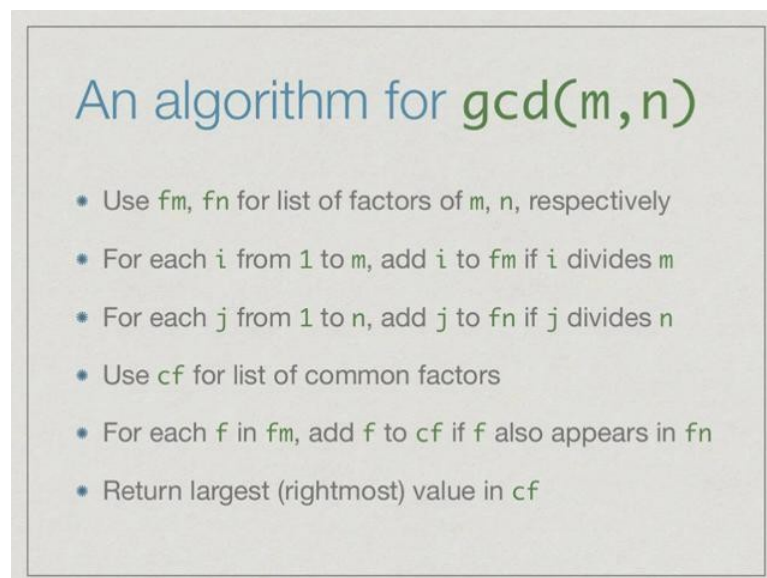**Programming, Data Structures and Algorithms in Python**
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 01
Lecture - 02
Improving naive gcd

In the first lecture we used gcd as an example to introduce some basic concepts in programming. We will continue to look at the same example and see how to refine our program and explore new ideas.

(Refer Slide Time: 00:14)

Here was our basic algorithm for gcd, which as we said more or less follows the definition of the function. We construct two lists of factors for the inputs m and n. So, we construct fm the factors of m, fn the factors of n, and then from these we compute cf the list of factors in both lists or common factors. Our goal is to return the greatest common divisor of the largest number in this common list which happens to be the last one in this list, since we add these factors in ascending order.

(Refer Slide Time: 00:51)

Can we do better?

- We scan from 1 to m to compute fm and again from 1 to n to compute fn
- Why not a single scan from 1 to max(m,n)?
  - For each i in 1 to max(m,n), add i to fm if i divides m and add i to fn if i divides n

So can we do better than this? The way we have proceeded, we first scan all numbers from 1 to m to compute the list fm of factors of m, and then we again start from 1 to n to compute fn. So, an obvious improvement is to just directly scan the numbers from 1 to the larger of m and n and in one scan compute list fm and fn.

In another words for each i in this list 1 to the maximum of m and n we first check if i divides m, if so we add it to the list of factors of m, and then we check if i divides n and if so we add it to list fn. Instead of doing two separate scans over 1 to m and then 1 to n and repeating the past we do it in one scan.
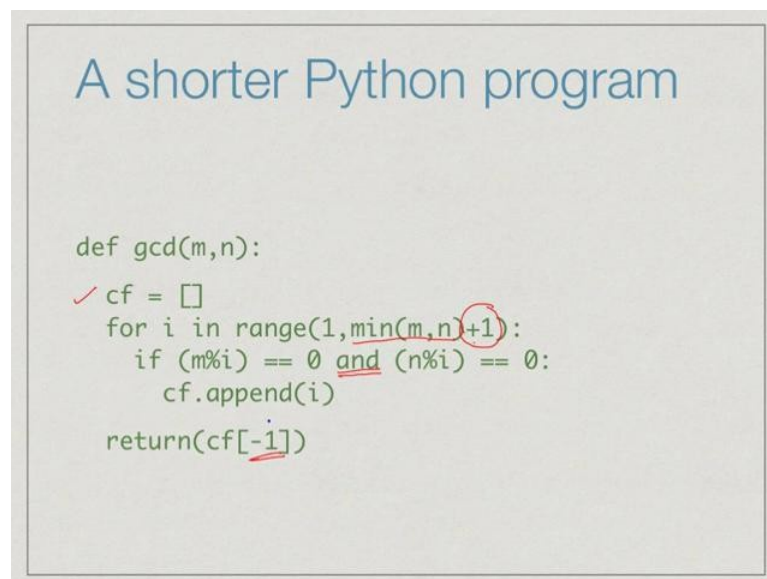
(Refer Slide Time: 01:46)

Even better?

