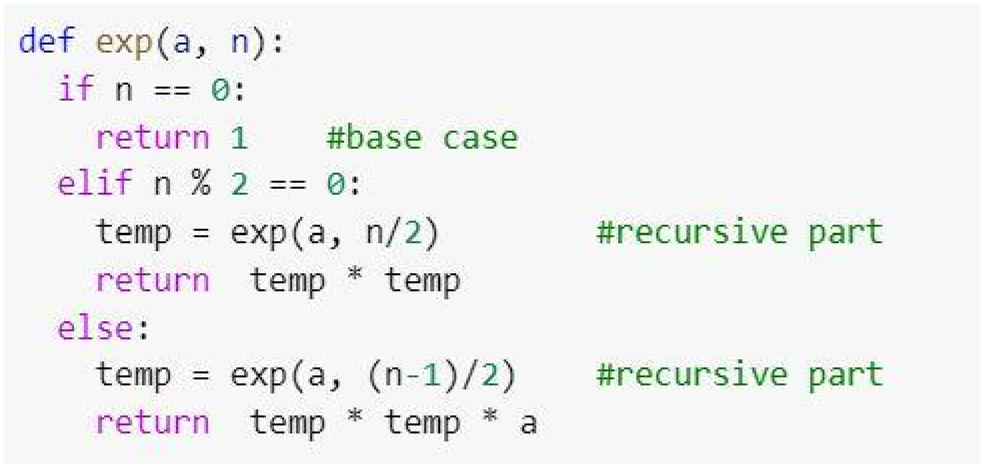
$$a^n = \begin{cases} 1 & n = 0 \\ a^{n/2} \times a^{n/2} & n \text{ is even} \\ a^{(n-1)/2} \times a^{(n-1)/2} \times a & n \text{ is odd} \end{cases}$$

Recursive method:

```
def exp(a, n):
  if n == 0:
    return 1    #base case
  elif n % 2 == 0:
    return exp(a, n/2) * exp(a, n/2)                  #recursive part
  else:
    return exp(a, (n-1)/2) * exp(a, (n-1)/2) * a     #recursive part
```

But why would we care about this formulation over the more familiar one? If we solve for the running time, both solutions take the same time, so what is the benefit of this approach? Notice how we're computing the following expressions twice:
1. exp(a, n/2)
2. exp(a, (n - 1)/2)

Why not compute it once, and then use the result twice (or as many times as needed)? We can, and as we'll find out, that'll give us a huge boost when we compute the running time of this algorithm. Here's the modified version.



```
def exp(a, n):
  if n == 0:
    return 1    #base case
  elif n % 2 == 0:
    temp = exp(a, n/2)        #recursive part
    return  temp * temp
  else:
    temp = exp(a, (n-1)/2)    #recursive part
    return  temp * temp * a
```

All we're doing is removing the redundancy in computations by saving the intermediate results in temporary variables. This is a simple case of a technique known as Memoization, which is the next topic we study. Remember that we've already seen such redundancy in recursive computation — when computing the Fibonacci numbers.

Issues/problems to watch out for

1. **Inefficient recursion:**
The recursive solution for Fibonacci numbers outlined in these notes shows massive redundancy, leading to very inefficient computation. The 1st 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 (a top-down technique — see next topic), or Dynamic Programming (a bottom-up technique — survive this semester to enjoy it in the next one).

2. **Space for activation frames:**
Each recursive method call requires that it's 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.

3. **Infinite recursion:**
Ever forgot a base case? Or miss one of the base cases? You end up with infinite recursion, since there is nothing stopping your recursion! Whenever you see a Stack Overflow error, check your recursion!

Advanced Recursion Part 2
Optimizing Recursive Program
Memoization

Introduction:

To develop a recursive algorithm or solution, we first have to define the problem (and the recursive definition, often the hard part), and then implement it (often the easy part). This is called the **top-down** solution, since the recursion opens up from the top until it reaches the base case(s) at the bottom.
Once we have a recursive algorithm, the next step is to see if there are **redundancies** in the computation— that is, if the same values are being computed multiple times. If so, we can benefit from **memoizing** the recursion. And in that case, we can create a memoized version and see what savings we get in terms of the running time.

