INDIAN INSTITUTE OF TECHNOLOGY MADRAS

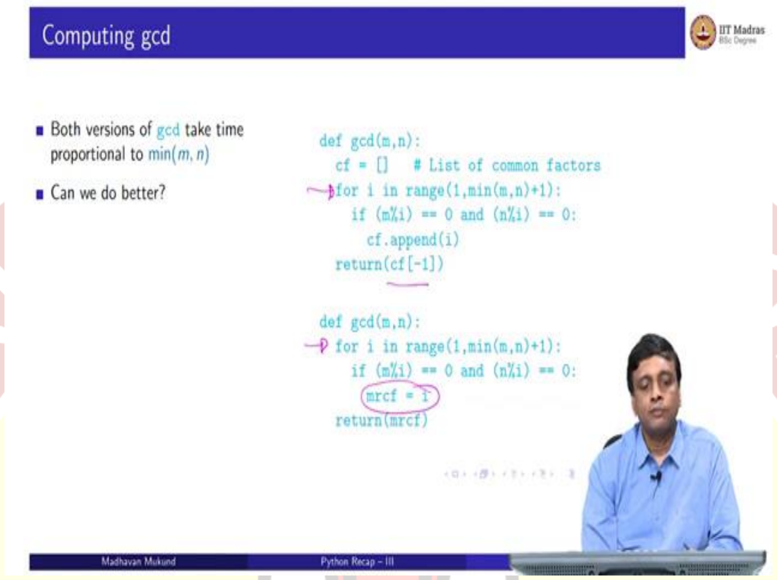
सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhavan Mukund
Python Recap - III

(Refer Slide Time: 0:20)



Computing gcd

- Both versions of gcd take time proportional to $\min(m, n)$
- Can we do better?

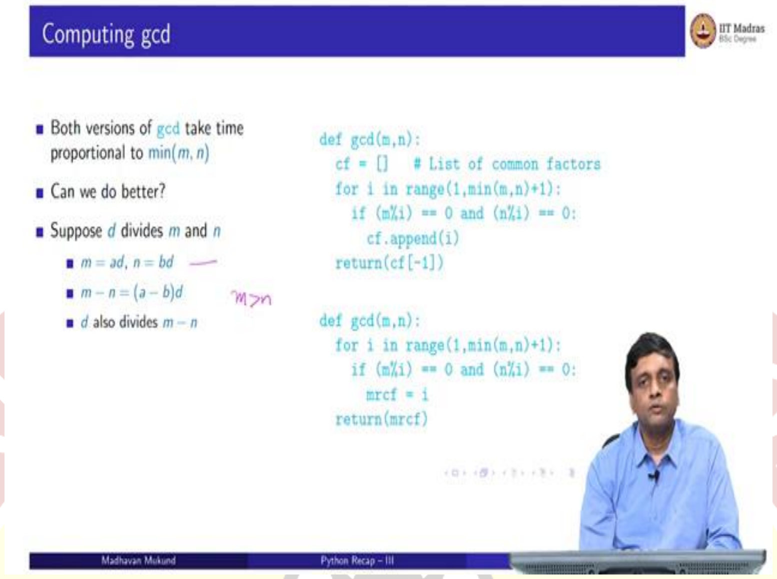
```
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])

def gcd(m,n):
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      mrcf = i
  return(mrcf)
```

So, let us continue our recap of python and go back to the gcd which we saw in the first of these recap lectures. So, remember that the gcd is the number which, the largest divisor of both m and n and we had two versions of gcd that we had computed, one which actually computed the list of common factors between m and n and then found the last one, so this is the last common factor or we said we do not actually need the entire list because only the last one is important, so I can just incrementally keep track of this most recent common factor.

But in both of these the operational aspect that has to be executed is this for loop, so I have to run over all i ranging from 1 to the minimum of m and n and this is something that we wanted to improve on.