- Why compute two lists and then compare them to compute common factors cf? Do it in one shot.
  - For each i in 1 to max(m,n), if i divides m and i also divides n, then add i to cf
- Actually, any common factor must be less than min(m,n)
  - For each i in 1 to min(m,n), if i divides m and i also divides n, then add i to cf

But even this can be improved upon. If we are doing it in one pass and we are checking if numbers divide both - m and n, then why not we just directly check for common factors. In another words instead of computing two lists and then combining them we can just directly do the following: for each i from 1 to the maximum of m and n, if i divides both m and n then we directly add i to the list of common factors. If it divides neither or if it divides only one of them then it is not a common factor and we can discard.

In fact, notice that rather than going to the maximum of m and n we should go to the minimum of m and n, because once we cross the smaller number we will not get a factor for the smaller number. Remember that the factors of m lie between 1 and m and for n lie between 1 and n. If m is smaller than n for example, if we input m plus 1 though it made a factor of n it certainly cannot be a factor of m. So our better strategy is for each i in the range 1 to the minimum of m, and n if i divides both m and n then we add i to the list of common factors.

(Refer Slide Time: 03:03)


```python
def gcd(m,n):
    cf = []
    for i in range(1,min(m,n)+1):
        if (m%i) == 0 and (n%i) == 0:
            cf.append(i)
    return(cf[-1])
```
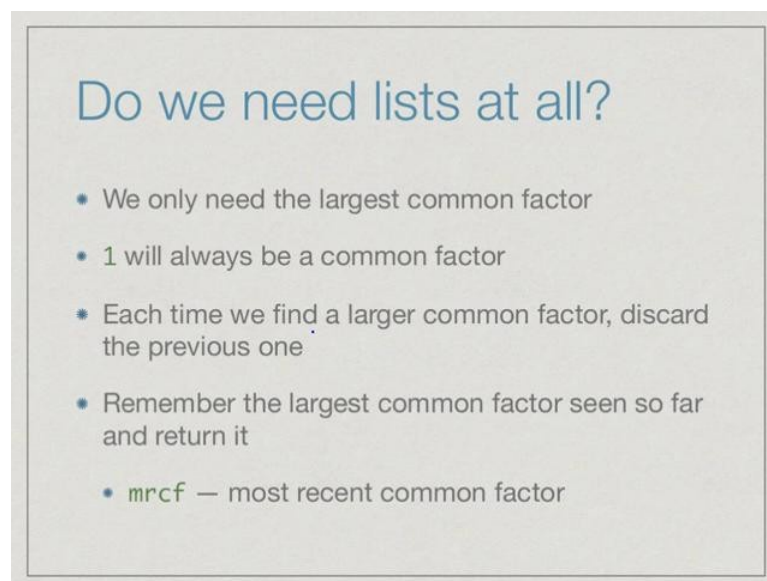
Here is a much shorter Python program implementing this new strategy. So instead of computing the lists fm and fn we directly compute the list of common factors. We let i range from 1 to the minimum of m plus m and n and remember that Python requires us to give the limit of the range is one more than the limit we want to go up to, so we go from 1 to the minimum m n plus 1. And now we have an extra connective it is called a logical connective and which says that we want two conditions to be proved, we want the

remainder when m is divided by i to be 0, in another words i divides m and we also want the remainder when n is divided by i to be 0, so i should divide both m and n and if so we add i to the list of common factors.

And having done so once again we are doing it in ascending order, so the common factors are being added as we go along the larger ones come later. So, we finally want the last element which in Python is given as the minus 1th element of the list cf.

(Refer Slide Time: 04:15)

So having done this, maybe we can simplify things further. Do we need a list of common factors at all? Remember that we only need the largest common factor. We observed that there will always be at least one common factor namely 1. So, the notion of the largest common factor will always be well defined for any pair m and n.