Recursion has certain overhead costs that may be minimized by transforming the memoized recursion into an iterative solution. And, finally, we see if there are further improvements that we can make to improve the time and space complexity.

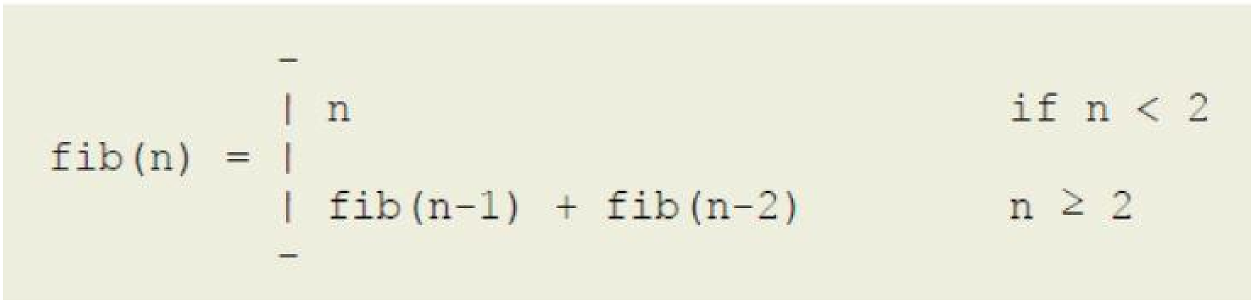
The steps are as follows:
1. write down the recursion,
2. implement the recursive solution,
3. memoize it,
4. transform into an iterative solution, and finally
5. make further improvements.

Example using the Fibonacci sequence:

To see this in action, let's take Fibonacci numbers as an example. Fortunately for us, the mathematicians have already defined the problem for us – the Fibonacci numbers are ‹ 0, 1, 1, 2, 3, 5, 8, 13,... › (some define it without the leading 0, which is ok too). Each number, except for the first two, is nothing but the sum of the previous two numbers. The first two are by definition 0 and 1. These two facts give us the recursive definition to compute the n^{th} Fibonacci number for some n >= 0.

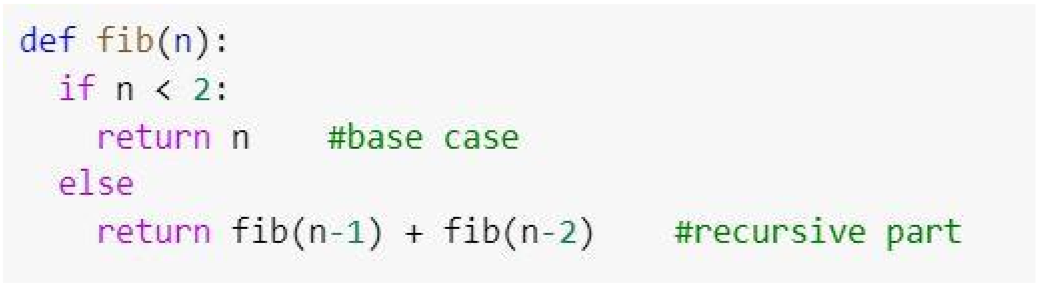
Let's go through the 5 steps below.

Step 1: Write or formulate the recursive definition of the nth Fibonacci number (defined only for n>=0)



```
          -
          | n                             if n < 2
fib(n) =  |
          | fib(n-1) + fib(n-2)           n ≥ 2
          -
```

Step 2: Write the recursive implementation. This usually follows directly from the recursive definition



