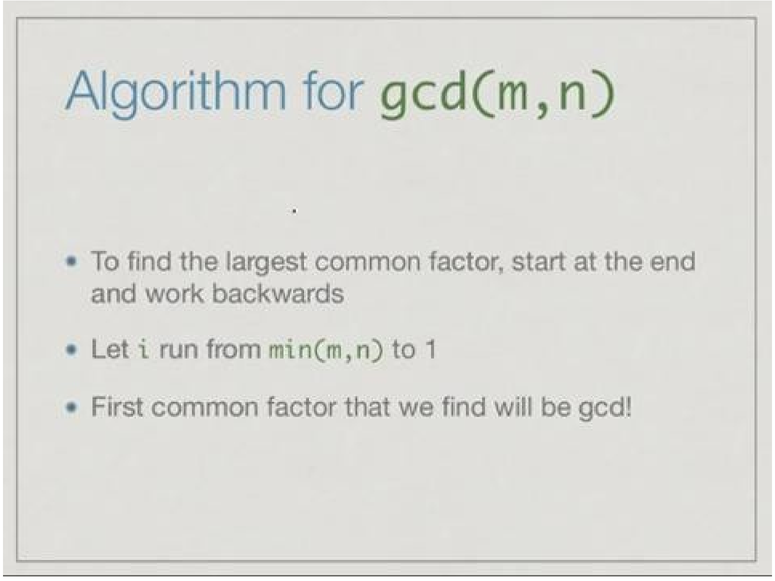
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 01
Lecture - 03
Euclid's Algorithm for gcd

Let us continue with our running example of gcd to explore more issues involved with program.

(Refer Slide Time: 00:09)



Algorithm for gcd(m,n)

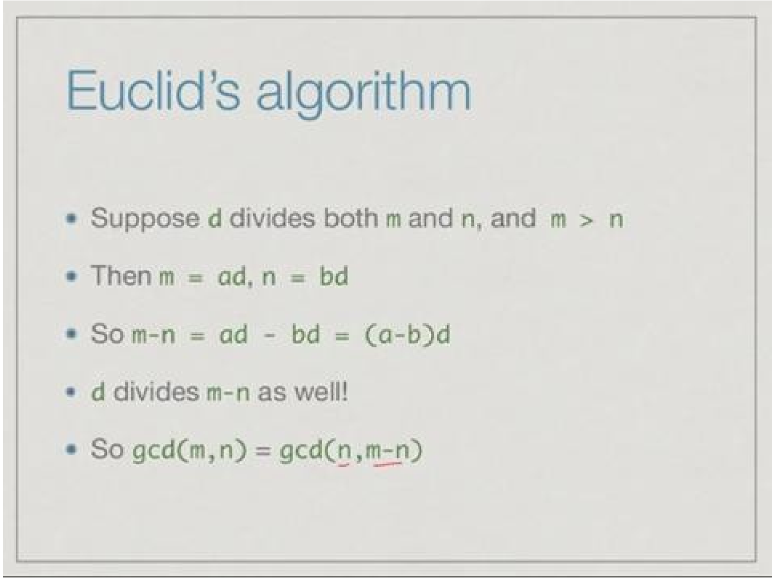
- To find the largest common factor, start at the end and work backwards
- Let i run from min(m,n) to 1
- First common factor that we find will be gcd!

We started with the basic definition of gcd, which said that we should first compute all the factors of m, store it in a list, compute all the factors of n, store it in another list, from these two lists, extract the list of common factors and report the largest one in this common factor list. Our first simplification was to observe that we can actually do a single pass from 1 to the minimum of m and n and directly compute the list of common factors without first separately computing the factors on m and the factors of n. We then observe that we don't even need this list of common factors since our interest is only in the greatest common factor or the greatest common divisor. So, we may as well just keep track of the largest common factor we have seen so far in a single name and report it at the end.

Our final simplification was to observe that if we are interested in the largest common factor, we should start at the end and not the beginning. So, instead of starting from 1 and working upwards to the minimum of m and n its better to start with minimum of m and n and work backwards to one, and as soon as we find a common factor we report it and exit.

Remember always that 1 is guaranteed to be a common factor. So when we start from minimum of m and n and go backwards, if we don't see any other common factor, we are still guaranteed that we will exit correctly when we hit one. So what we notice that was, that though these different versions are simpler than the earlier versions they all have the same efficiency in terms of computation, which is that they will force us in the worst case to run through all the numbers between 1 and the minimum of m and n, before we find the greatest common factor whether we work forwards or backwards.

