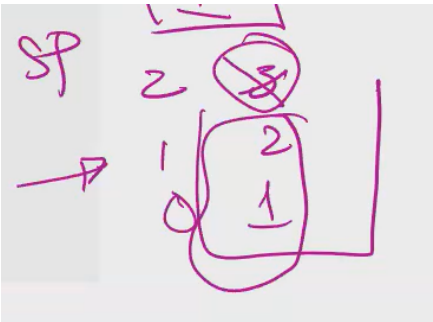
Solution to HW3

Implement Stack using Pylist : Constructor definition

```
    #MUST USE ONLY PYTHON LIST
    self._a = []
    self._sp = 0
    self._maxspace = 0
```

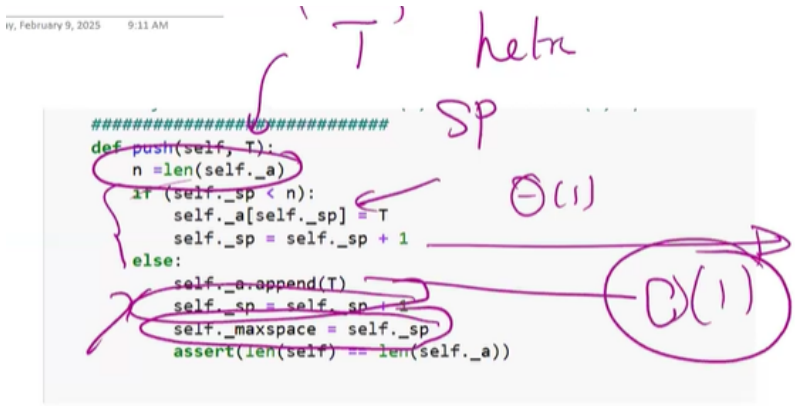
- We are not using typical len function over here as from the stack point of view if we pop a variable then after sometime the pointer _sp points to the last element and hence gives us the length.

```
##############################
# WRITE All private functions BELOW
# YOU CAN HAVE ANY NUMBER OF PRIVATE FUNC
##############################
def __len__(self)->'int':
    return self._sp ;
```



Push Solution explanation

- N points put to the stack pointer, compares it with the len of the list , and adds value accordingly. if there is sp is lesser than actual length it adds in that value with theta(1) complexity otherwise it appends with amortized(1) complexity .



- Space routine helps us understand the max space allocated in the list despite multiple add and pop operations in stack.
- Since list is dynamic array , it 's never full hence _isFull is always false
- Empty routine calls below defined len function and accordingly says if its empty or not
- Top points to the pointer sp and gives the respective last value of the stack

```
  def pop(self)->'int':
      if (self.empty()):
          return None #to make Leetcode happy
      self._sp = self._sp - 1
      return self._a[self._sp] #to make Leetcode happy

  def top(self)->'int':
      if (self.empty()):
          return None
      return self._a[self._sp-1]

  def empty(self)->'Bool':
      if (len(self) == 0):
          return True
      return False

  def is_full(self)->'Bool':
      return False #our stack can never be full

  def space(self)->'int':
      return self._maxspace
```

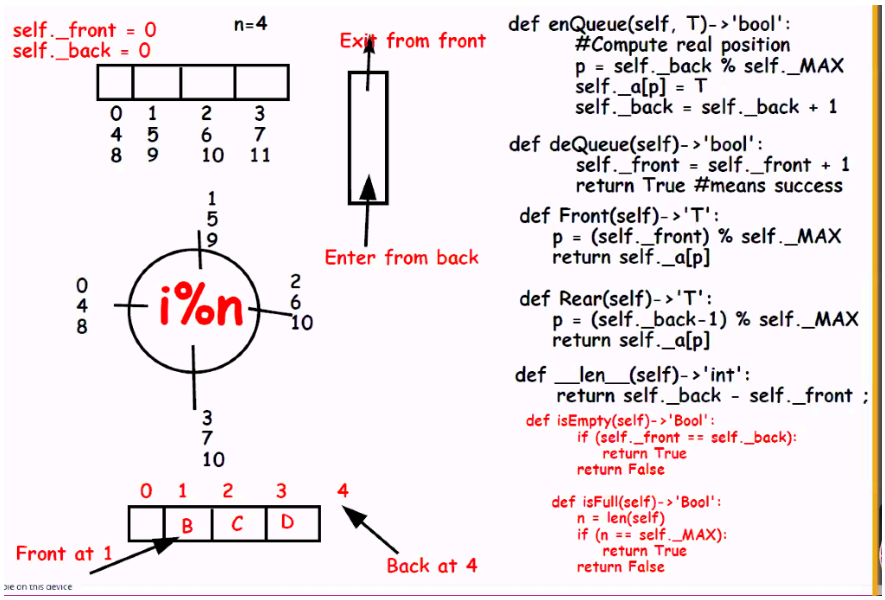
Queue Explanation

```
class Queue():
    def __init__(self,k:'int'):
        # You cannot change init
        # You must use Python List
        # You cannot add anything here
        # Must use all spaces
        self._a = [None]*k #CANNOT CHANGE THIS
        self._MAX = k
        ## YOU CAB HAVE YOUR PRIVATE DATA MEMBER HERE
        self._front = 0
        self._back = 0
```

Explanation of circular queue

- So The crux element is the mod of list elements occupy the same positions . In ourcase size of the row which is n=4, so 0 4,8 are same positions . Any number given we do mod

by and allocate the position . We take in the concept of entering through front and _front lies between n to n-1 and _back extends accordingly .

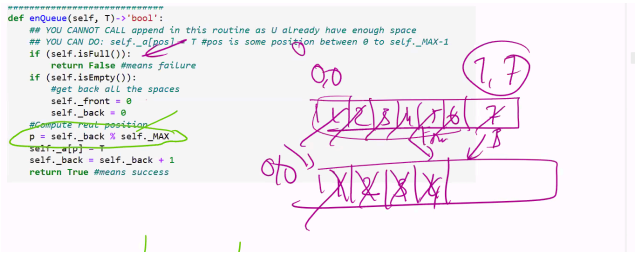



```
def __len__(self)->'int':
    return self._back - self._front ;
```

- The length routine checks the two pointers and gives the length accordingly . In case of no elements case both front and back are placed as 0 so the length is 0 .

```
def isEmpty(self)->'Bool':
    if (self._front == self._back):
        return True
    return False

def isFull(self)->'Bool':
    n = len(self)
    if (n == self._MAX):
        return True
    return False
```

- Is full an is Empty routines .



- The logic behind self._empty is that once a bunch of numbers have been entered and then serviced , the front pointer and back pointer point of 7 in our example case which is equivalent to 0,0 .
- The mod operation is performed on the number and even larger number can be divided and can allocated in positions between 0 to n-1 .

```
def deQueue(self)->'bool':
    ## YOU CANNOT CALL pop(0). NOTE: pop(0) is O(n). We want THETA(1)
    ##
    if (self.isEmpty()):
        return False #means failure
    self._front = self._front + 1
    return True #means success
```

