



IIT Madras

ONLINE DEGREE

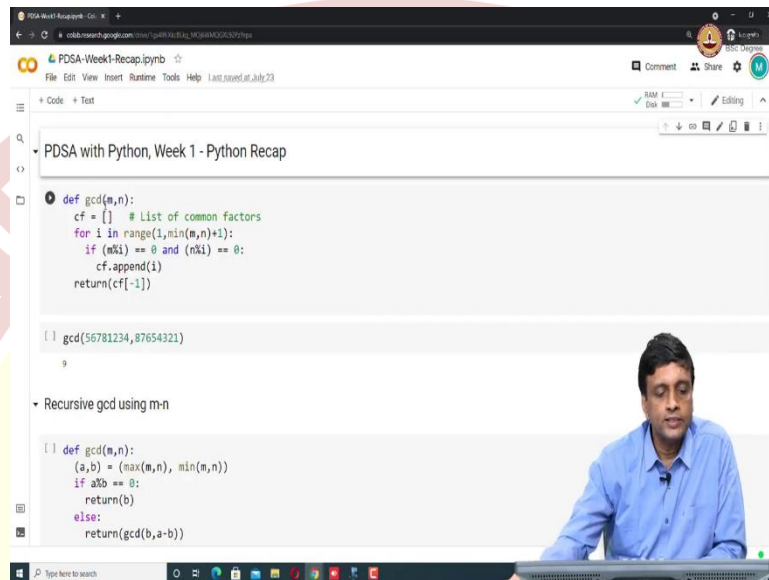
Programming, Data Structures and Algorithm using Python

Professor Madhavan Mukund

Implementation of Python Codes-Part 4

So, since we cannot construct this list of Aadhaar cards and SIM cards, let us look at this efficiency in our running situation with respect to our GCD algorithms.

(Refer Slide Time: 00:22)



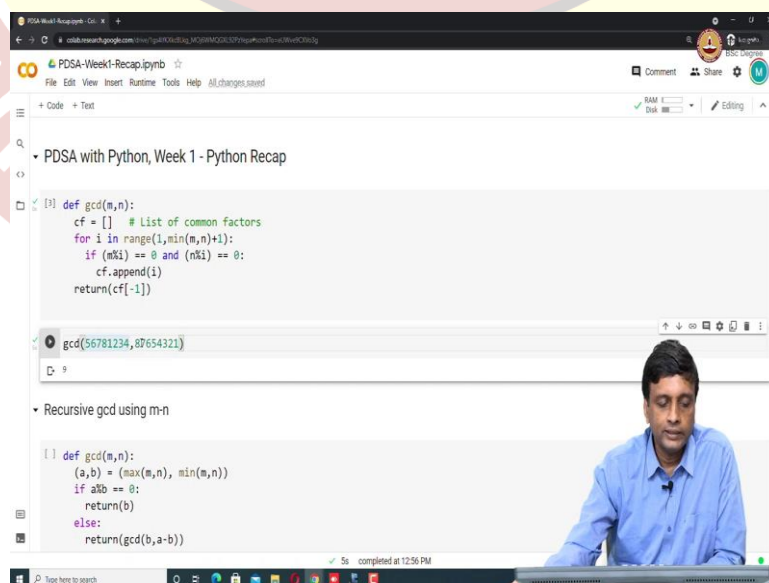
The screenshot shows a Jupyter Notebook titled "PDSA-Week1-Recap.ipynb" with the following code:

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])  
  
gcd(56781234,87654321)  
  