(Refer Slide Time: 02:08)



Euclid's algorithm

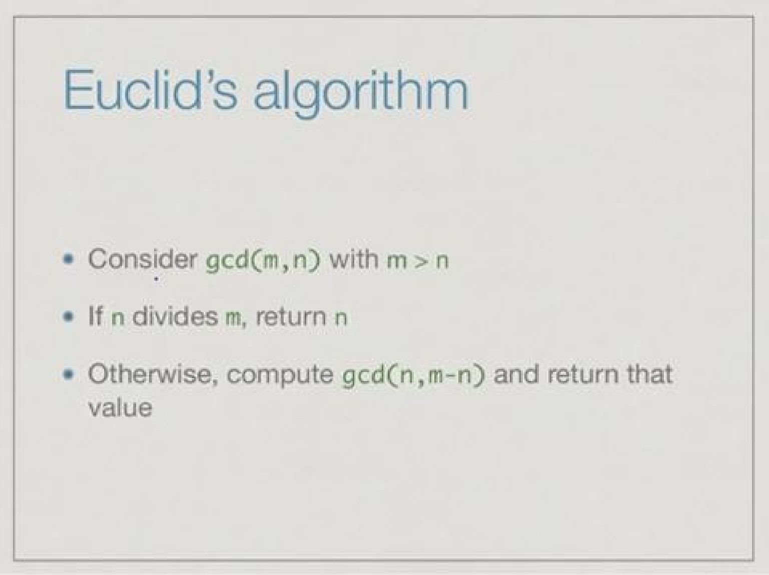
- Suppose d divides both m and n, and m > n
- Then m = ad, n = bd
- So m-n = ad - bd = (a-b)d
- d divides m-n as well!
- So gcd(m,n) = gcd(n,m-n)

So at the time of the ancients Greeks, what was possibly the first algorithm in modem terminology was discovered by Euclid, and that was for this problem - gcd. So what Euclid said was the following. Suppose we have a divisor d which divides both m and n, so this is a common divisor and we are looking for the largest such d. Let us assume also for the sake of argument that m is greater than n. So if d divides both m and n, we can write m as a times d and n as b times d for some values a and b, so m is multiple of d and so is n.

So if we subtract the equations then the left hand side is m minus n. So, we take m and subtract n from m, so correspondingly we subtract b d from a d. So, m minus n is equal to a d minus b d, but since d is a common term this means m minus n is a minus b times d. This is where we are using the assumption that m is greater than n, so a minus b will be a positive number. But the important thing to note is that m minus n is also a multiple of d. In other words, if d divides both m and n, it also divides m minus n. And since d is the largest divisor of m and n, it will turn out that d is also the largest divisor which is common to m, n and m minus n.

In other words, the gcd of m and n is the same as the gcd of the smaller of the two, namely n and the difference of the two m and n, m minus n. So, we can use this to drastically simplify the process of finding the gcd.

(Refer Slide Time: 04:00)

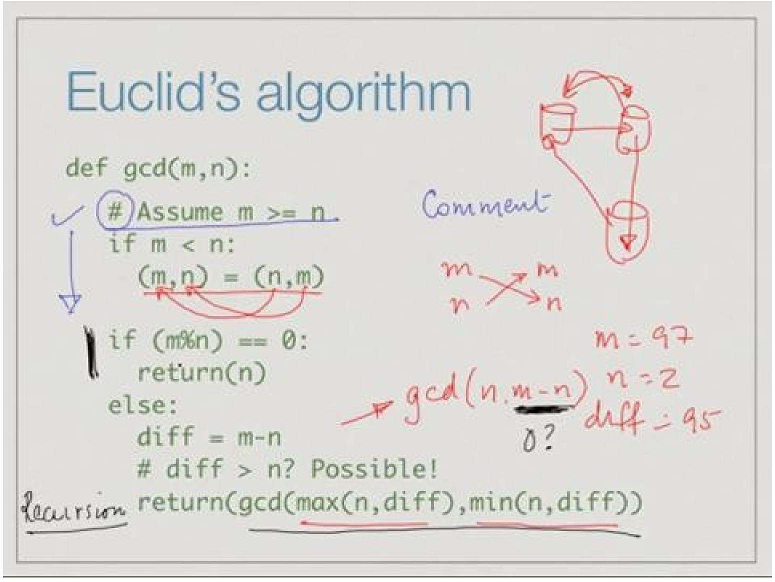


Euclid's algorithm

- Consider gcd(m,n) with m > n
- If n divides m, return n
- Otherwise, compute gcd(n,m-n) and return that value

So here is the first version of Euclid's algorithm. So, consider the value: gcd of m n assuming that m is greater than n. So if n is already a divisor of m, then we are done and we return n. Otherwise, we transform the problem into a new one and instead of computing the gcd of m and n that we started with, we compute the gcd of n and m minus n and return that value instead.