(Refer Slide Time: 1:02)


Computing gcd

- Both versions of gcd take time proportional to $\min(m,n)$
- Can we do better?
- Suppose d divides m and n
  - $m = ad$, $n = bd$
  - $m - n = (a-b)d$ $m > n$
  - d also divides $m - n$

```
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])

def gcd(m,n):
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      mrcf = i
  return(mrcf)
```

Madhavan Mukund Python Recap – III

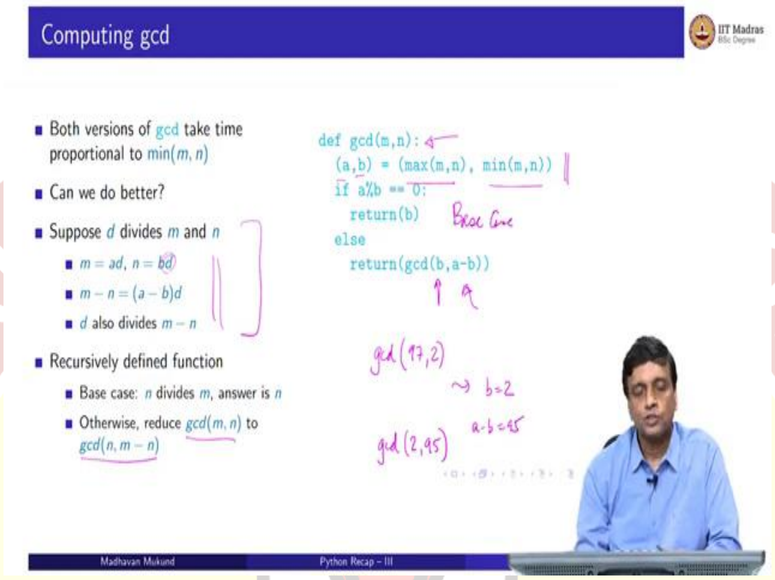
So, let us see whether there is a kind of different strategy to look at gcd, instead of, so what we have done right now is what you might call the naive or the brute force strategy. We took the took the definition of factors and we just literally computed all the common factors and tried to find the largest common factor.

So, sometimes you need to think in a kind of different or orthogonal way about the problem in order to find a better way to solve it. So, here is a different way of thinking about it. So, supposing d is indeed a common factor of m and n, so d divides m and d divides n, that means I can write m as a multiple of d some a times d, and b is also a multiple of d some b times d, n is a multiple of d b times d, so m and n are both multiples of d because d by assumption is something that divides both m and n.

But now let me look at m minus n then with some very simple algebra you can say that m minus n is a d minus b d and therefore it is a minus b times d. So, assuming that, let us assume for the moment it does not really matter if m is bigger than n, a minus b is going to be positive if m is smaller than n, m minus, a minus b is going to be negative, but the main point is that d is also a number that divides m minus n.

Now, m minus n is a smaller number. So, what we are really saying is that if we want to solve a problem with m and n, we can actually reduce it to a problem with smaller numbers, so we can keep simplifying the problem we are solving and come up with a final solution which is based on very small numbers.

(Refer Slide Time: 2:44)