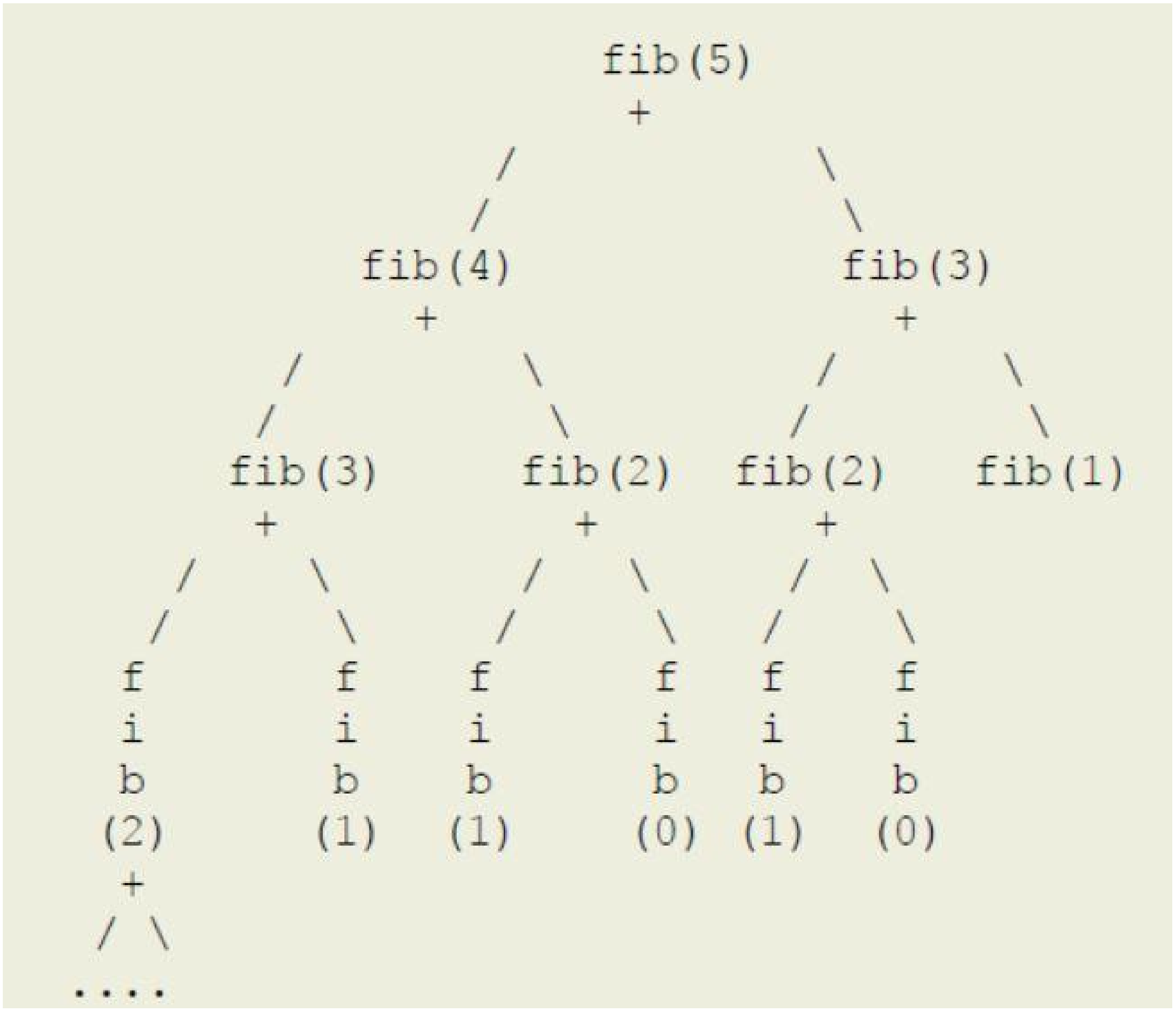
```
def fib(n):
  if n < 2:
    return n    #base case
  else
    return fib(n-1) + fib(n-2)    #recursive part
```

Step 3: Memoize the recursion

Would this recursion benefit from memoization? Well, let's see by "unrolling" the recursion **fib(5)** a few levels:

Now you should notice something very interesting — we're computing the same fibonacci number quite a few times. We're computing fib(2) 3 times and fib(3) 2 times. Is there any reason why we couldn't simply save the result after computing it the first time, and re-using it each time it's needed afterward?