(Refer Slide Time: 04:32)



Euclid's algorithm

```
def gcd(m,n):
  # Assume m >= n
  if m < n:
    (m,n) = (n,m)
  if (m%n) == 0:
    return(n)
  else:
    diff = m-n
    # diff > n? Possible!
    return(gcd(max(n,diff),min(n,diff))
```

So, here is a python implementation of this idea. There are a couple of new features that are introduced here, so let us look at them. The first is this special statement which starts with symbol hash. So in python, this kind of a statement is called a comment.

So a comment is a statement that you put into a program to explain what is going on to a person reading the program, but it is ignored by the computer executing the program. So, this statement says that we are assuming that m is bigger than or equal to n. So, this tells us that when the program continues this is the assumption. Of course, it is possible that the person who invokes gcd does not realize this, so they might invoke it with m smaller than n and so we fix it.

This is a special kind of assignment which is peculiar to python; it is not present in most other programming languages. So what we want to do is, basically we want to take the values m and n and we want to exchange them, right. We want to make the new value of m, the old value of n and the new value of n, the old value of m, so that in case m and n were in the wrong order we reverse them. So, what this python statement does is it takes a pair of values and it does a simultaneous assignment so it says that the value of n goes into the value of m and the value of m goes into the value of n.

Now it is important that it is simultaneous, because if you do it in either order, if you first copy the value of n into m, then the old value of n is lost. So, you cannot copy the old value of m into the new value of n because you have lost it. So imagine that you have two mugs of water, and now you want to exchange their contents. Now you have to make space, you cannot pour this into that without getting rid of that and once you got rid of that you cannot pour that into that, so you need third mug normally.

You need to first transfer this here and keep it safe, and then you need to transfer this there and then you need to copy it back. So this is the normal way that most programming languages would ask you to exchange two values, but python has this nifty feature by which you can take a pair of values and simultaneously update them and in particularly this simultaneous update allows us to exchange the values without worrying about having this extra temporary place to park one value.

Anyway, all that this first part is doing is to ensure that this condition that we have assumed is satisfied. So now we come to the crux of the algorithm. If m divides n that is remainder of m divided by n is 0 then we have found n to be the gcd and we return n. If this is not the case, then we go back to what we discovered in the last slide and we now are going to compute gcd of n and the difference m minus n. We would ideally like to compute gcd of n and m minus n. So, we compute the difference m minus n and we could just invoke this.

But, it is possible, for example - if m is say 97 and n is 2 then the difference will be 95. The difference could very well be larger than n, and we would ideally like to call this function with the first number bigger than the larger number. So we will just ensure this even though our function does take care of this. What we want to do is, we want to call gcd with n and m minus n instead we will call gcd with the maximum value of n and the difference as a first argument and the minimum value of n and the difference. So it will make sure that the bigger of the two values goes first and the smaller of the two values go. And whatever this gcd, the new gcd returns is what this function will return.

This is an example of what we will see later, which is quite natural, which is called Recursion. Recursion means, that we are going to solve this problem by solving the smaller problem and using that answer in this case directly to report the answer for our current problem. So we want to solve the gcd of m and n, but the gcd of m and n instead we solve the gcd n and m minus n and whatever answer that gives us we directly report it back as the gcd for this, so we just invoke the function with the smaller values and then we return it.