- Dequeue routine

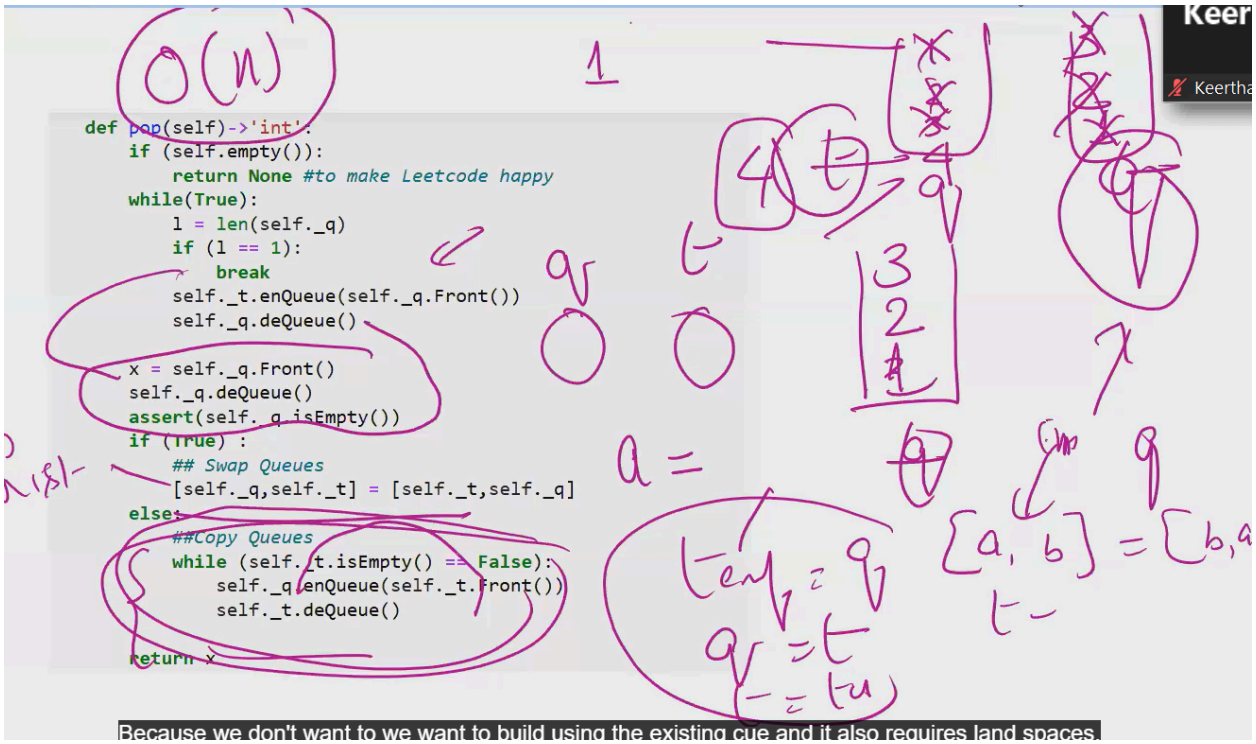
```
def Front(self)->'T':
    ## YOU CANNOT CALL pop(0). NOTE: pop(0) is O(n). We want THETA(1)
    if (self.isEmpty()):
        return -1 #means failure(because of Leetcode)
    #Compute real position
    p = (self._front) % self._MAX
    return self._a[p] #means success

def Rear(self)->'T':
    ## YOU CANNOT CALL pop here
    if (self.isEmpty()):
        return -1 #means failure(because of Leetcode)
    #Compute real position
    p = (self._back-1) % self._MAX
    return self._a[p] #means success
```

- Finding out rear value routine involves finding taking self_back and 1 value lesser as that's the exact location of the value.

```
#Our StackUsingQueue is finite size n
#############################################################
class StackUsingQueue():
    def __init__(self, n:'size'=10):
        # ONLY DATA STRUCTURE YOU CAN USE HERE IS ONLY QUEUE THAY YOU WROTE
        self._q = Queue(n)
        self._t = Queue(n)

    def push(self, T):
        self._q.enQueue(T)
```

- The main difference in stack and queue is FIFO and stack is LIFO so for queue to behave like stack we The push routine remains the same as it enqueue the values accordingly but pop a little complicated .

- With help of a temporary queue (_t) , we add in all values until the last one reaches and further convert the temporary queue to the original one using the below image logic.



```
    def top(self)->'int':
        if (self.empty()):
            return None
        return self._q.Rear()

    def peek(self)->'int':
        return self.top()

    def empty(self)->'Bool':
        return (self._q.isEmpty())

    def is_full(self)->'Bool':
        return False #our stack can never be full

    def space(self)->'int':
        return (len(self._q) + len(self._t))

    def __len__(self):
        return len(self._q) + len(self._t)
```

- Other routines involved in stack using queue.

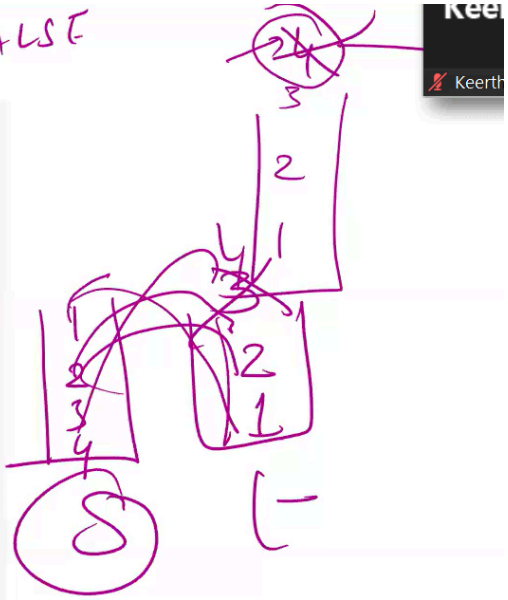
```
class QueueUsingStack():
    def __init__(self):
        # ONLY DATA STRUCTURE YOU CAN USE HERE IS ONLY STACK THAT YOU WROTE
        self._s = Stack()
        self._t = Stack()

    def push(self, T):
        self._s.push(T)
```

- Now similarly since the last element to pop is different from queue and stack we are also using a temporary stack with copying all the values . The pop operations tensed to be complex as first it checks whether if its pop or peek operation and does the following operation accordingly

11

Sunday, February 9, 2025 9:46 AM

```
def pop_or_peek(self,delete:'bool')->'T':
    if (self.empty()):
        return None #to make Leetcode happy
    if (len(self._t)):
        if (delete):
            return (self._t.pop())
        else:
            return (self._t.top())
    #That means temp is empty
    while((self._s.empty()) == False):
        self._t.push(self._s.pop())
    if (delete):
        return (self._t.pop())
    else:
        return (self._t.top())

def pop(self)->'T':
    return self.pop_or_peek(True)

def top(self)->'int':
    return self.pop_or_peek(False)
```



```
def top(self)->'int':
    return self.pop_or_peek(False)

def peek(self)->'int':
    return self.top()

def empty(self)->'Bool':
    l = len(self)
    if (l == 0):
        return True
    return False

def is_full(self)->'Bool':
    return False #our Queue can never be full

def space(self)->'int':
    return (len(self._s) + len(self._t))

def __len__(self):
    return len(self._s) + len(self._t)
```

Other routines in Queue using stack

Simialrly for finding the min stack we take another stack . Every time an element is inserted and popped , it makes difference in both s and t but t holds the minimum value . so top of temporary

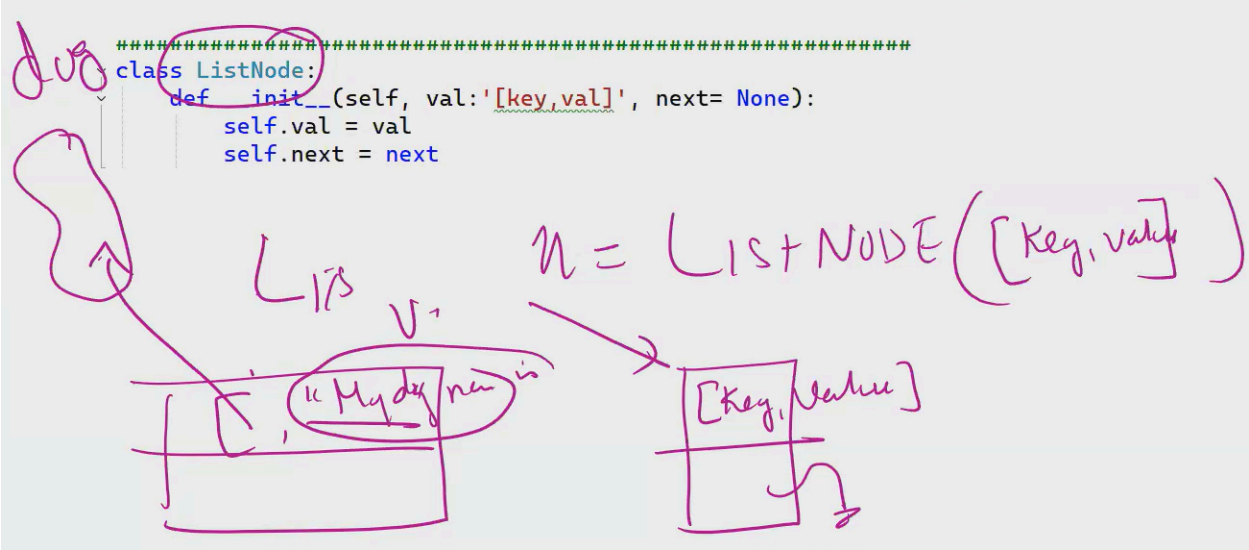
is used to find out the minimum value

```
##################################################################
class MinStack():
    def __init__(self):
        # ONLY DATA STRUCTURE YOU CAN USE HERE IS ONLY STACK THAT YOU WROTE
        self._s = Stack()
        self._t = Stack()

    def push(self, T):
        if (len(self._t) == 0):
            self._t.push(T)
        elif (T <= self._t.top()):
            self._t.push(T)
        self._s.push(T)

    def pop(self):
        l = len(self._s)
        if (l == 0):
            return None #to make Leetcode happy
        if (self._s.top() == self._t.top()):
            self._t.pop()
        self._s.pop()
```

```
def top(self)->'T':
    l = len(self._s)
    if (l == 0):
        return None #to make Leetcode happy
    return self._s.top()

def getMin(self)->'T':
    l = len(self._s)
    if (l == 0):
        return None #to make Leetcode happy
    return self._t.top()
```

