Improving the efficiency of primality testing

FEB 2ND

Quit the loop when compositeness is detected

- As soon as we discover that n is composite, we are done and we should quit the loop and produce output.
- While this does not improve the *worst case efficiency* of the program, it does improve the typical case.
- We'll see two ways of doing this.

The break statement

- The break statements forces the program to exit out of the smallest enclosing while-loop (or for-loop).

Example:

```
n = 10
while n < 20:
        if n % 7 == 0:
                break
        n = n + 1
print n
```

Python code (Version 3)

```
import math
number = int(raw_input("Enter a positive integer: "))

factor = 2
isPrime = True
factorBound = math.sqrt(n)
while(factor <= factorBound):
        if(number % factor == 0):
                isPrime = False
                break
        factor = factor + 1

if(isPrime):
        print number, "is prime“
else:
        print number, “is composite”
```

A simple way to time Python programs

- The `time` module contains a bunch of functions that help us time code fragments.
- Calling `time.time()` returns the time elapsed (usually in seconds) since some epoch (maybe Jan 1st, 1970).
- Call `time.time()` twice, once before and once after the code fragment and take the difference.

Another approach

- We want to stay in the loop while

n <= factorBound (there are more factors to consider)

AND

isPrime == True (we have not yet found a factor)

- We can express this using the boolean operator **and** in Python.

Python code (Version 3)

```
import math
number = int(raw_input("Enter a positive integer: "))

factor = 2
isPrime = True
factorBound = math.sqrt(n)
while(factor <= factorBound) and (isPrime):
        if(number % factor == 0):
                isPrime = False
        factor = factor + 1

if(isPrime):
        print number, "is prime"
else:
        print number, "is composite"
```

Python boolean operators

- `and`, `or`, and `not` are the three Python boolean operators

- `A and B` is true only when both A *and* B are true.

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Examples: play with these

- (x <= 10) and (x > 4)
- (x < 4) and (x > 10)
- (x < 10) and True
- (x >= 0) and False

The or operator

A or B is true when A is true *or* B is true (or both).

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Examples: play with these

- (x <= 10) or (x > 4)
- (x < 4) or (x > 10)
- (x < 10) or True
- (x >= 0) or False

The `not` operator

A	not A
True	False
False	True

Examples:

- not (x < 10)
- not (x == 10)
- not (x >= -10)