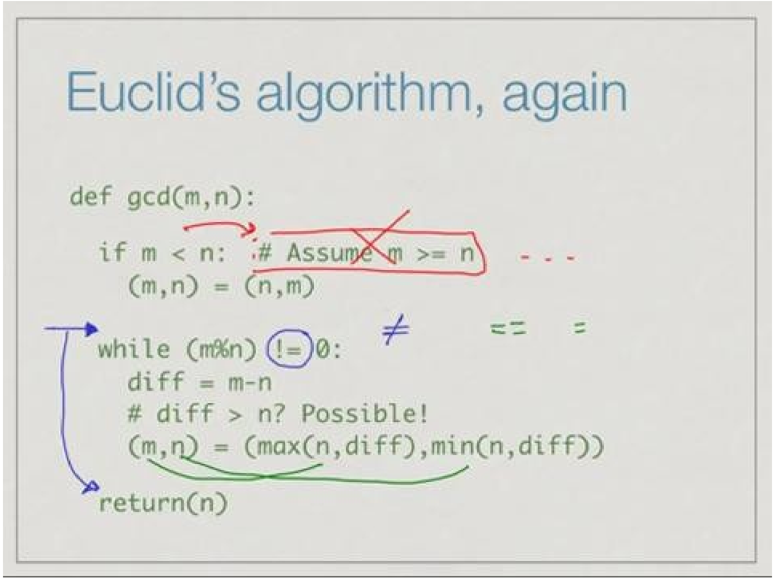
Now whenever you do a recursive call like this, it is like a while loop; it will invoke the function again, that in turn will invoke a smaller function and so on. And you have to make sure that this sequence in which gcd keeps calling gcd with different values does not get to an infinite progression without a stopping point. So, formally what we have to ensure is that this guarantee of finding an n which divides m, so this is where gcd actually exits without calling itself. We have to make sure that eventually we will reach this point. Now what is happening if you see here is that the values that are passed to gcd are getting smaller and smaller.

Now what can we have for m minus n? What can be the value? Can it be 0? Well, if m minus n is 0 that means m is equal to n, if m is equal to n then certainly m is divisible by n. If m minus n is 0 then it could have exited, so it cannot be 0. It must be at least 1, so whenever we call this function m minus n it's at least one. On the other hand each time we are reaching smaller values. So, we start with some value and m minus n keeps decreasing.

What happens when it actually reaches 1? Well, when it reaches 1 then 1 divides every other number, so m percent n or m divided by n, the remainder will be 0, so we will return gcd of 0. In other words, we had guaranteed that this function because it keeps reducing the number that we invoke the function with will eventually produce a call where gcd terminates. This is important and we will come back to this later but whenever you write a function like this, you have to make sure that there is a base case which will be reached in a finite number of steps no matter where you start.

This is Euclid’s algorithm, the first version where we observe that the gcd of m and n can be replaced by the gcd n and m minus n. And what we have seen in this particular implementation are three things rather, we have seen how to put a comment in our code, we have seen that python allows this kind of simultaneous updation of two variables at the same time so m comma n equal to n comma m. We have also seen that we can use the same function with new arguments in order to compute the current functions. So there is no problem with saying that in order to compute gcd of m and n, I well instead compute gcd's on some other value and use that answer to return my answer.

(Refer Slide Time: 11:53)



Euclid’s algorithm, again

```
def gcd(m,n):

  if m < n:  # Assume m >= n
    (m,n) = (n,m)

  while (m%n) != 0:
    diff = m-n
    # diff > n? Possible!
    (m,n) = (max(n,diff),min(n,diff))
  return(n)
```

Let us look at a different version of this algorithm, where we replace the recursive call by a while loop. We saw while in our last version of this standard algorithm when we were counting down from minimum of m comma n to 1, so we kept checking whether i was greater than 0 and we kept decrementing. Well, here we are doing the recursion using a while, so the first thing to notice here is that I have moved this comment which used to be in a separate line to the end of the line.

What python says is that, if there is hash then the rest of the line can be ignored. So, it reads this line it sees a valid condition and then sees the hash, so it’s as though this statement was not part of the python program when it is executed. Comment can either be in a separate line or it can be in end of a line. Of course, remember that you cannot put anything after this which you want python to execute because once it sees a hash the rest of the line is going to be ignored, so it cannot be in middle of a line you cannot put a comment in middle of a line, but you can put it on separate line or you can put it at end of the line.

So anyway so this is our comment as before. So up to here there is no change except that I have shifted the comment position. Now we reach this point where we actually have to do some computation. At this point if we have found n such that n divides m we are done and we can directly return n. So, this is what our recursive code would do. If we have not found such an n we have to do some work. The condition is to check whether m divided by n actually produces a remainder. So, this not equal to symbol is return with this exclamation mark, so this is the same as the mathematical not equal to.