9
```

Below the code, there is a video feed of Professor Madhavan Mukund.

So, this is our original GCD algorithm. The one which computed the list of common factors, and reported the last one. And what we observed was that this takes time proportional to the minimum of the 2 numbers because this for loop runs for that long.

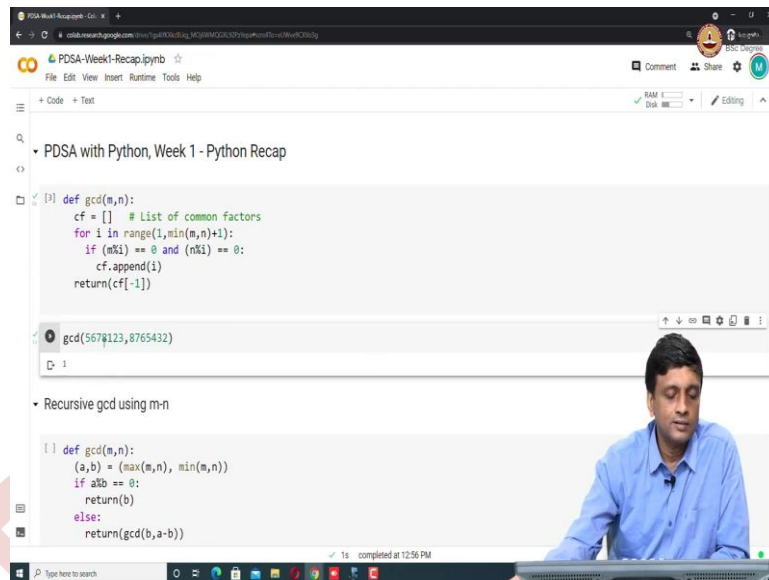
(Refer Slide Time: 00:40)



The screenshot shows the same Jupyter Notebook with the following code:

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])  
  
gcd(56781234,87654321)  
  
9
```

Below the code, there is a video feed of Professor Madhavan Mukund. At the bottom of the notebook, a status bar indicates "5s completed at 12:56 PM".

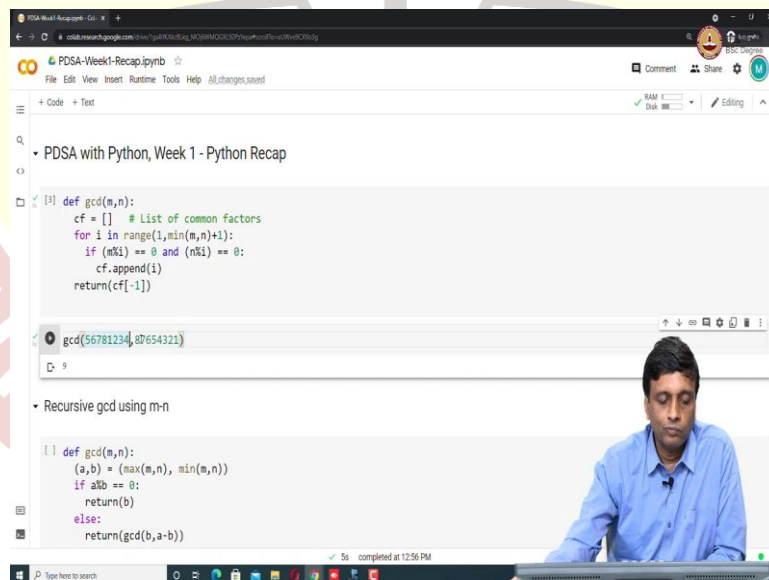


The image shows a Jupyter Notebook titled "PDSA-Week1-Recap.ipynb". It contains two code cells. The first cell defines a function `gcd(m,n)` that finds common factors. The second cell calls `gcd(5678123, 8765432)` and returns the value 1. A video inset shows a man in a blue shirt.

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])  
  