This is the primary motivation for **memoization** – to avoid re-computing overlapping subproblems. In this example, fib(2) and fib(3) are overlapping subproblems that occur independently in different contexts.



```
                     fib(5)
                       +
                 /            \
                /              \
            fib(4)             fib(3)
              +                  +
         /        \           /       \
        /           \        /         \
      fib(3)      fib(2)   fib(2)   fib(1)
        +            +        +
    /     \       /    \    /   \
   /       \     /      \   /    \
  f        f    f       f   f    f
  i        i    i       i   i    i
  b        b    b       b   b    b
 (2)      (1)  (1)     (0) (1)  (0)
  +
 / \
....
```

Memoization certainly looks like a good candidate for this particular recursion, so we'll go ahead and memoize it. Of course, the first question is how we save the results of these overlapping subproblems that we want to reuse later on.

The basic idea is very simple — before computing the ith fibonacci number, first check if that has already been solved; if so, just look up the answer and return; if not, compute it and save it before returning the value. We can modify our fib method accordingly (using some pseudocode for now). Since fibonacci is a function of 1 integer parameter, the easiest is to use a 1-dimensional array or table with a capacity of **n + 1** (we need to store fib(0) ... fib(n), which requires n + 1 slots) to store the intermediate results.

What is the cost of memoizing a recursion? It's the space needed to store the intermediate results. Unless there are overlapping subproblems, memoizing a recursion will not buy you anything at all, and in fact, cost you more space for nothing!
Remember this — **memoization trades space for time.**

Let's try out our first memoized version. (M_fib below stands for **"memoized fibonacci"**).

```
def M_fib(n):
  # Assume that we have some "global" array (also called a table)
  # F with n+1 capacity. Why can F not be a local variable?

  if n < 2:
    return n    #base case
  else:

    #Compute (and save) if it's not already computed

    if (F[n] is empty)  # <<<< NOTE PSEUDOCODE!
      F[n] = M_fib(n-1) + M_fib(n-2)    #recursive part

    # Now just return the computed (and saved) value.
    return F[n]
```

This is all that we need to avoid redundant computations of overlapping subproblems. The first time fib(3) is called, it'll compute the value and save the answer in F[3], and then subsequent calls would simply return F[3] without doing any work at all!