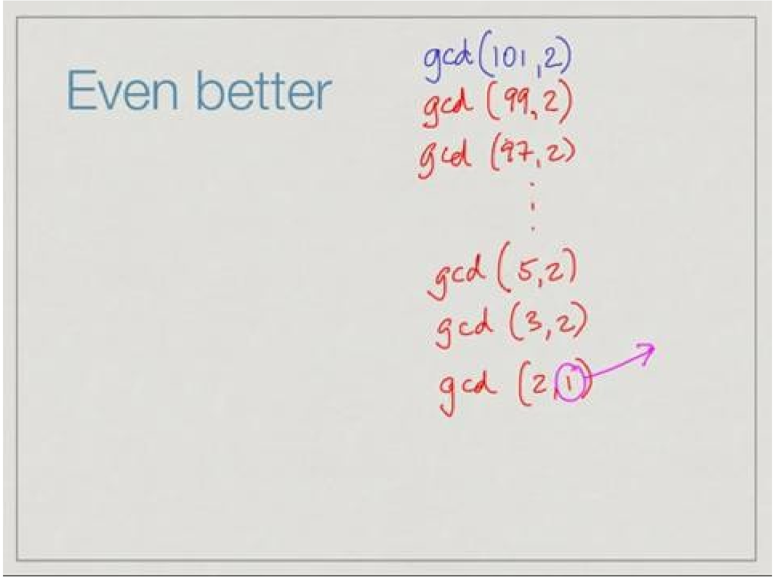
Remember that this double equal to was what we use for the mathematical symbol of equality. This is our symbol for not equal to. So, so long as there is remainder, that is the remainder m divided by n is not 0 we do what we did before we compute the difference and we replace m by the maximum of the two values and n by the smaller of the two values. We have a pair m n whose gcd we are trying to find right, with assumption that m is bigger than n at each step we replace m by the larger of n and the difference and n by the smaller of n and the difference.

This exactly what we are doing in the recursive call, we are saying pretend we are computing gcd of that. Here in this while loop effectively we are saying replace the gcd of m n by the computation of maximum n diff and minimum n diff. We keep doing this until we hit a condition where n actually divides m, and exactly like we said in the recursive case that there will be a boundary case where at worst case n will become 1 and 1 will divide everything.

In the same way here the difference will keep reducing, the difference cannot be 0, because if difference is 0 it could have divided, so difference can at most go down to 1 and when it hits one we are done. This a while version of the same recursive function we wrote earlier, so if it helps you can look at these side by side and try to understand what this recursive things is doing and what the while is doing and see that they are basically doing the same thing.

And the idea that the recursion must terminate is exactly analogous to the claim that we said earlier that when you write a while you must make sure that you make progress towards making the while condition false, so that the while exits. So, just like the recursion can go on forever, if you are not careful and you do not invoke it with arguments which guarantee termination, the while can also go on forever if you do not make progress within the while in order to make sure that the while condition eventually becomes false.

(Refer Slide Time: 15:42)



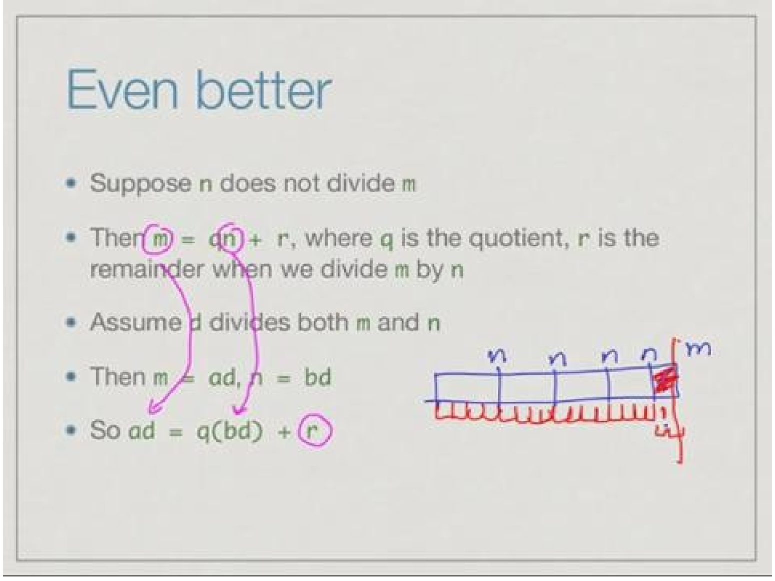
Even better

gcd(101,2)
gcd (99,2)
gcd (97,2)
⋮
gcd (5,2)
gcd (3,2)
gcd (2,1)