Solution to HW4

List Node takes a list of key and value of dog object



- In the append routine of SList , It checks the key value and accordingly sets the val or builds the node upon it .

```
##############################
# Append: add a node at the end of a slist
# Time:   THETA(1)
# Space:  THETA(1)
##############################
def append(self,x:'[key,val]')->'bool':
    a = self._find(x[0])
    if (a[0]):
      a[0].val = x
      return False
    else:
      self._build_a_node(x,True)
      return True
```

(method) def _find(x: Any) -> list

Build a node routine checks if its an empty list node or already has existing values and allocates first and last accordingly.

```
##############################
def _build_a_node(self,i:'T',append:'bool' = True):
    n = ListNode(i)
    if (self._first == None and self._last == None ): #Handle empty case
        self._first = n
        self._last = n
    else:
        if (append):
            self._last.next = n
            self._last = n
        else:
            n.next = self._first
            self._first = n
```

Prepend routine : Assigns the the first element to the specified key , value pair.

```
def prepend(self,x:'[key,val]')->'bool':
    a = self._find(x[0])
    if (a[0]):
      a[1] = x[1]
      return False
    else:
      self._build_a_node(x,False)
      return True
```

- In Find routine in Slist , It firstly cells the helper find function . It looks out for the T value in Slist and once it finds out the it gives a list of prev and curr node

```
    ##############################
    # Find an element in slist
    # Time:   THETA(n)
    # Space:  THETA(1)
    ##############################
    def find(self,x:'key')->'[key,val]':
        nodes = self._find(x)
        if (nodes[0]):
            return [nodes[0], nodes[1]]
        else:
            return [None,None]
```

```
##############################
# Find an element in slist
# Time:   THETA(n)
# Space:  THETA(1)
##############################
def _find(self,x:'T')->'list of [currentnode,prevnode]':
    nodes = [self._first,None]
    while(nodes[0]
        val = nodes[0].val
        if (val[0] == x):
            return nodes
        nodes[1] = nodes[0]
        nodes[0] = nodes[0].next;
    return nodes
```

(variable) _first: Any | ListNode | None

Delete routine

```
def delete(self,x:'T')->'bool':
    nodes = self._find(x)
    if (nodes[0]):
        currentnode = nodes[0]
        previousnode = nodes[1]
        if ( (currentnode == self._first) and (currentnode == self._last) ):
            ## list has only one element
            self._first = None
            self._last  = None
        elif (currentnode == self._first):
            ## first element being removed and list has more than 1 element
            self._first = currentnode.next
        elif (currentnode == self._last):
            ## last element being removed and list has more than 1 element
            previousnode.next = None
            self._last = previousnode
        else:
            ## You are removing middle element
            previousnode.next = currentnode.next
        return True
```

Now coming to Dog Class function , Hash routine

```
##############################
# Must write hash
# if you don't write __hash__ you get Unhashable type Int
##############################
def __hash__(self)->'int':
  l = self._get_key()
  #note our key is Python list of [self._a, self._breed]
  h = 0
  for e in l:
      h ^= hash(e)
  return h
```

For all relational operators , we have declared __lt__operator , and other relational operators can be operated from this .

```
##############################
# Overload
# ONLY ONE ROUTINE TO WRITE
##############################
def __lt__(self, b: "Dog") -> "bool":
    #return (self._age, self._name, self._breed) < (b._age, b._name, b._breed)
    a = self
    if (a._age < b._age):
      return True
    if (a._name < b._name):
      return True
    if (a._breed < b._breed):
        return True
    return False
```

Dict Class

```
############################################################
#  class  Dict
############################################################
class Dict():
    def __init__(self, size:'int'):
        #NOTHING CAN BE CHANGED HERE
        self._table_size = size
        self._size = 0 # real number of elements in dict
        self._max_size = 0 # max number of elements in dict in the whole life spar
        self._list = []
        for i in range(self._table_size):
            x = self._list.append(Slist())
```

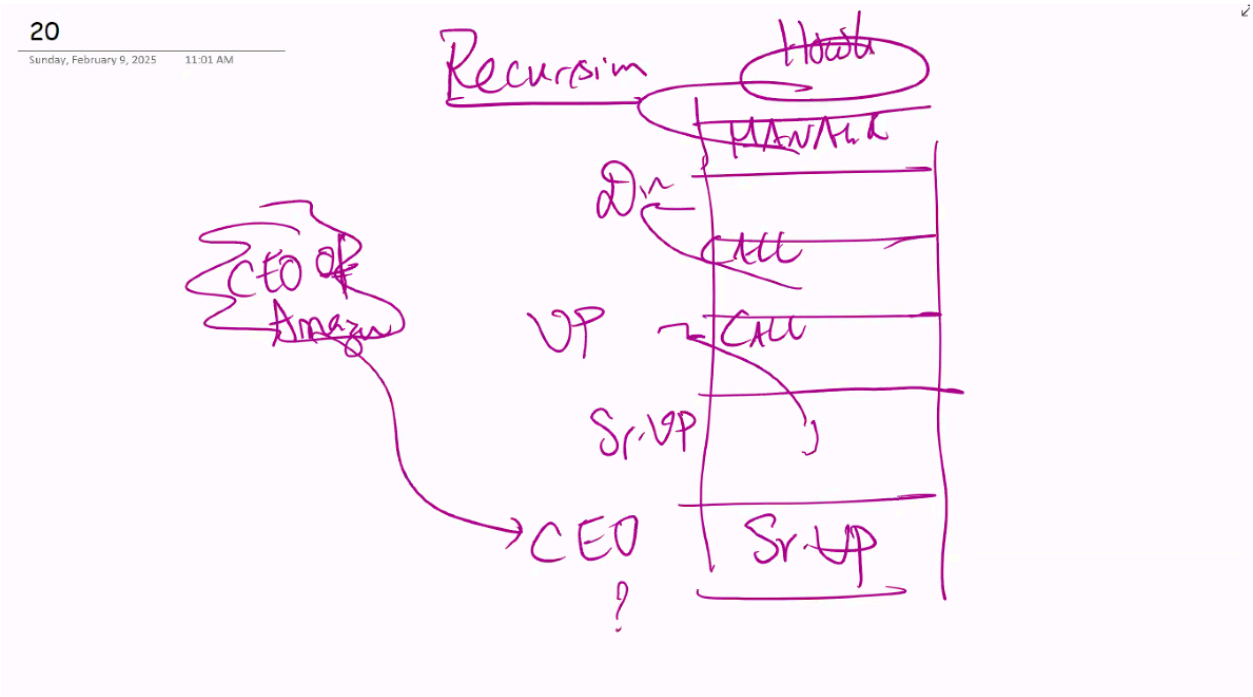
Max_size and Table_size are introduced to get an idea of what actually capacity is after multiple insert remove operations and accordingly add and remove them .

```
##############################
# insert: add key and value
# Time:   THETA(1)
# Space:  THETA(1)
##############################
def insert(self,key:'T',value:'V'):
    x = self._hash_func(key);
    b = self._list[x].append([key,value])
    if (b):
      self._increment_size()
    if (self._max_size < self._size):
      self._max_size = self._size
```