gcd(5678123, 8765432)  
1  
  
Recursive gcd using m-n  
  
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else:  
        return(gcd(b,a-b))
```

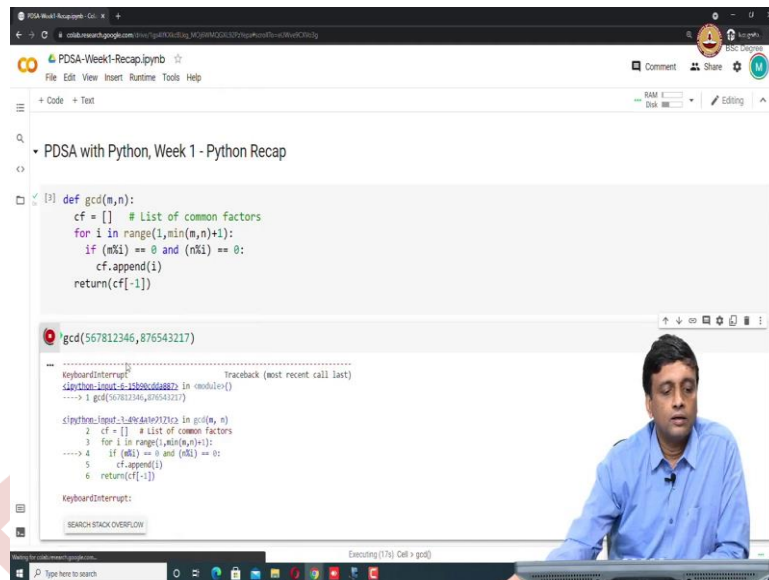
So, let us look at 2 GCD of 2 large numbers. So, let me take smaller numbers. So, let me take two 7-digit numbers. So, let me so this is now GCD of 5678123, which is a 7-digit number, and 8765423. So, let me first execute this so that this Python function is, so now GCD, has been defined. So, now if I run this GCD on this number, then it reports an answer, reasonably fast as 1. So, it actually says that this is, now if I suppose I increase the number of digits by 1.

(Refer Slide Time: 01:11)



The image shows the same Jupyter Notebook as above, but the second code cell now calls `gcd(56781234, 87654321)` and returns the value 9. The video inset shows the same man in a blue shirt.

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])  
  
gcd(56781234, 87654321)  
9  
  
Recursive gcd using m-n  
  
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else:  
        return(gcd(b,a-b))
```



```
PDSA-Week1-Recap.ipynb
File Edit View Insert Runtime Tools Help

+ Code + Text
RAM 1.0 GB
Disk 100 MB
Editing

PDSA with Python, Week 1 - Python Recap

[3]: def gcd(m,n):
      cf = [] # List of common factors
      for i in range(1,min(m,n)+1):
          if (m%i) == 0 and (n%i) == 0:
              cf.append(i)
      return(cf[:-1])

gcd(567812346,876543217)

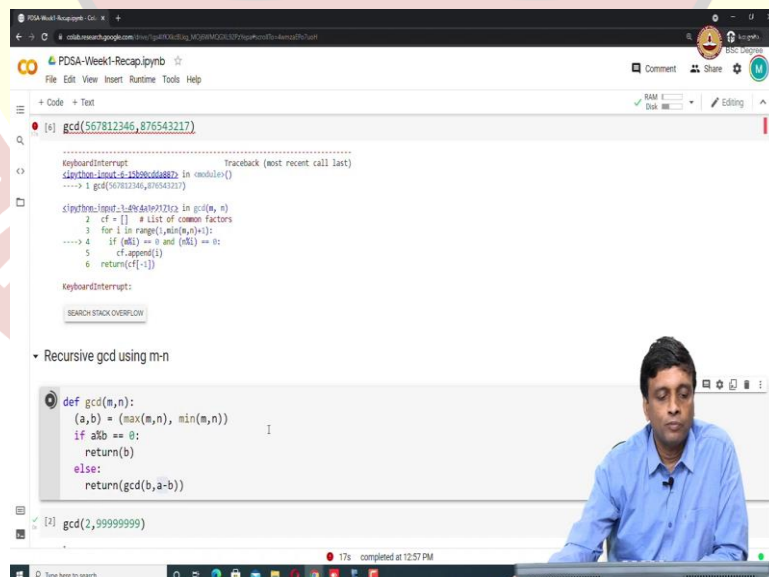
KeyboardInterrupt:
Traceback (most recent call last)
<ipython-input-6-19290c0a882> in module()
----> 1 gcd(567812346,876543217)

<ipython-input-1-444444444> in gcd(m, n)
      2 cf = [] # List of common factors
      3 for i in range(1,min(m,n)+1):
----> 4     if (m%i) == 0 and (n%i) == 0:
      5         cf.append(i)
      6 return(cf[:-1])

KeyboardInterrupt:
SEARCH STACK OVERFLOW
```

So, supposing I make this two 8-digit numbers. If I make these 2-into-8-digit numbers, and then I run this GCD, then you can see that it is really taking a long time, because now the minimum of 2 numbers is now an 8-digit number, and if I take yet another digit, so supposing I make them 9-digit numbers, right, then actually, if I start running this, this will run for an enormous amount of time, I mean, so I can just keep talking and nothing is going to happen. So, you can see that having a kind of naive algorithm will seriously limit the efficacy of what you are doing.