We can actually do a little better than this. Let us see one problem with this by doing a hand execution. So supposing we start with some number like gcd of 101 and 2, then our algorithm will say that this should now become gcd of the difference and n, the difference is 99 so will have 99 and 2, and then this will become gcd of 97 and 2 and so on. So, we will keep doing this about 50 steps then eventually we will come down to gcd of 5 and 2, and then gcd of 3 and 2. Now when we compute the difference we get gcd of 2 and 1, so now the difference will become smaller. Then at this point we will report that the answer is 1. So, it actually takes us about 50 steps in order to do gcd of 101 into 2.

One of our criticisms of naive approach is that it takes time proportional to the numbers themselves. If you had numbers m and n we would take in general number of steps equal to minimum of m and n. Now here, in fact we are taking steps larger than the minimum because the minimum is 2, if you were just computing factors we will see that the only factor of 2 is 2 and it is not a factor 101 we would have stopped right at beginning. This actually seems to be worst then our earlier algorithm in certain cases.

(Refer Slide Time: 17:05)



Even better

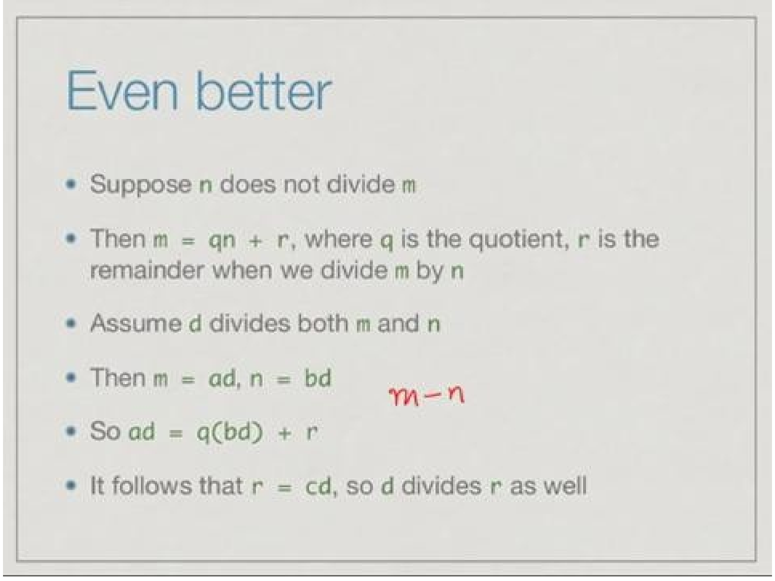
- Suppose n does not divide m
- Then m = qn + r, where q is the quotient, r is the remainder when we divide m by n
- Assume d divides both m and n
- Then m = ad, n = bd
- So ad = q(bd) + r

Here is a better observation suppose n does not divide m. In other words if I divide m by n i will get a quotient and a remainder. So, I can write n as q times n plus r where q is quotient and r is the remainder, so you may remember these terms from high school arithmetic. N goes into n q times and leaves a remainder r and we are guaranteed that r is smaller than n, otherwise r it could go one more time it will become q plus 1. We have the remainder r which is smaller than n. So for example if i say 7 and i want to divide it by 3 for example, this will be 2 times 3 plus 1, so this will be my quotient and this will be my remainder. And the important thing is remainder is always smaller than what I am dividing by.

Now, let us assume as before that we have a common divisor for both m and n. In other words like before we can write m itself as a times d and n as b times d for some numbers a and b, because m is is multiple of d and so is n. If you plug this into the equation above here, then we see that m which is a times d is equal to q times n which is b times d plus r. So, d divides the left and d divides one part of the right. You can easily convince yourself that d must also divide r.

The way to think about it if you want to pictorially is that I have this number m and I can break it up into units of n and then there is a small bit here. On the other hand if I look at d, d evenly divides everything. So it divides each of these blocks it also divides the whole thing. If I continue with d, it is going to stop exactly at this boundary because d also divides n, therefore d must also divide this last bit which is r exactly. In other words, we can argue very easily that r must also be a multiple of d. So d must divide r as well.