Insert routine : computes the hash , appends the value and modifies the max_list

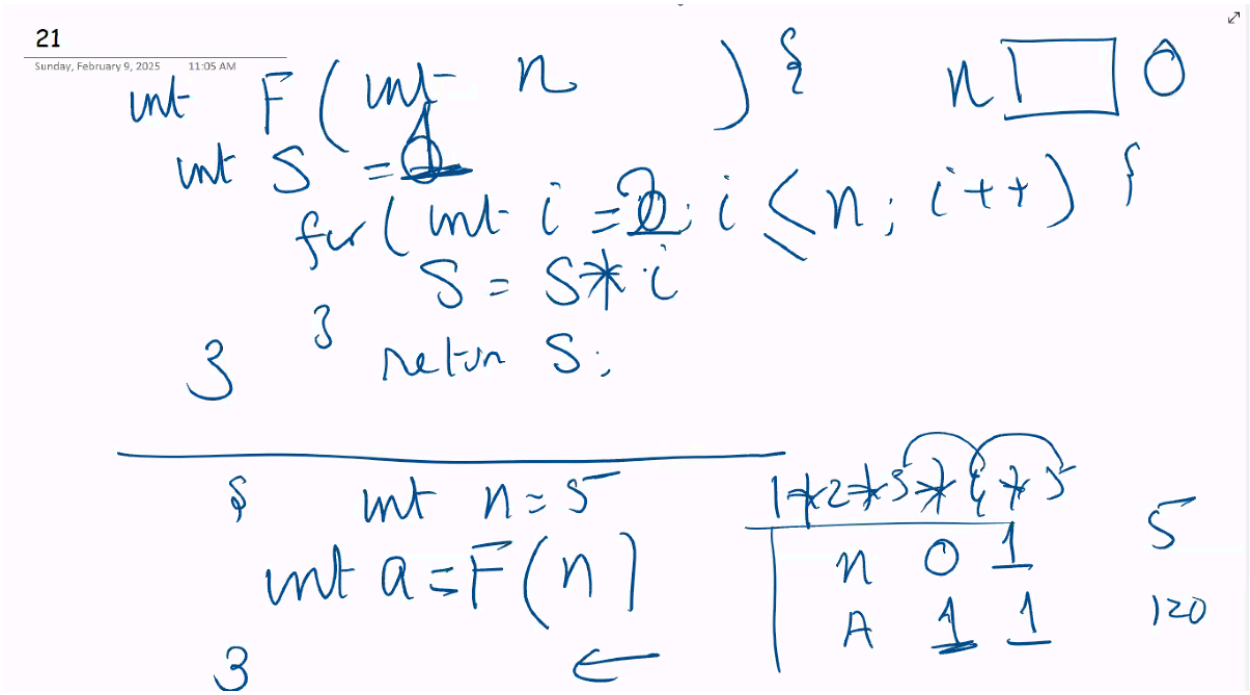
Students should Transcribe from this point

Recursion

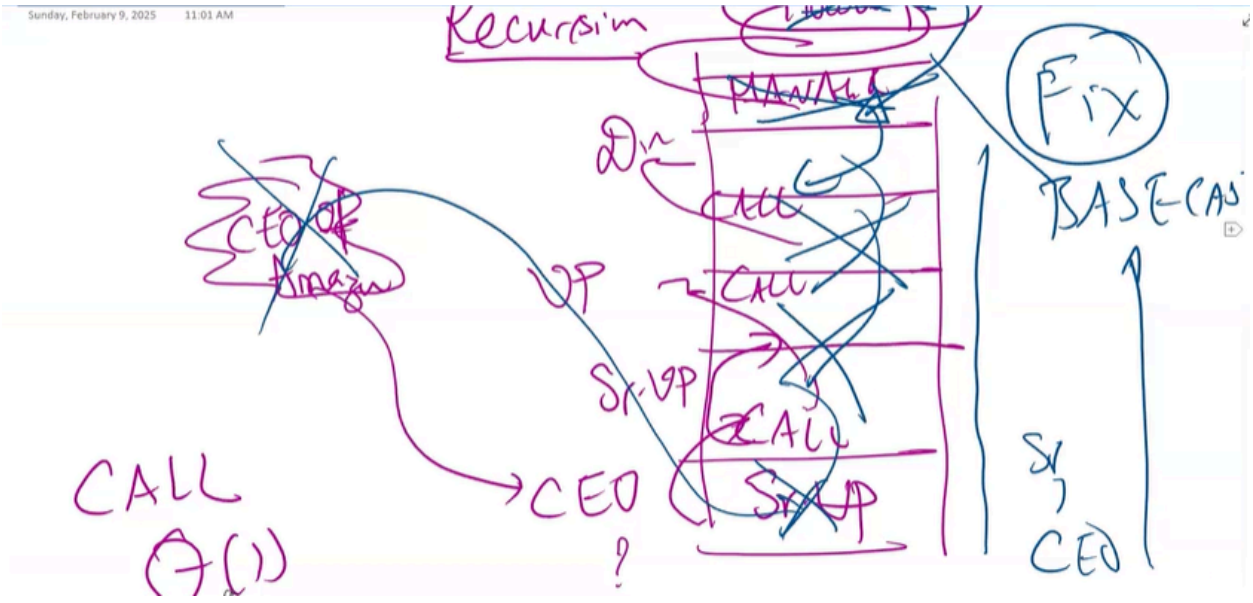


- When the CEO of Amazon identifies an issue with a product, it is escalated to the Senior Vice President, then to the Director, and subsequently to an Engineer—the individual with the expertise to resolve the bug (the base case). Once the issue is fixed, the Engineer reports the resolution to the Manager, who then informs the Senior Vice President. This communication continues recursively up the hierarchy, ensuring that each level is updated until the CEO receives confirmation that the bug has been successfully resolved.

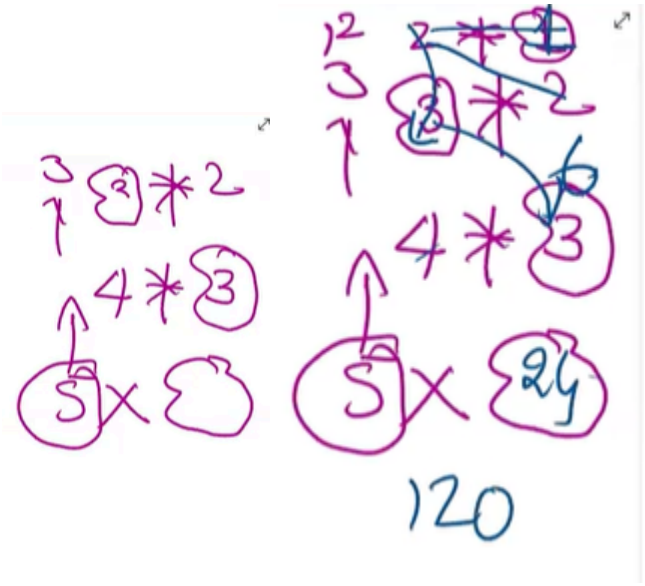
Factorial example



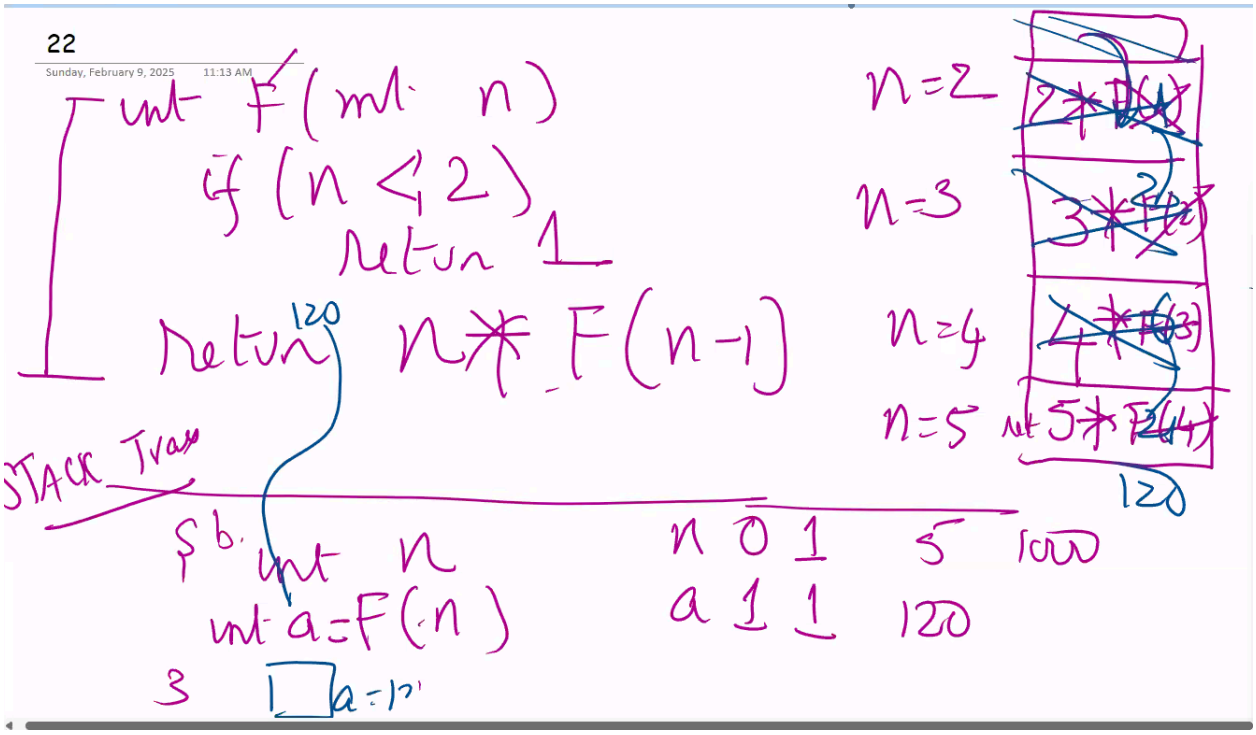
- The above method talks about iteratively adding and below method recursively . The F function is finding out factorial of a number . In recursive way , F is already defined and let's take the example of 5 the f will return 1*2*3*4*5 = 120 . In iterative way , Since the for loop sterts from 2 , any number reentered before that let's say 0 or 1 the factorial is the respective number and hence it returns that same number . Three spaces i.e n,i and s have been utilised . Now until the i<=n , the loop goes and in s , every number is been multiplied until the the loop condition. The time complexity is theta(n) in both cases but in terms of space complexity, it is theta 1 in iterative and recursive it is theta n