(Refer Slide Time: 01:51)



```
PDSA-Week1-Recap.ipynb
File Edit View Insert Runtime Tools Help

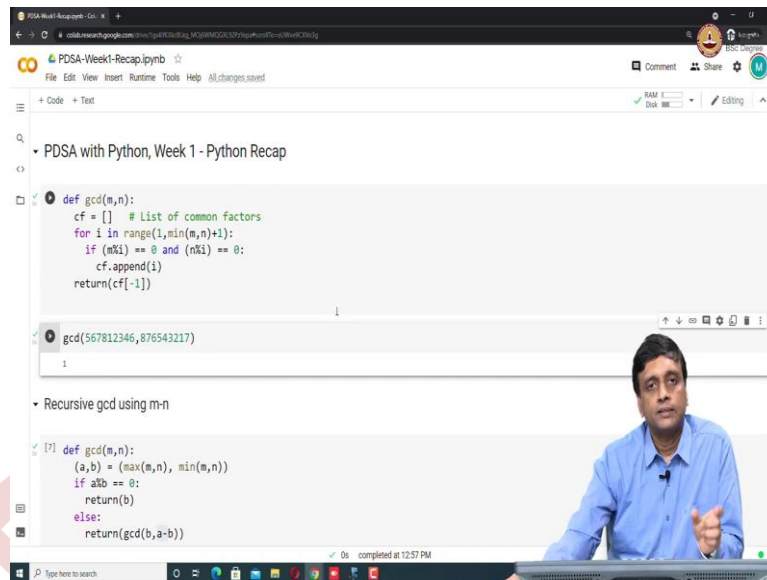
+ Code + Text
RAM 1.0 GB
Disk 100 MB
Editing

Recursive gcd using m-n

def gcd(m,n):
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0:
        return(b)
    else:
        return(gcd(b,a-b))

gcd(2,999999999)

17% completed at 12:57 PM
```



The screenshot shows a Jupyter Notebook titled "PDSA-Week1-Recap.ipynb". The code defines a recursive function `gcd(m,n)` that finds common factors. The execution cell shows `gcd(567812346, 876543217)` returning `1`. A video inset shows a man in a blue shirt speaking.

```
def gcd(m,n):
    cf = [] # List of common factors
    for i in range(1,min(m,n)+1):
        if (m%i) == 0 and (n%i) == 0:
            cf.append(i)
    return(cf[-1])

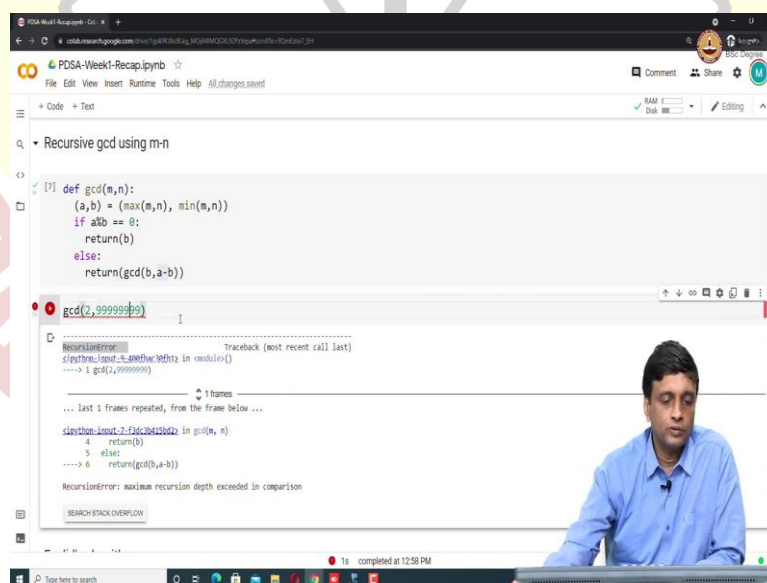
gcd(567812346, 876543217)
1
```

Recursive gcd using m-n

```
def gcd(m,n):
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0:
        return(b)
    else:
        return(gcd(b,a-b))
```

So, now, I just stopped that. So, now let me use instead, this first recursive algorithm that we defined, which uses this minus a minus b format. So, let me use this, a minus b format on this. And now let me go and try to re-evaluate this thing, which was taking a long time. So, I take this 9-digit number. And now if I do a minus b format, fairly fast, it gives me 1. So, switching to this recursive algorithm, gives me a 1. But we already saw that this a minus b format itself has problems because it will keep subtracting and coming down very slowly.