Computing gcd

- Both versions of gcd take time proportional to $\min(m, n)$
- Can we do better?
- Suppose d divides m and n
  - $m = ad$, $n = bd$
  - $m - n = (a - b)d$
  - d also divides $m - n$
- Recursively defined function
  - Base case: n divides m, answer is n
  - Otherwise, reduce $gcd(m, n)$ to $gcd(n, m - n)$

```
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, these are what are called recursive definitions, so we want a function which is recursively defined so we have to have a situation where we can declare that we know the answer and the situation we know the answer is when one of the numbers divides (()) (2:56), supposing I ask you what is the gcd of 9 and 3, then you know that 3 actually divides 9, so you can just return 3 because if the smaller of the 2 numbers actually divides the bigger number directly then it is the gcd, because it is the largest factor of itself and there is no larger factor of 3 than 3 and 3 divides 9.

So, if the second number, we are assuming the second number is smaller in this category. So, if n divides m then we can just directly say that we are done, if n does not divide m then we have to do some work. So, earlier our work consisted of actually explicitly computing all the factors of m and all the factors of n, up to the minimum of m comma n.

Instead what we are going to do is exploit this observation and say let us take the smaller number n and look at the gcd of that number with the difference m minus n. So, we reduce this problem of m n to the problem of n m minus n where n we assume is small. So, here is how we write it, so we have this gcd of m comma n but we do not know of course whether m is bigger than n or smaller than n.

So, what we do is we first figure out which is bigger and which is smaller and call them a and b. So, a is the bigger one the maximum, and b is the smaller one the minimum. So, now if the smaller one that is b divides a, so a divided by b has remainder 0, then b is already the gcd and I am done so this is the base case.

Otherwise I take the smaller number which I know is b because I have already computed that b is a minimum and I take the difference a minus b. So, you might be tempted to think that after this a minus b will be smaller than b but it is not the case in general. So, supposing I say something like gcd of 97 comma 2 then 2 does not divide 97 because 97 is an odd number.

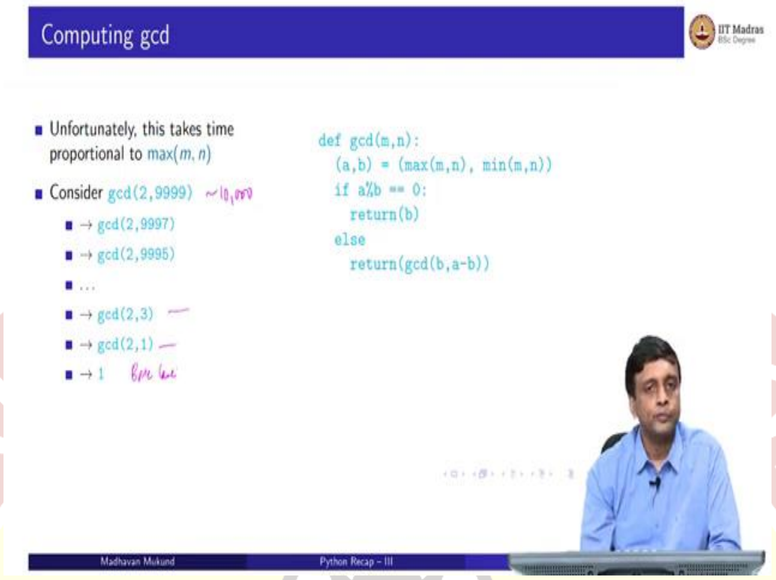