- This works in python and not in other languages because there is size limitation in java andc++ whereas a large number int is possible to define in Python.
- In the previous example of recursion , The calling routine is thet(1) and every person knows how to call the other person and person who knows how to resolve resolves it .
- The crux concept of this problem is that one needs to know how to break this problem and one person needs to know how to resolve the problem.

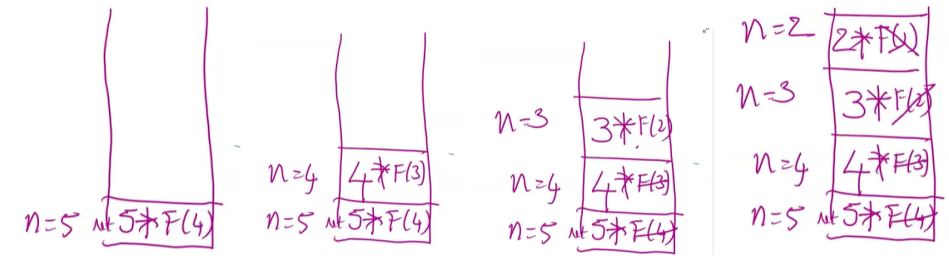


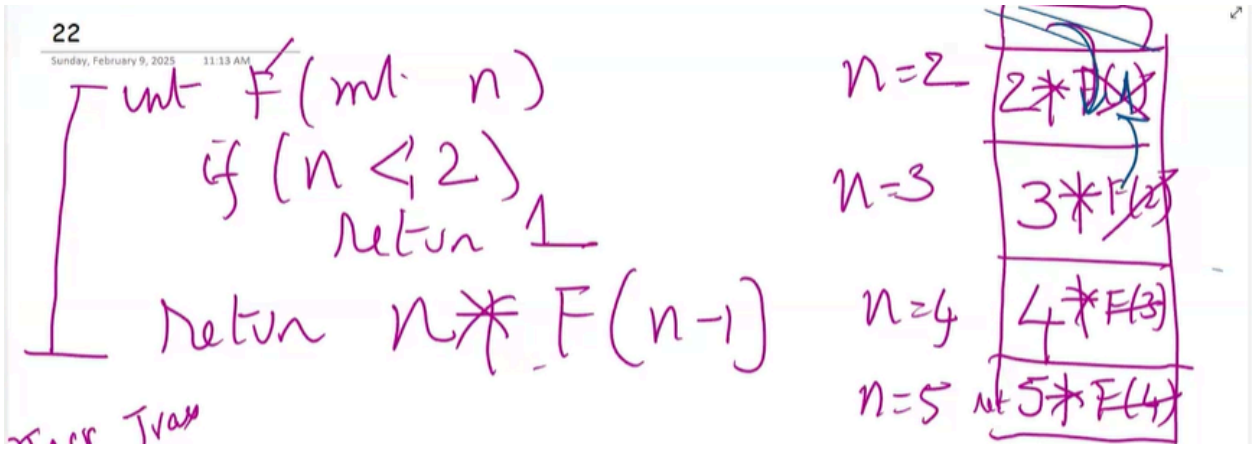
- Similarly for this problem, F calls the F of the previous value and it goes on until the base case. The base case returns with an answer and the further cases recursively tke in previous answer and compute their own answer .



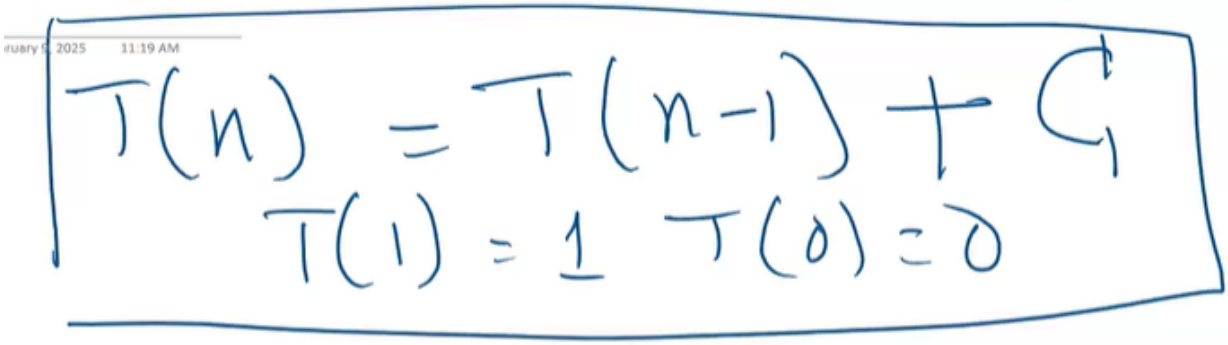
- In this case , manager's calling activity is multiplication and F(n-1) the recursive contact person . To determine the whole function of this , we use a stack trace .

Working of a Stack Trace

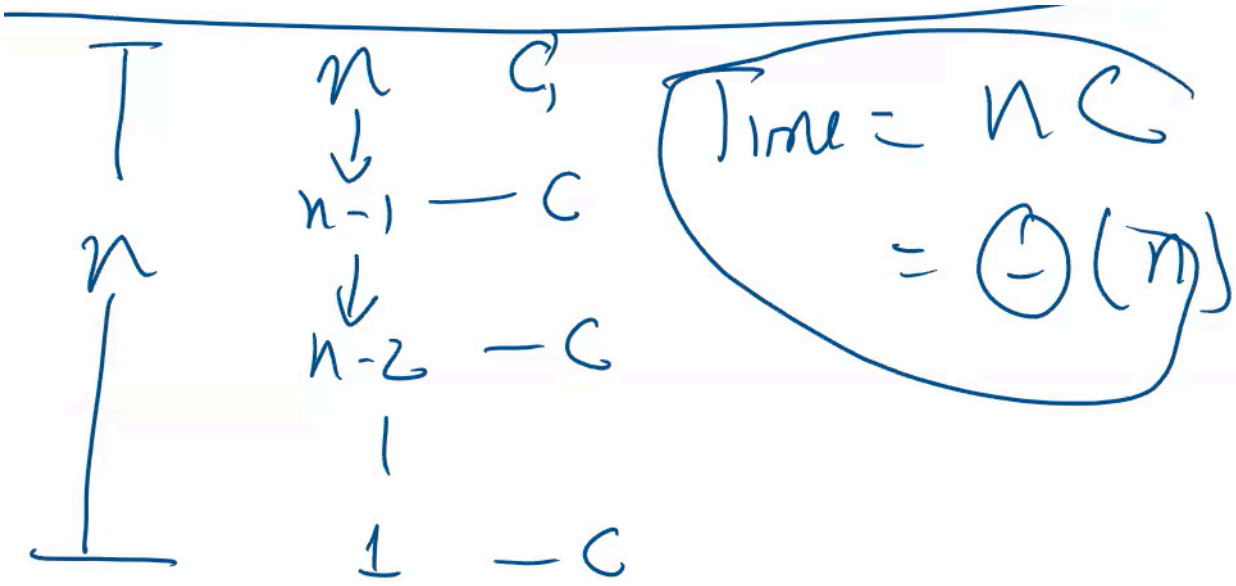




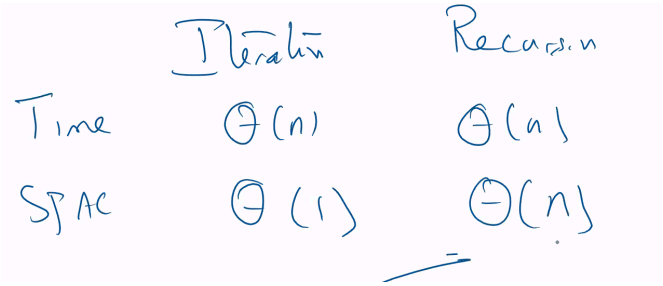
- The compiler brings in the stack trace whose availability is just within the function . The whole fuicntion of n is just limited within the function and beyond that stack trace ain't available.
- In the first case , it takes n value as 5 and every return statement is stored in stack . For example take 5 , for n =5 the return statement is stored in stack . For n = 4 frame , It calls for F(3) and hence it recurisvely goes on for further number until it finds the base case . In this case n= 1 is the base case and according to the if condition it gives return as 1 . After the returned value as 1 it goes to below frame and returned answer for every frame goes down .