(Refer Slide Time: 02:27)



The screenshot shows the same Jupyter Notebook. The code for the recursive GCD function is the same. The execution cell shows `gcd(2, 999999999)` which has resulted in a `RecursionError: maximum recursion depth exceeded in comparison`. A traceback is visible, showing the function calling itself repeatedly. A video inset shows the same man in a blue shirt.

```
def gcd(m,n):
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0:
        return(b)
    else:
        return(gcd(b,a-b))

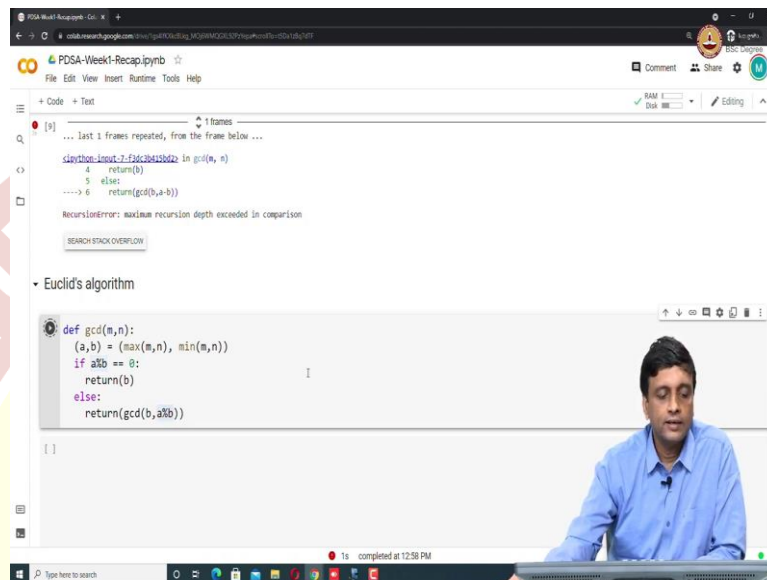
gcd(2, 999999999)
```

RecursionError: maximum recursion depth exceeded in comparison

So, supposing I take something like 2 and a large number of 9s. So, this will take a long time. But it will also explore some other thing about Python, which is that Python has something called a recursion limit. So, this is, every time you run a recursive function, it has to remember the previous thing.

So, actually what this gives us is something more serious, it gives us a recursion error. Now, it is possible to fix this recursion limit and make it larger. But in this particular case, if I want to make it large enough to do this, it actually still does not work. So, but even if I take a smaller thing, I mean, this recursion limit is going to be a problem.

(Refer Slide Time: 03:05)



```
[0] ... last 1 frames repeated, from the frame below ...
> cleython-ipynt-2-f34c041062a in gcd(m, n)
    4     return(b)
    5     else:
----> 6         return(gcd(b,a-b))

RecursionError: maximum recursion depth exceeded in comparison
```

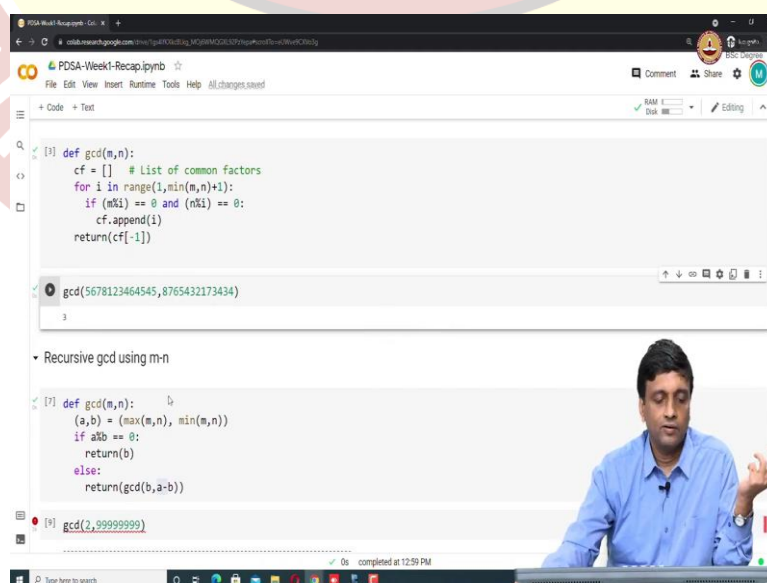
SEARCH STACK OVERFLOW

Euclid's algorithm

```
def gcd(m,n):
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0:
        return(b)
    else:
        return(gcd(b,a-b))
```