(Refer Slide Time: 19:21)



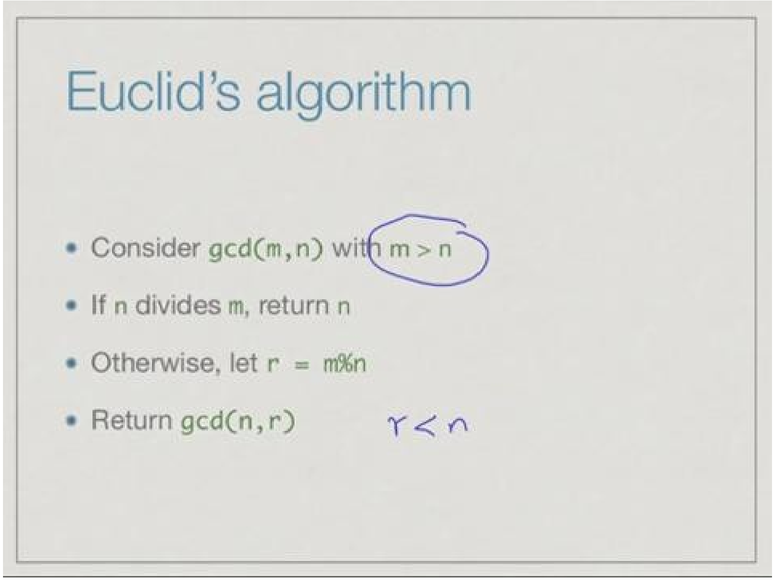
Even better

- Suppose n does not divide m
- Then m = qn + r, where q is the quotient, r is the remainder when we divide m by n
- Assume d divides both m and n
- Then m = ad, n = bd
- So ad = q(bd) + r
- It follows that r = cd, so d divides r as well

m−n

If d divides m and b divides n then d must divide the remainder of m divided by n. And as we saw before with the difference, the last time we said we would look at the difference m divided by n. Now we are saying we look at the remainder of m divided by n and d must divide that and d will be in fact the gcd of n and this remainder.

(Refer Slide Time: 19:44)



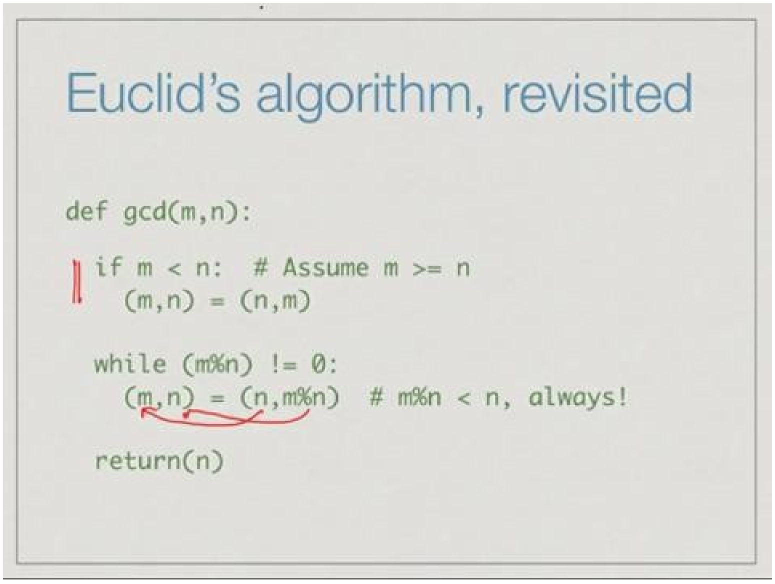
Euclid's algorithm

- Consider gcd(m,n) with m > n
- If n divides m, return n
- Otherwise, let r = m%n
- Return gcd(n,r) $r < n$

This is an improved and this is the actual version of the algorithm that Euclid proposed, not the difference one but the remainder one. It says consider the gcd of m and n assuming that m is bigger them n. Now if n divides m we are done we just return n, this is the same as before.

Otherwise, let r be the remainder with the value of m divided by n get the remainder and return the gcd of n and r, and at this point one important thing to remember is that r is definitely less than n. So we do not have to worry about this condition here, we do not have to take the max and the min as we did for the difference because the remainder is guaranteed to be less than n otherwise n would go one more time.