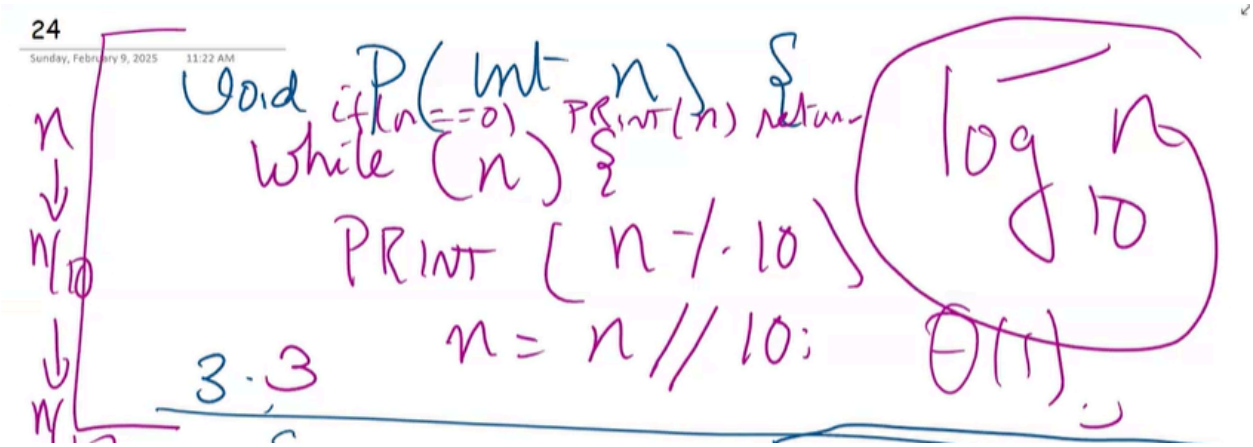


- Recurrence formula means that we did call T(N-1) and constant operations to call the recursive members . in our case , C is constant multiplication and has base cases of T(1) and T(0) .
-
- The C which is constant c operation s is called n times until n =1 and hence the total time complexity os Theta(n)

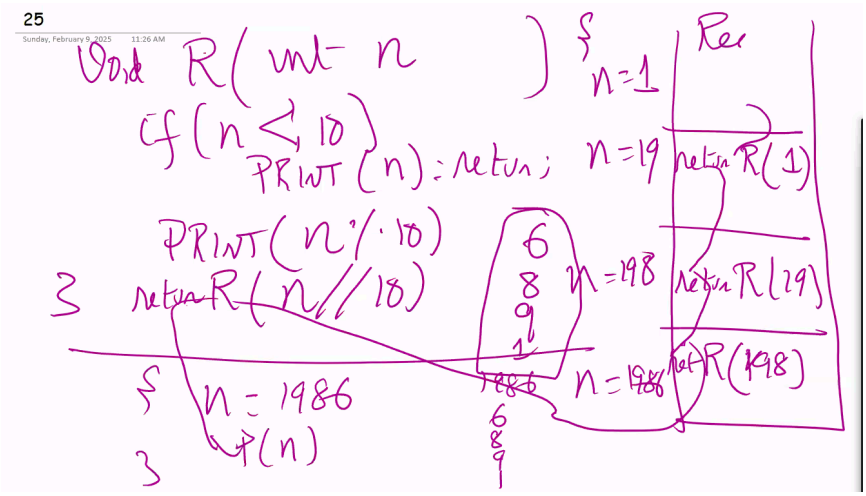


- The space and time complexity remains same for both cases but space is Theta(N) in Recursion and Theta(1) in iterative.



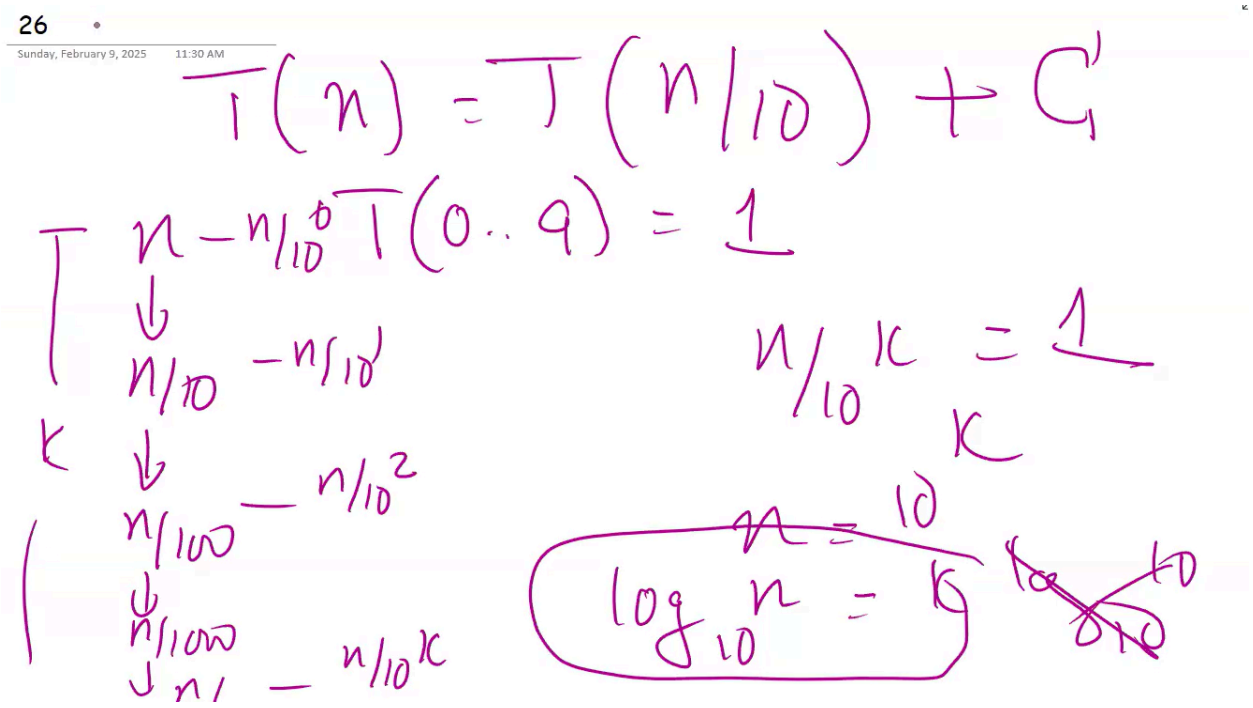


- Iteratively, The whole function prints numbers reversely and continues until the base case . The overall time complexity is lon n to base 10 and space is theta(1)