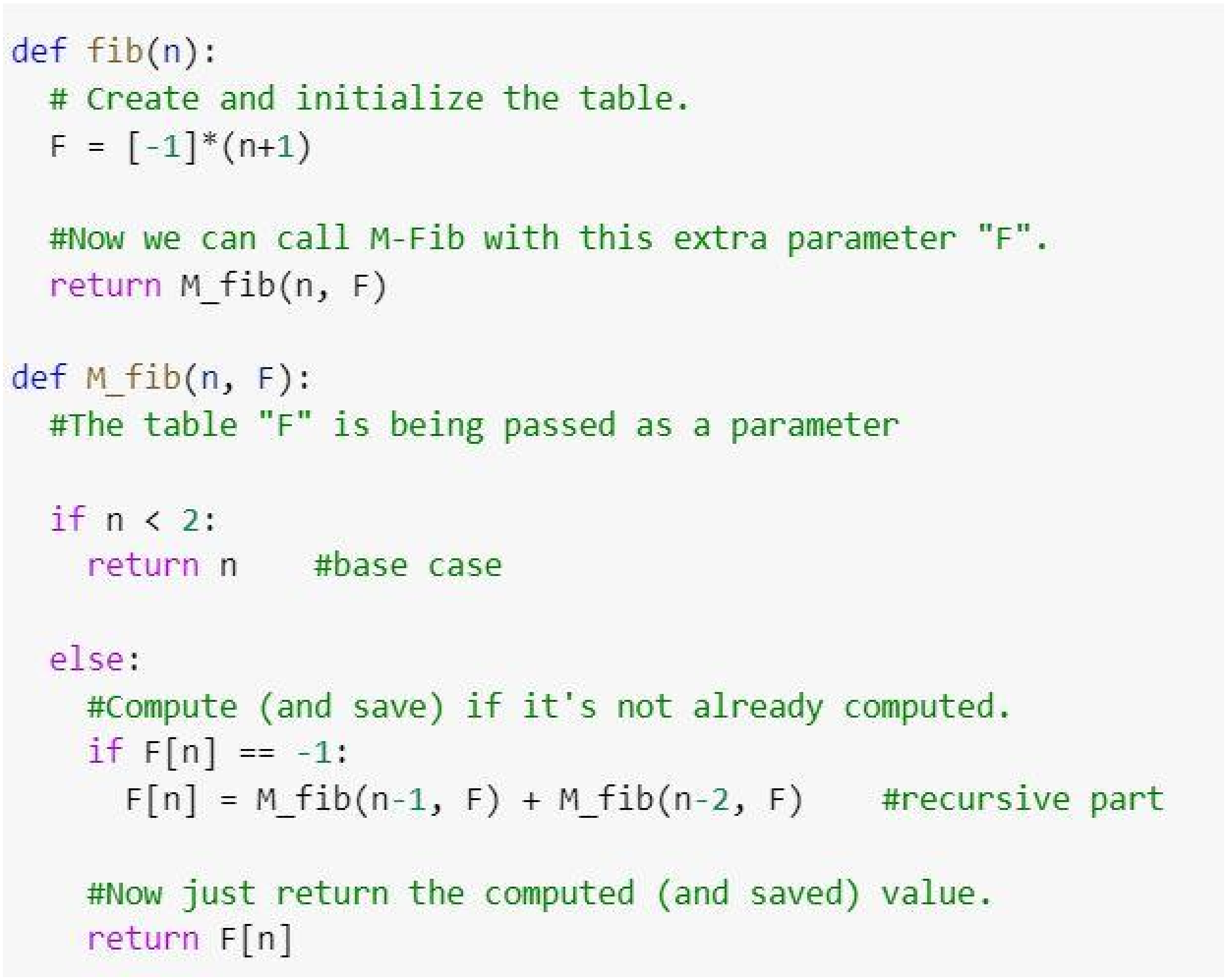
There are a few details we've left out at this point:
1. Where would we create this "table" to store the results?
2. How can we initialize each element of F to indicate that the value has not been computed/saved yet?(Note the **"is empty"** in the code above).

Let's take these one at a time.

1. The "fib" method is a function of a single parameter — n, so if we wanted to save the intermediate results, all we need is an array that goes from 0 ... n (i.e., of n + 1 capacity). Since the local variables within a method are created afresh each time the method is called, F cannot be a local array. We can either use an instance variable within an object, or create an array in the caller of fib(n), and then pass the array to fib (in which case we'll have to modify fib to have another parameter).

2. We need to use a sentinel which will indicate that the value has not been computed. Since it's an array of integers, we can't use null (which is the sentinel used to indicate the absence of an object).
However, we know that the nth fibonacci number is a non-negative integer, so we can use any negative number as the sentinel. **Let's choose -1**. So, let's have an array F of n+1 capacity that holds all the values of the intermediate results we need to compute the nth Fibonacci number.
We can have a wrapper method, which creates this array or table, initializes the table and passes it onto M_fib as a parameter.

First, the wrapper method called **fib**, which basically sets up the table for **M_fib**, and calls it on the user's behalf.