Each time we can start with 1 and each time we find a larger common factor we can discard the previous one, we do not need to remember all the common factors we only need the largest one. So this can be greatly simplifying our strategy because we do not need to keep the list of every possible common factor in this list; we just need to keep the largest one that we have seen. We can use a name say mrcf for the most recent common factor, and keep updating this name with the value of the common factor that we saw last.
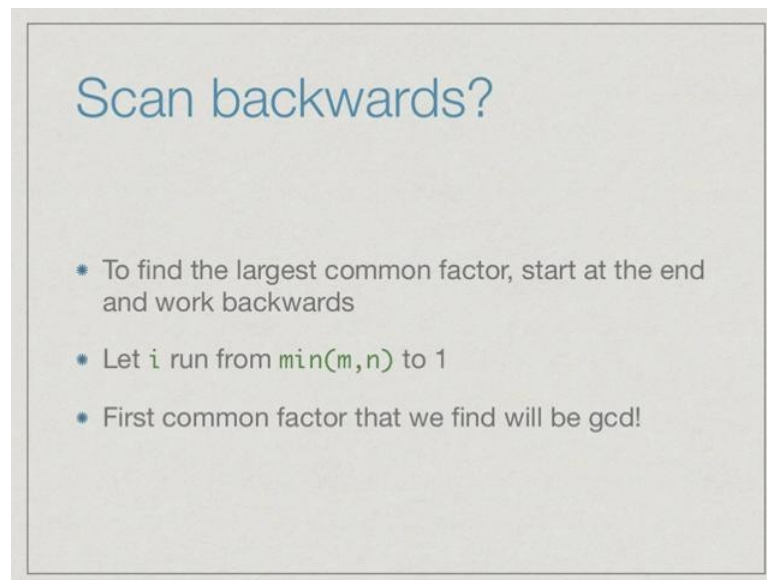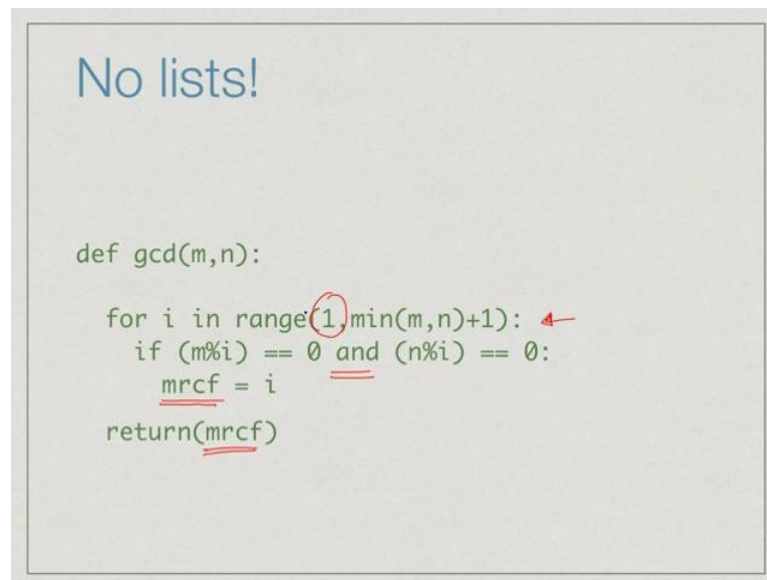
So here is a Python implementation of this idea where we do not have a list at all. We directly scan all the possible common factors from 1 to the minimum of m and n. Whenever we find a common factor we update the value of our name mrcf to be the current common factor that we have found. Now remember that 1 will always be a common factor, so initially mrcf will be assigned the value 1, it will not be that we go through this repeated execution of this and never assign mrcf because the condition is never true.

Since one is the common factor we will at least have mrcf equal to 1, but if we find a larger common factor the one will be replaced by the later common factor. At the end of this for iteration or loop what we end up with is the largest common factor that we found between 1 and the minimum of m and n which is indeed the gcd of m and n.

(Refer Slide Time: 06:18)

We can still do some further optimizations. Since, we are looking for the largest common factor, why do we start at the beginning which will give us the smallest common factor. So, we can start at the end of the list and work backwards. Instead of running from 1 to the minimum of m and n we can start from the minimum of m and n and work backwards to 1. Again the guarantee is that the 1 will always show up as a common factor, so if there are no other common factors at the very end we will find 1 as the greatest common factor.

And in this process we do not have to ever go past any common factor that we find, if we are working backwards from largest to smallest the very first common factor we find must be in fact the gcd of m and n.