So, I will end up saying that b is 2 and a minus b is 95. So, the next call will be gcd of b and b minus a, a minus b, so it will be 2 comma 95. So, in general each time when I call this gcd recursively I cannot I am not guaranteeing that I am calling it with the right order bigger and smaller which is why I need to fix that inside the function, so I get two numbers I first identify which is smaller which is bigger and then I proceed.

But otherwise this now you see that there are no factors, no list nothing, I am not explicitly keeping track of the factors of any number, I am only signaling when one number is smaller than the other number, divides the other number. So, now because of this property, supposing a number divides both m and n then it must also divide m minus n.

So, in particular if you look at the gcd, the largest number that divides m and n that number is also a d of some form, so the largest number that divides m and n also divides m minus n, so at every point when I reduce the problem I am not changing the value I report, the gcd before and the gcd after are the same, anything that divides m comma n also divides n m minus n.

So, by reducing this problem each time I am guaranteeing that the answer does not change and finally I reduce it to a simple case where I just check whether smaller number divides a bigger number. So, this is our recursive function.

(Refer Slide Time: 6:32)



Computing gcd

- Unfortunately, this takes time proportional to max(m, n)
- Consider gcd(2,9999) ~10,000
    - → gcd(2,9997)
    - → gcd(2,9995)
    - ...
    - → gcd(2,3)
    - → gcd(2,1)
    - → 1 Base case

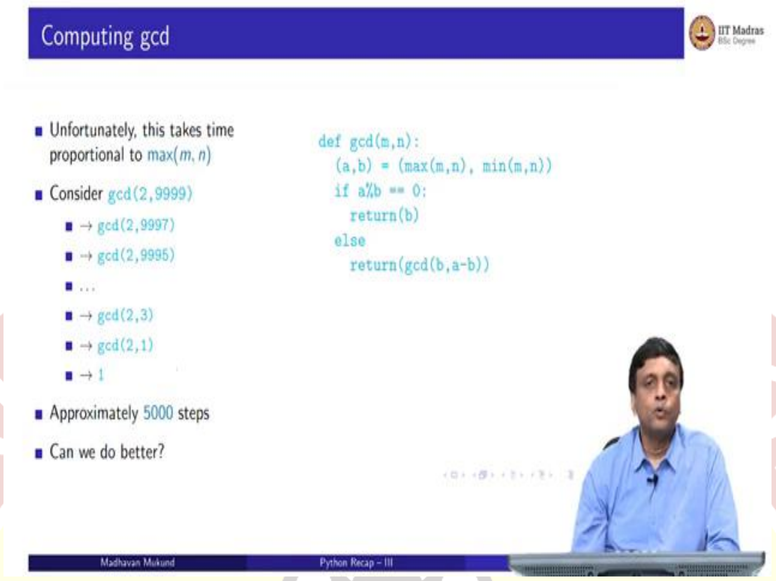
```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a-b))
```

Now, the problem with this is actually that this does not solve our original problem, our original problem was when we were computing this list of factors it was taking time proportional to the minimum of m and n. Now, here actually is going to be a little worse, it is going to take time proportional to the maximum of m and n. So, to understand this let us look at an example.

So, supposing I take the gcd of 2 and some large odd number, then as we saw b minus, b is going to be 2 and a minus b is going to be the large odd number minus 2. So, 9999 is going to reduce to 9997, that is going to reduce by 2 to 9995, so I am going to keep doing this approximately so I am reducing by 2 so this is approximately 10000 the second number.