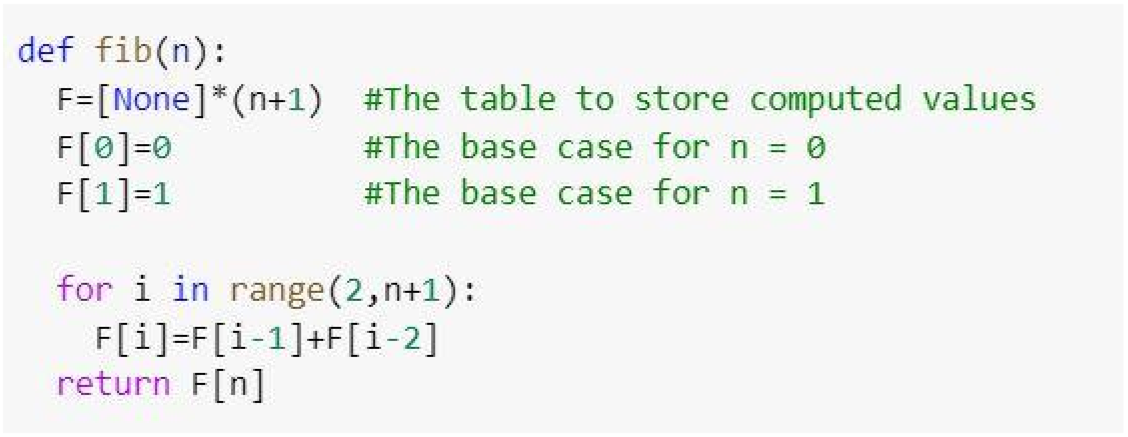


```
def fib(n):
  # Create and initialize the table.
  F = [-1]*(n+1)

  #Now we can call M-Fib with this extra parameter "F".
  return M_fib(n, F)

def M_fib(n, F):
  #The table "F" is being passed as a parameter

  if n < 2:
    return n    #base case

  else:
    #Compute (and save) if it's not already computed.
    if F[n] == -1:
      F[n] = M_fib(n-1, F) + M_fib(n-2, F)    #recursive part

    #Now just return the computed (and saved) value.
    return F[n]
```

To compute the 5th fibonacci number, we simply call **fib(5)**, which in turn calls **M_fib(5, F)** to compute and return the value.
Now that we have a memoized fibonacci, the next question is to see if we can improve the space overhead of memoization.

Step 4: Convert the recursion to iteration – the bottom-up solution.

To compute the 5th fibonacci number, we wait for 4th and 3rd, which in turn wait for 2nd, and so on until the base cases of n = 0 and n = 1 return the values which move up the recursion stack. Other than the n = 0 and n = 1 base cases, the first value that is actually computed and saved is n = 2, and then n= 3, and then n = 4 and finally n = 5. Then why not simply compute the solutions for n = 2, 3, 4, 5 by iterating (using the base cases of course), and fill in the table from left to right? This is called the **bottom-up** solution since the recursion tree starts at the bottom (the base cases) and works its way up to the top of the tree (the initial call). The bottom-up technique is more popularly known as **dynamic programming**, a topic that we will spend quite a bit of time on next semester!



```python
def fib(n):
  F=[None]*(n+1)  #The table to store computed values
  F[0]=0          #The base case for n = 0
  F[1]=1          #The base case for n = 1

  for i in range(2,n+1):
    F[i]=F[i-1]+F[i-2]
  return F[n]
```

You should convince yourself that this is indeed a solution to the problem, only using iteration instead of memoized recursion. Also, that it solves each subproblem (e.g., fib(3) and fib(2)) exactly once, and re-uses the saved answer.

This one avoids the overhead of recursion by using iteration, so tends to run much faster.

Can we improve this any further?

Step 5: improving the space-requirement in the bottom-up version

The nth Fibonacci number depends only on the (n - 1)th and (n - 2)th Fibonacci numbers. However, we are storing ALL the intermediate results from 2 ... n - 1 Fibonacci numbers before computing the nth one. What if we simply store the last two? In that case, instead of having an array of n + 1 capacity, we need just two instance variables (or an array with 2 elements). Here's what the answer may look like.

```python
def fib(n):
  f_2 = 0     #The (n-2)th value
  f_1 = 1     #The (n-1)th value
  f = n
  #The result f is initialized to n (why? So that it works when n is 0 or 1).

  for i in range(2, n+1):
    f = f_1 + f_2

    #Now update the f_1 and f_2 for the next iteration (if any).
    f_2 = f_1
    f_1 = f

  return f
```