(Refer Slide Time: 07:09)
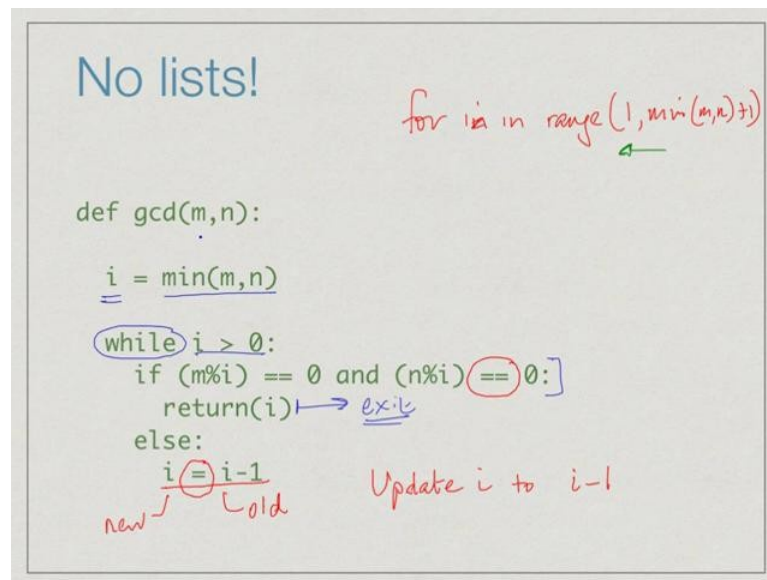
How would we write this in Python? Well, you can modify that for i in range, so notice that normally this function goes from a smaller value to a bigger value, you can modify this to go backwards instead. But instead of doing this which we will see how to do later on when we actually get into formal Python, let us explore a new way of going through a list of values.

We start by assigning to the index i, the value that we want to start with namely the minimum of m and n, remember we want to start at the largest possible value for i and go backwards. So what we have is, a new word instead of for called while, so while as the English word suggests is something that happens while the condition is true. So, while the i that we are looking for is positive. So while i is greater than 0, what do we do? We check if i is a common factor. This is the same as before, we check whether i divides m and i also divides n.

If we find the common factor we are done, we just return the value of i that we have found and we implicitly exit from this function. Every time you see a return statement in a function, the function terminates and the value in the return is what the function gives back to us. So we start with i equal to the minimum of m and n and we check whether i is a common factor if it is so we exit and return the value of i that we last found. And this is the only value that we need, we do not need any other common factors. So, we return the very first time we see a common factor.
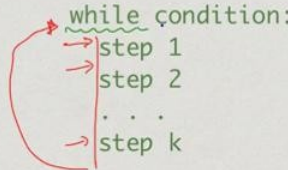
On the other hand, if i is not a common factor we need to proceed by checking the next one which is to go backwards and this is the achieved by this update. So, remember that we said that we can assign values or update values using this equality operation. This equality operation is not mathematical equality as it looks, but rather it is the assignment of a value. It says, take the old value of i and make it the new value. So it says, update i to i minus 1, take the current value of I, subtract 1 and replace it in i. The mathematical equality is written as double equal too this is what we use in our conditions.

So, it is important to remember this that double equal to means equality as in the left hand side is equal to the right hand side, whereas the single equality in Python and many other programming languages means assign a value to a variable. So, this is the final optimization that we have of this naive algorithm which is to basically scan for common factors from the beginning to the end. So now we are doing it from the end to the beginning and keeping only the first factor that we find.

(Refer Slide Time: 10:03)

What we saw in this example is a new kind of loop. So, this new kind of loop has a special word while. And while is accompanied by condition, so long as the condition is true, we do whatever is within the body of the while. Now notice that Python uses indentation, so these statements here are offset with respect to the while. This is how Python knows that steps 1 to k belong to this while. So, these are the steps that must be repeated at the end of this thing, you come back and you check whether the condition is still true, if it is true you do it one more time and so on. So, while is useful when we do not know in advance how many times we will repeat the steps.

When we were scanning for the list of factors we knew that we would start with one and go up to the minimum of m and n. We could predict in advance that we would do precisely that many steps and so we could use this for loop, so for loop has a fixed number of repetitions. On the other hand, a while loop is typically used when you do not know in advance when you are going to stop. So in this case we going to start with the minimum of m and n, work backwards and stop as soon as we find the factor, but we have no idea in advance whether this will come early or will have to go all the way back to 1 which we know is guaranteed to be a valid factor.

One of the problems that one could face with the while is that we keep coming back and finding that the condition is true. So we never progressed out of the while. So, so long as the condition is true these steps will be executed and then you go back and do it again. If

you have not changed something which makes the condition false you will never come out.