So, if I keep reducing by 2 after about 5000 steps I will come down to 2 comma 3 and when I do 2 comma 3 and I do this thing now 2 becomes a smaller number so then I get 3 minus 2 the difference is 1 and so I get gcd of 2, 1 and now since 1 divides 2 finally I use the base case to get the fact which was kind of obvious right from the beginning that I have 2 and I have an odd number, then I do not have any possibility that that number is divisible by 2. So, the gcd is 2 saying there is no common factor between 2 and any odd number.

(Refer Slide Time: 8:00)



Computing gcd

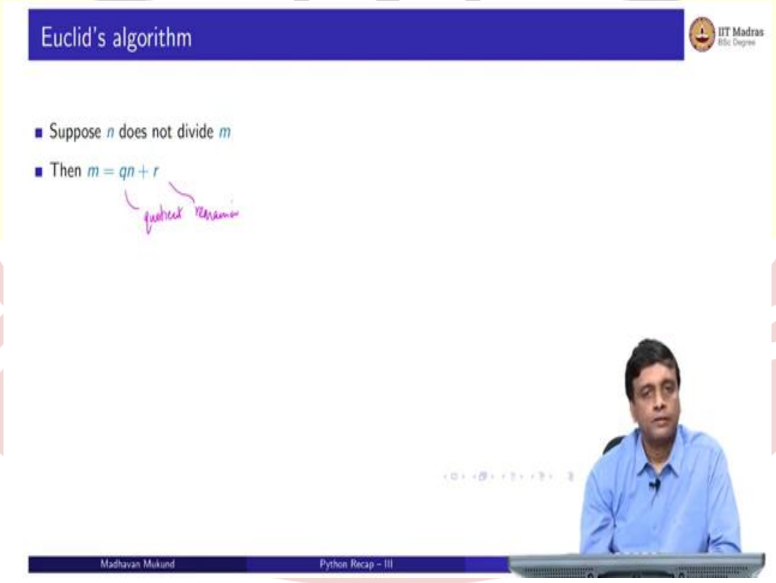
- Unfortunately, this takes time proportional to $\max(m, n)$
- Consider `gcd(2,9999)`
    - → `gcd(2,9997)`
    - → `gcd(2,9995)`
    - …
    - → `gcd(2,3)`
    - → `gcd(2,1)`
    - → `1`
- Approximately 5000 steps
- Can we do better?

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

Madhavan Mukund Python Recap – III

So, the problem here is that this takes approximately 5000 steps, so this is not a nice thing as we will see, so we would like to do something much faster than this. But we can use the basis of this recursive thing to come up with a more clever recursive thing.

(Refer Slide Time: 8:18)



Euclid's algorithm

- Suppose n does not divide m
- Then $m = \underbrace{q}_{\text{quotient}}n + \underbrace{r}_{\text{remainder}}$

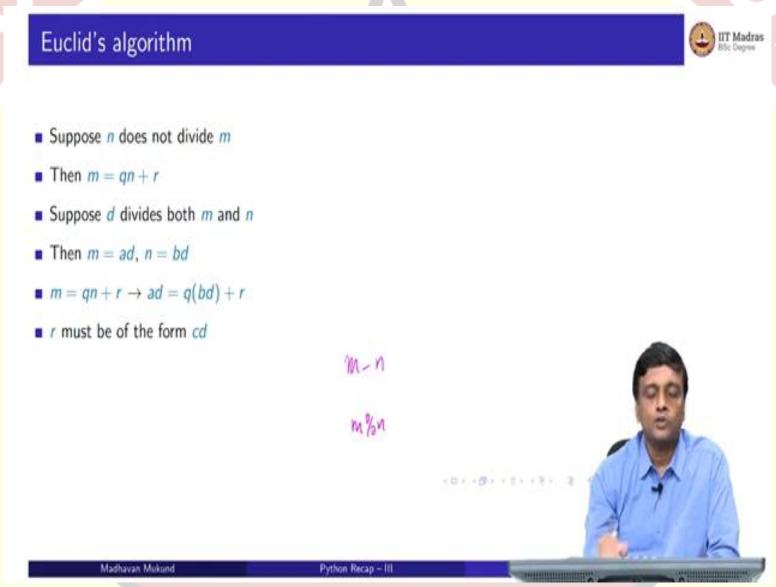
Madhavan Mukund Python Recap – III

So, this is what is called Euclid’s algorithm. So, suppose now earlier we looked at common factors we said suppose d divides both, now let us suppose that we are not in the base case, the second number does not divide the first number that is where we have to figure out what to do next.

If the second number n divides m then we are done, so the question is what do we do when we are not finished, how do we reduce it to a smaller problem. So, in the earlier case we took m minus n but now we want to do something more clever than that. So, if n does not divide m then when I divide m by n there is a remainder.

So, this is written as, this is if you remember from school this is called the quotient and the remainder so m divided by n it goes a certain number of times so that it goes into it q times and leaves a remainder r, so I can write m as q times n plus r.

(Refer Slide Time: 9:13)



Euclid's algorithm

IIT Madras
BSc Degree

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd

$m - n$

$m \% n$