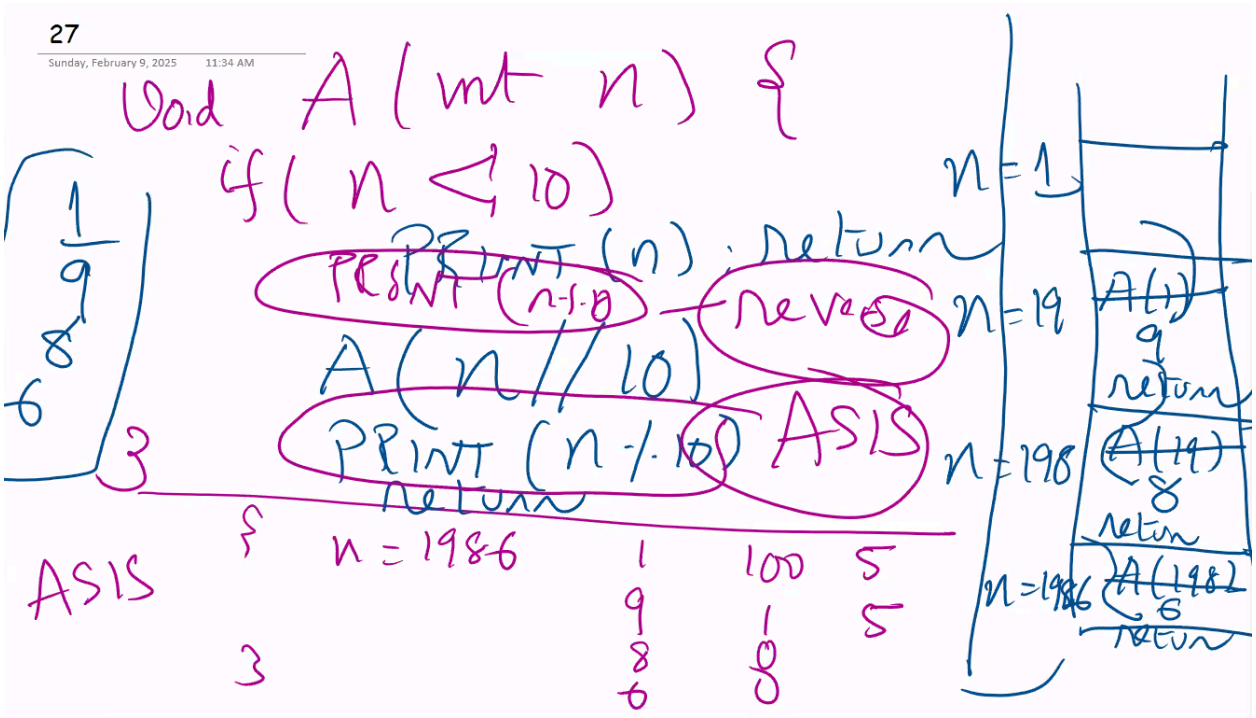


- In this case we are printing digits of a number, the time and space complexity has gone down to logn . Let's take example of 1986 . it firstt takes R(198) as return statement to be present in that that value of N and recursively calles the previous frames until it

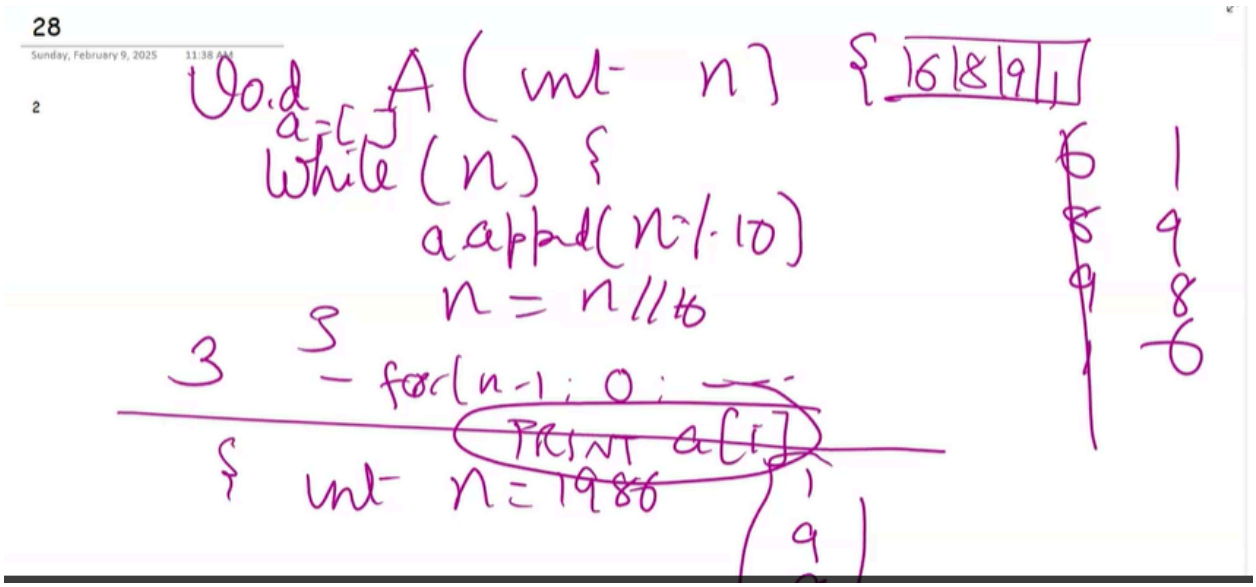
reaches the base case that n = 1 which according to the loop condition returns n = 1 and then recursively from the base case all the below n frames gets the previous value answer and computes their own answer and that's how print work. .Since the space complexity is log(n) , in this case , we can use recursion but in terms of factorial we cannot because theta(N) complexity .



- With respect to the formula , We are performing (K) Constant operation on n/10 values until it reaches it base case . And recursion can be done if its log(N) complexities as the space takes let's e say for 1 million will be just 20 spaces .

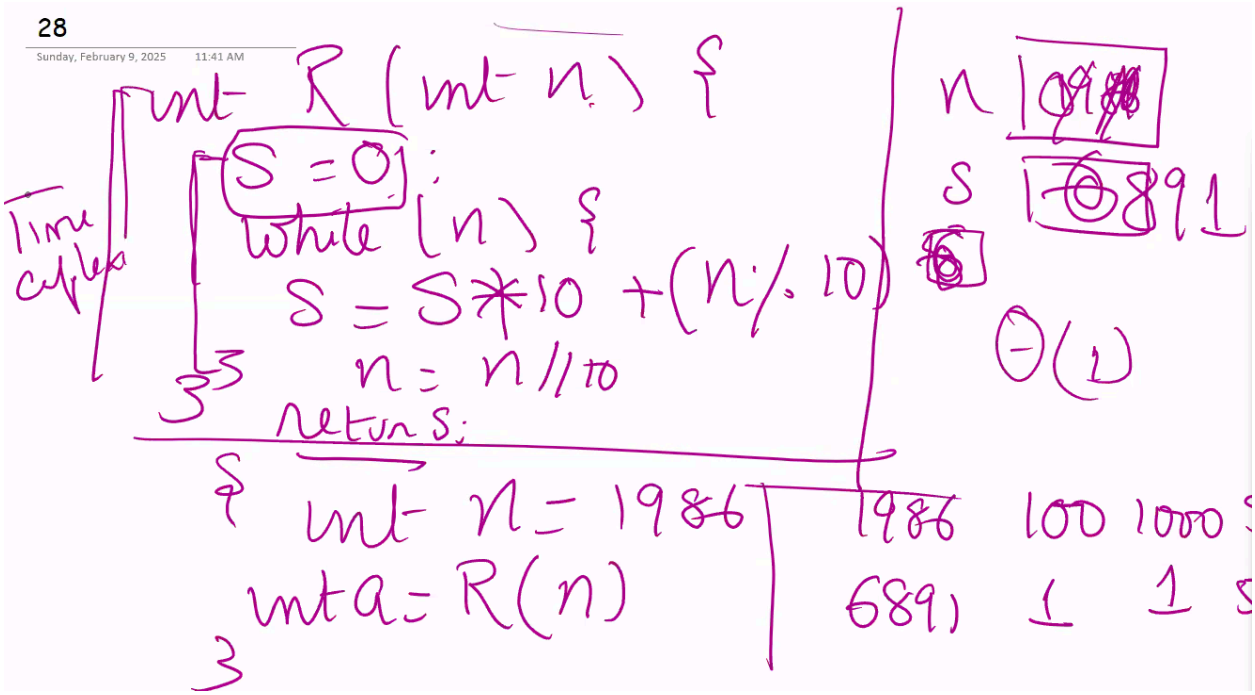


- If print(n%10) is used before the recursive function of A(n//10) , then it gives the reverse printing style of the number as the function A(n//10) is executed whereas if i have the print function after the recursive function then i will get the number as it is as it goes until the base case and from base case it prints values so firstly it will will print 1 then 9 8 and finally 6 .