(Refer Slide Time: 11:51)
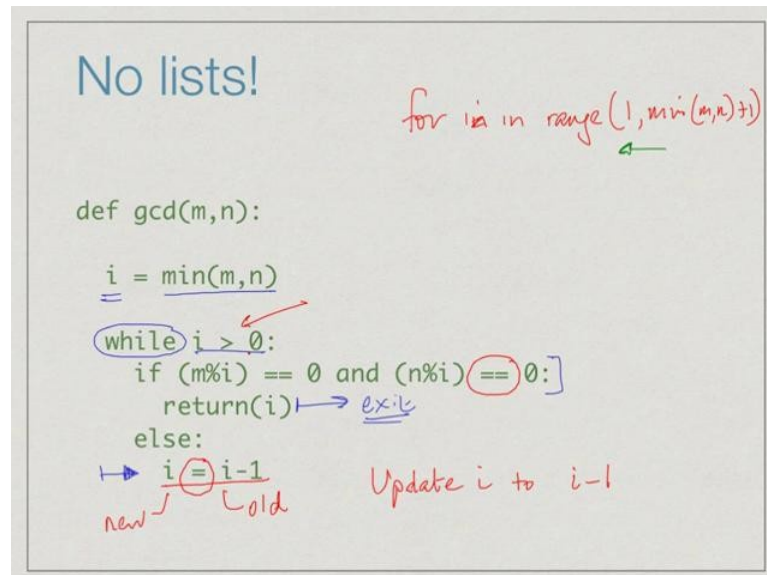
In our previous example, in order to make the condition false we need to i to become 0. So we start with the minimum of m and n. So, what we guarantee is that every time we go through this while and we do not finish what we wanted to do we reduce i by 1, and so since we start with some fixed value and we keep reducing i by 1 eventually we must reach 0.

So in general when you use a while loop you must make sure that you are making progress towards terminating the loop otherwise you have a dangerous kind of behavior called an infinite loop where the computation just keeps going on and on without returning a value and you have no idea whether it is just taking a very long time to compute the answer or whether it is never going to finish.

(Refer Slide Time: 12:35)
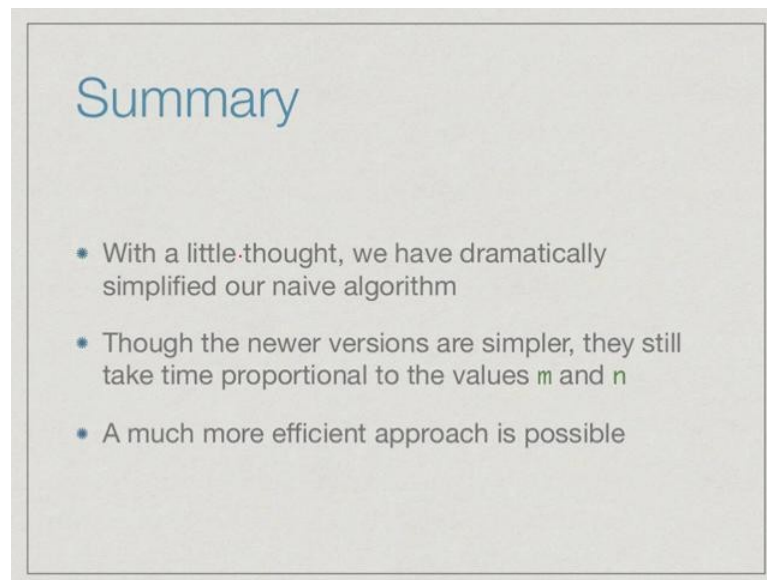
So in this lecture what we have seen is that we can start with a very naive idea which more or less implements the function as it is defined and work our ways to dramatically simplify the algorithm. Now one thing from a computational point of view, is that though the newer versions are simpler to program and therefore to understand, the amount of time they take is not very different. We are still basically running through all values in principle from 1 to the minimum of m and n. If we start from the beginning then we will run through all these values anyway because we scan all these numbers in order to find the common factors.

In the last version where we were trying to work backwards and stop at the first common factor it could still be that the two numbers have no common factor other than 1. So again, we have to run all the way back from minimum of m and n back to 1 before we find the answer. Although, the programs look simpler computationally, they are all roughly the same and that they take time proportional to the values m and n.

What we will see in the next lecture is that we can actually come up with a dramatically different way to compute gcd, which will be much more efficient.