But now let us look at Euclid's algorithm. So, Euclid's algorithm was the same recursive thing except I used the remainder operation for subtracting, so I take in the base case, I check whether b divides a, if b divides a I return b, otherwise, I compute the GCD of b and a remainder b. So, let us now use this as our operational definition.

(Refer Slide Time: 03:30)



```
[3] def gcd(m,n):
    cf = [] # List of common factors
    for i in range(1,min(m,n)+1):
        if (m%i) == 0 and (n%i) == 0:
            cf.append(i)
    return(cf[-1])

gcd(5678123464545, 8765432173434)
3
```

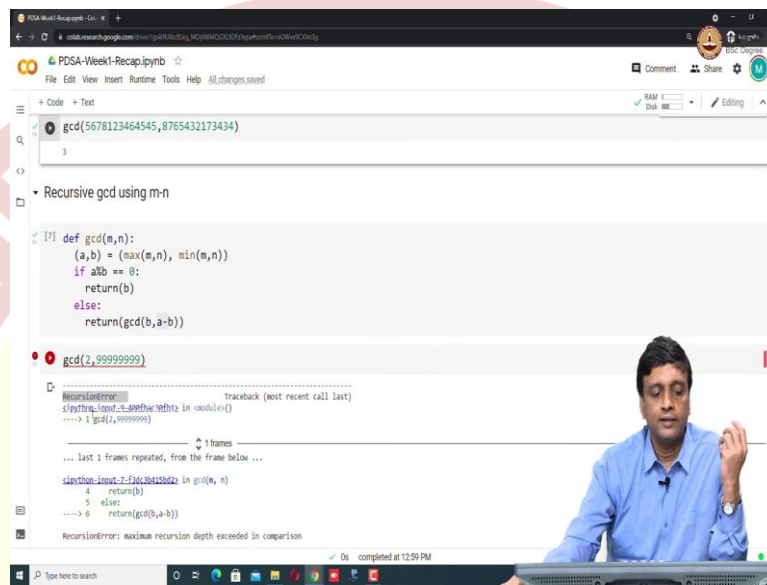
Recursive gcd using m-n

```
def gcd(m,n):
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0:
        return(b)
    else:
        return(gcd(b,a-b))

gcd(2,999999999)
```


So, first of all, we can go back and try the first one that our earlier thing failed to do. And you can see that it does is very fast. And I can probably add a lot more digits to it. Let me see, I hope it will work. So, you can see that our claim was that GCD, Euclid GCD actually grows proportional to number of digits. So, if I add 2 more digits, it grows only as plus 2, not time 10.

(Refer Slide Time: 03:55)



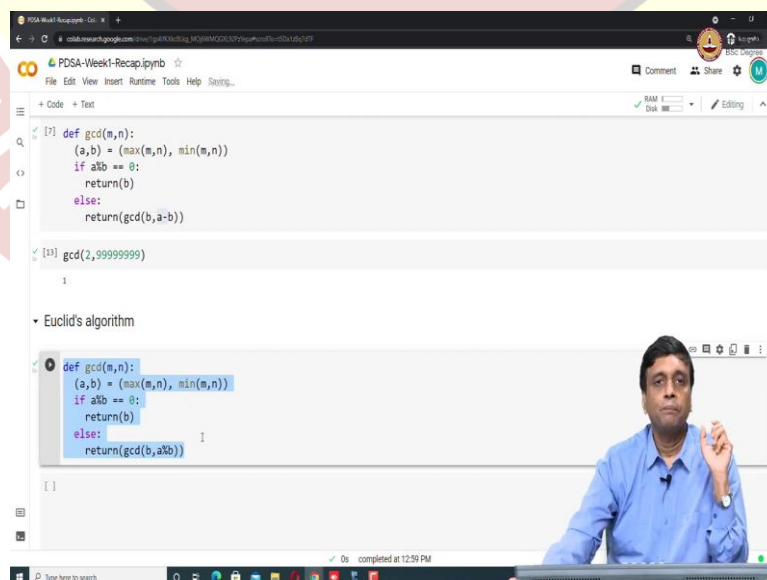
The screenshot shows a Jupyter Notebook interface with a file named 'PDSA-Week1-Recap.ipynb'. The code cell contains a recursive function for GCD:

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else:  
        return(gcd(b,a-b))
```

Below the code, the command `gcd(5678123464545, 8765432173434)` is executed successfully, returning 3. However, the next command `gcd(2, 999999999)` results in a `RecursionError: maximum recursion depth exceeded in comparison`. The error message includes a traceback and a note that the last 1 frames are repeated.

And in particular, if I do this one, which our earlier thing failed, because of the limit GCD, the Euclid GCD, very quickly tells me that the GCD of 2 and a large odd number is 1.

(Refer Slide Time: 04:05)



The screenshot shows the same Jupyter Notebook interface. The code cell now contains Euclid's algorithm for GCD:

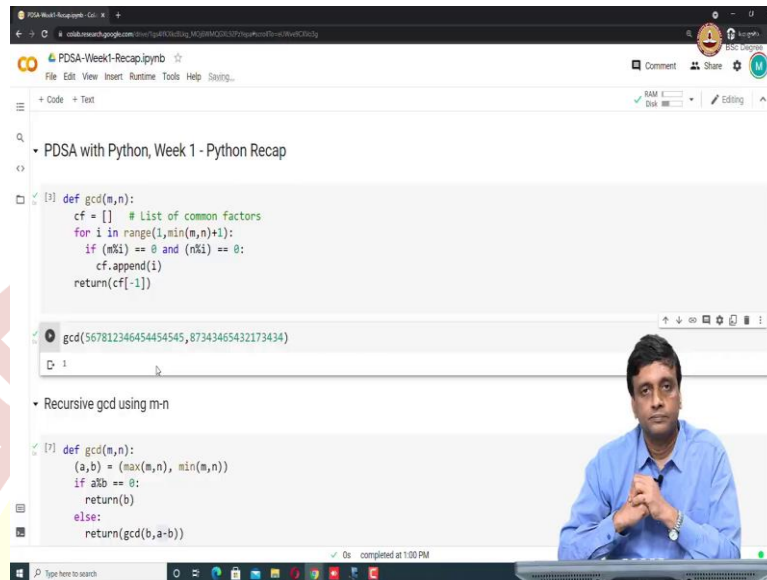
```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else:  
        return(gcd(b,a%b))
```

Below the code, the command `gcd(2, 999999999)` is executed successfully, returning 1. The notebook also shows the function definition for Euclid's algorithm.

So, this is the real power of Euclid's algorithm. So, it actually gives you a very efficient way to calculate GCD. And hopefully this gives you some feel for the dependencies on some

things which are proportional to the number value as a magnitude and the number of digits in that value. So, Euclid actually operates in number of digits.

(Refer Slide Time: 04:27)



The screenshot shows a Jupyter Notebook interface with the title 'PDSA-Week1-Recap.ipynb'. The code cell contains a function `gcd(m,n)` that finds common factors. Below the code, the output shows the result of `gcd(5678123464544545, 87343465432173434)` as `1`. A second code cell shows a recursive implementation of `gcd(m,n)`. A video inset in the bottom right corner shows a man in a blue shirt speaking.

```
[3]: def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])  
  
gcd(5678123464544545, 87343465432173434)  
1  
  
Recursive gcd using m-n  
  
[7]: def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else:  
        return(gcd(b,a-b))
```

So, if I give you a large number of this is basically operating on the length of the string. So, I can actually put much so Python, as you know, can take very large numbers. So, I can take much larger numbers and hopefully run this. And Euclid's algorithm works. So, it can actually handle very large numbers because it is actually it is like multiplying, I mean, multiplying two very large numbers, if I add few more digits, it just takes a little bit more time. Euclid's algorithm is like that. So, I hope you get a feel for this from this running example.