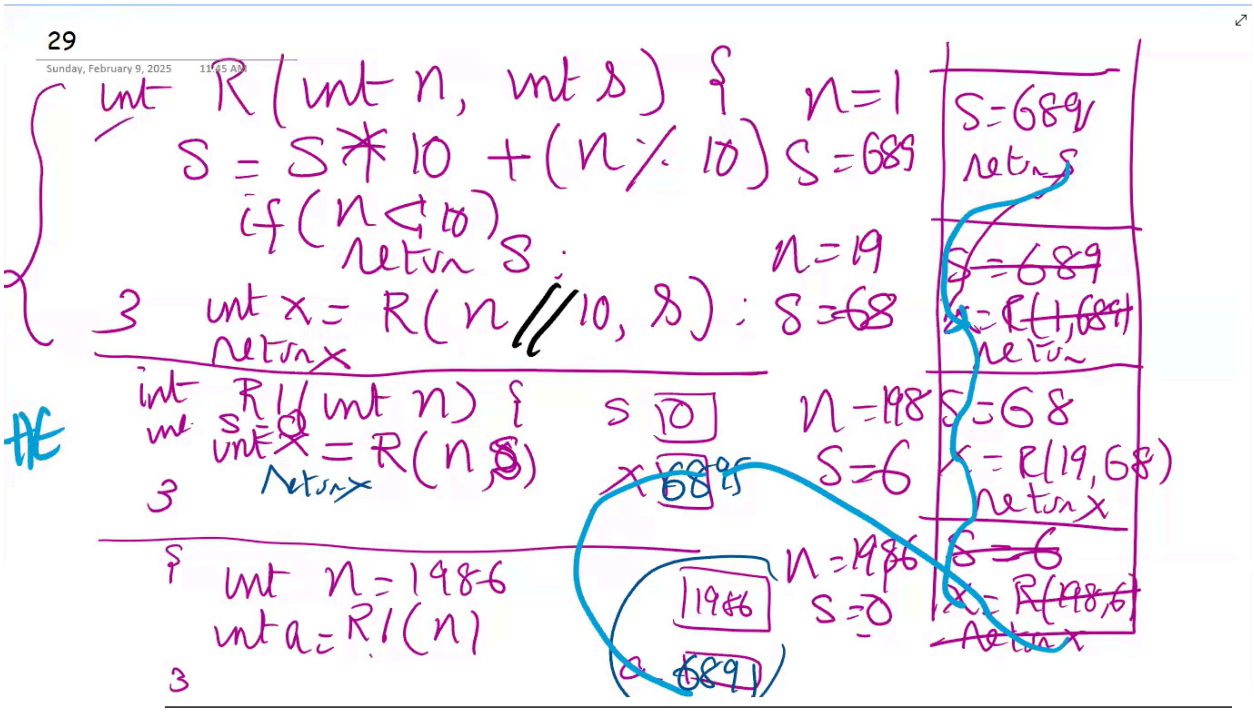


- If i have to iteratively perform the same operation , I need to include a list to append all values and later a for loop to get all values and print as it is.

Need for helper function in Recursion

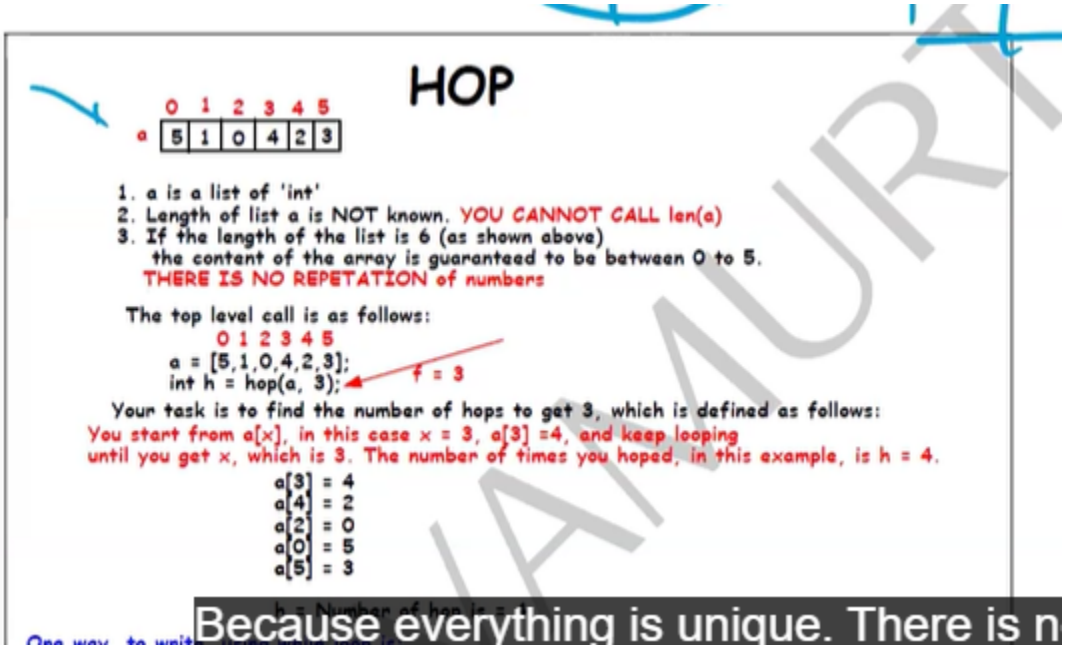


- In this code snippet , n holds the value and s holds the vlue of number . For every n//10 values , we arecomputing s and evemtuqally the whole number if stored in s and returns iut . s is being memorized with the output for every loop and time complexity goes to logn and space to theta(1). Now let's consider for recursion



- In this case , The helper function R1 is used and recursively calls R which in turn stores two values that is n which goes n//10 th values and s which holds the combined number at that instance in every frame of stack trace. This continues until it reaches the base case (n < 10 which is 1) and recursively returns the s value from the topmost frame to the downmost and finally gives out 6895 values from R which insturn is stored in R1 and anf finally to the main function which inturn gives the whole value

HW-5 HOP



For this hw , we need to find the number of hops in order to find value untill a[i] = i (example a[3] = 3) .

h = Number of hop is = 4

One way, to write, using while loop is:

```
def _hop_easy(self, a:List[int],f:'int') -> 'int':
    ## n = len(a)
    ##YOU CANNOT CALL len
    t = f
    h = 0
    while (True):
      if (a[t] == f):
        return h
      else:
        t = a[t]
        h = h + 1
    return h
```

Now write "hop" subroutine as follows:

```
def _hop(self, a:List[int],f:'int') -> 'int':
```

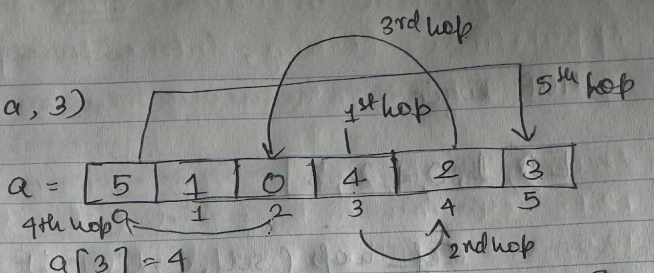
0. Content of list a should be exactly same after executing procedure hop
1. You cannot change interface of hop function
2. You cannot use global/static variables
3. You cannot use any loop statements like while, do, for and goto
4. You cannot call any function except _hop
5. Your code should not be more than 10 lines

Now with the below set of instructions need to write code for _hop recursively

Stack Trace Assign

For

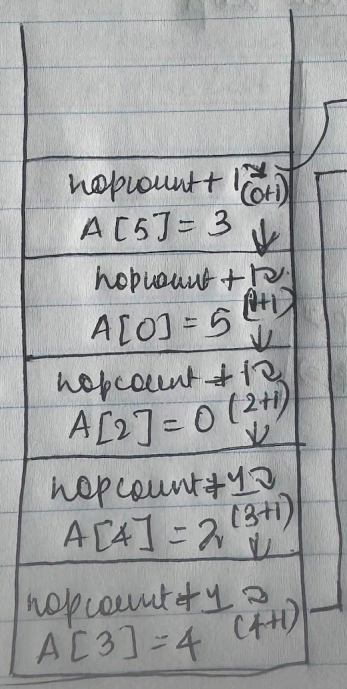
- hop(a, 3)



a[3] = 4
a[4] = 2
a[2] = 0
a[0] = 5
a[5] = 3

After 5 hops, a[3] = 3

Stack Trace (Basic Understanding)



→ This is the base case where a[i] = i so hop = 0

→ this hopcount value will be the result (5)