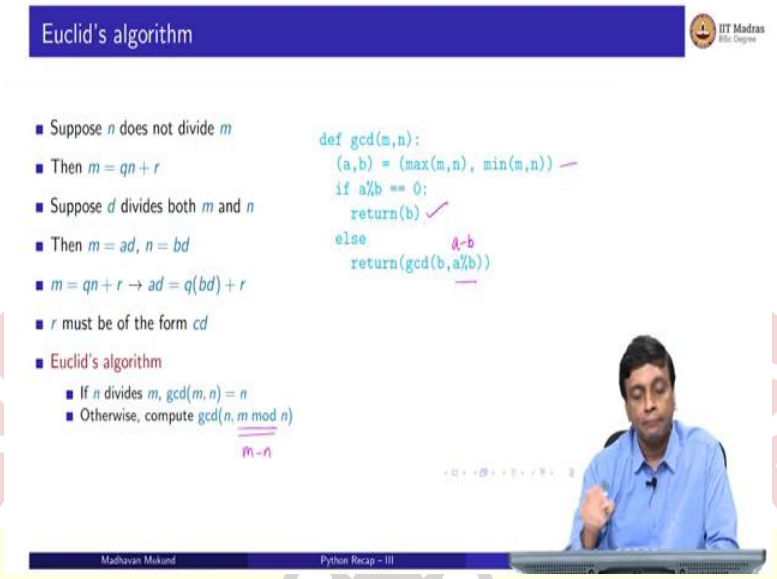
Madhavan Mukund — Python Recap – III

Now, let us look at a possible common divisor d for both m and n. So, as before if d divides m and d divides n, I can write each of them as a multiple of d. So, I can write m as a times d, I can write n as b times d. So, now if I look at the equation above m is equal to q n plus r, I can replace this m by a times d and I can replace this n by b times d.

So, I have on the left something which is a multiple of d and I have something on the right which is a multiple of d plus something. Now, this is not very difficult to show but you can easily justify to yourself that if the left hand side is something times d then the right hand side must also be something times d and if d is a factor here then d must also be a factor there.

So, from this expression you can conclude that r must also be of the form something times d, so let us call it c times d. So, in other words when I find that n does not divide m not only does m minus n as we saw before have d as a factor but m remainder n also has d as a factor.

(Refer Slide Time: 10:24)



Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
  - If n divides m, $\gcd(m, n) = n$
  - Otherwise, compute $\gcd(n, \underline{\underline{m \bmod n}})$ m-n

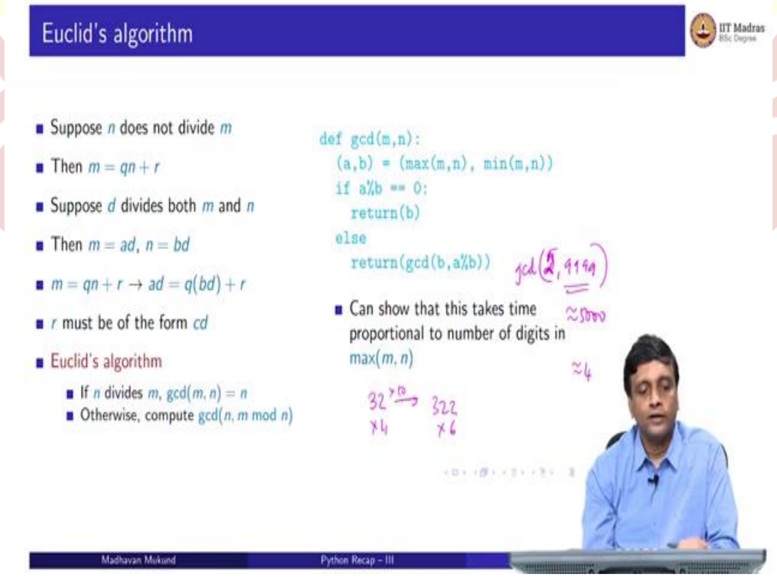
```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b) ✓
  else
    return(gcd(b,a%b))
```

a-b

Madhavan Mukund Python Recap – III

So, this gives us what is called Euclid's algorithm. So, in Euclid's algorithm the difference is that we instead of taking m minus n as in the previous case we take in mathematical terms m mod n, m mod n is the remainder which in python terms is this percent operator, so the only difference between the previous thing and this is earlier it was a minus b, now it is a percent b, everything else is the same, I compute a to be the maximum, b to be the minimum, if b divides a then I return b as before, the smaller number divides the larger number it is the gcd, otherwise I go down to the smaller number and the remainder not the difference.

(Refer Slide Time: 11:06)



Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
  - If n divides m, $\gcd(m, n) = n$
  - Otherwise, compute $\gcd(n, m \bmod n)$

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

$\gcd(2, \underline{\underline{9999}})$

- Can show that this takes time proportional to number of digits in $\max(m, n)$

≈ 5000

≈ 4