(Refer Slide Time: 20:31)



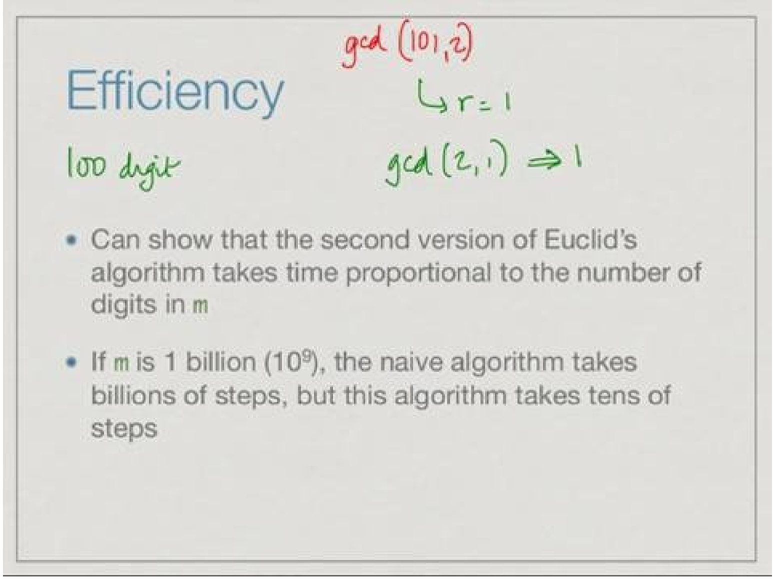
Euclid's algorithm, revisited

```
def gcd(m,n):

  if m < n:  # Assume m >= n
    (m,n) = (n,m)

  while (m%n) != 0:
    (m,n) = (n,m%n)  # m%n < n, always!

  return(n)
```

As before we have very simple recursive implementation of this, and this is even simpler because we do not have to do this max min business. So, like the previous time we first flip m and n in case they are not in the right order. Then if n divides m if the remainder of m divided by n is 0 we return n and we are done, otherwise we return the gcd of n and the remainder, so this is the remainder. And remember that the remainder is always less than n so we do not have to worry about flipping it and taking max and min at this point. And analogous tp the previous case we can do this whole thing using a while instead of doing it with recursive thing.

We first exchange m and n if they are in the wrong order, then so long as the remainder is not 0 we replace m by the smaller of the two numbers and we replace n by the remainder and we proceed. Now we are guaranteed that this remainder will either go to 0, but if it goes to 0 it means it divides or if it's not 0 in the worst case the remainder keeps decreasing because it is always smaller than the number that we started with. So it keeps decreasing and it reaches 1 then in the next step it will divide. So finally, we will return at least one.

(Refer Slide Time: 21:48)



Efficiency

gcd (101,2)
↳ r = 1
100 digit
gcd (2,1) ⇒ 1

- Can show that the second version of Euclid's algorithm takes time proportional to the number of digits in m
- If m is 1 billion (10^9), the naive algorithm takes billions of steps, but this algorithm takes tens of steps

If we go back to the example that we were looking at, so if we saw that gcd 101, 2, and we did it using the difference we said we took about 50 steps. Now here if we do the remainder I am going to directly find that r is equal to 1 right if I divide 101 by 2 it goes 50 times remainder 1. In one step I will go to gcd 2 comma 1 and I will get 1.

In fact, what you can show is that this version with the remainder actually takes time proportional to number of digits, so if I have say hundred digit number it will take about a hundred steps. So for instance if we have a billion as our number, so billion will have about 10 to the 9 will have about ten digits. Then if I do the naive algorithm then it could take some constant times of billion numbers of steps say a billion steps. But this algorithm because of the claim it takes time proportional to number of digits since 10 to the 9 has approximately 10 digits it will only take about 10 steps, so there is a dramatic improvement in efficiency in this version.

This is something that we will touch up on while we are doing this course. This course is about programming, data structures and algorithms. So the programming part talks about what is the best way to express a given idea in a program in a way that it is easy to make sure that it is correct and easy to read and maintain, so that is the programming part. How do you write, how do you express your ideas in the most clear fashion. But the idea itself has to be clear and that is where data structures and algorithm start. So you might write beautiful prose, but you may have no ideas or you may have very brilliant ideas but you may express yourselves clumsily, neither of them is optimal.

This is like writing in any other language. You may have brilliant ideas to express, but if you cannot convey them to the person you are talking to the ideas lose their impact. So, you need ideas and you need a language to express them. Programming is about expressing these ideas, but the ideas themselves come from algorithms and data structures.