$32 \xrightarrow{\times 10} 322$

$\times 4 \quad \times 6$

Madhavan Mukund Python Recap – III

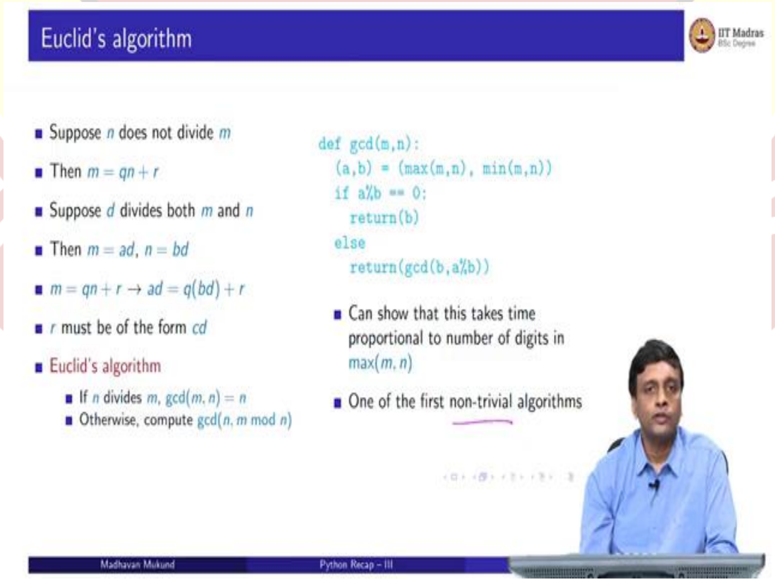
So, we can actually show that this is a dramatic improvement, so this takes time proportional to the number of digits so earlier we had looked at for example the gcd of 2 and 9999, and we said that this is going to take something like 5000 steps because it takes time proportionate to the maximum of the two numbers but this gcd will actually take something proportional to 4 because there are 4 digits in this.

So, in general when we are dealing with numbers this is what we would like, so think of multiplication, if I take 32 times 4 and I take 322 times 6, my number on the right has multiplied by a factor of 10 but clearly if I do the multiplication the extra work that I get is I have to multiply one more digit.

So, a reasonable problem, a reasonable algorithm for any numerical problem like this should grow as the number of digits, it would be tremendously difficult for us if in order to add two 4 digit numbers it takes 10 times as long as it takes to add two 3 digit numbers, it does not, we just have to carry one more step.

So, if I have 4 digits it takes me so many time, if I take 5 digit I have one more carry to answer, one more column to add. So, we really want things which are proportional to the number of digits and it turns out we are not going to prove it here, but Euclid's algorithm has this property.

(Refer Slide Time: 12:32)



Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
  - If n divides m, $\gcd(m, n) = n$
  - Otherwise, compute $\gcd(n, m \bmod n)$

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

- Can show that this takes time proportional to number of digits in $\max(m, n)$
- One of the first non-trivial algorithms

Madhavan Mukund Python Recap – III

The other important reason to look at Euclid's algorithm other than it being a nice example is actually is one of the first examples that we know of a non-trivial algorithm. So, remember that we have started with the problem talking about divisors and factors and all that and we have completely translated it into some properties of numbers and remainders.

So, we are kind of approach the problem from a different angle and come up with a significantly better solution. So, this is what I mean by a non-trivial algorithm, it is not an obvious algorithm and it significantly improves the performance of the obvious algorithm.