Extended Introduction to Computer Science CS1001.py

Lecture 15: Integer gcd: Euclid's Algorithm; Intro to object oriented programming (OOP)

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University

Spring Semester, 2017

http://tau-cs1001-py.wikidot.com

Lecture 14 : Highlights

- Prime Numbers
  - Trial division and its computational complexity.
  - Modular exponentiation (reminder).
  - Fermat's little theorem and Randomized primality testing.

- Cryptography:
  - Secure communication over insecure communication lines
  - The discrete logarithm problem
  - One-way functions
  - Diffie-Hellman scheme for secret key exchange over insecure communication lines

Diffie-Hellman with colors

- https://www.youtube.com/watch?v=YEBfamv-_do

Lecture 15 - Plan

- Integer greatest common divisor (gcd).
  - Euclid's gcd algorithm.

- A “gentle” intro to object oriented programming (OOP): Classes and methods

Integer Greatest Common Divisor (gcd)

- Computing the integer greatest common divisor (gcd) is maybe the oldest computational problem we have a record for.

- Integer gcd (and some ramifications) is tightly related to arithmetic of large integers, e.g. computing inverses modulo a large prime, p, which are important in modern cryptography.

Integer Greatest Common Divisor

- The greatest common divisor, or gcd, of two positive integers k,l is the largest integer, g, that is a divisor of both of them. Since 1 always divides k,l , the gcd g is well defined.
- If one of these two integers is zero, we define gcd(k, 0) = k.

For example,
gcd(28,32) = 4,
gcd(276,345) = 69,
gcd(1001,973) = 7,
gcd(1002,973) = 1.

If gcd(k,l) = 1, we say that k,l are relatively prime.

Computing Greatest Common Divisor Naively

- The naive approach to computing gcd(k,l) is similar to the trial division approach: start with min(k,l), and iterate, going down, testing at each iteration if the current value divides both k and l.
  - How far do we go? Till the first divisor is found.
  - Alternatively, we could also go up, starting with 1. It won't make much of a difference in the worst case.

- What is the (worst case) running time of this naive method?

  When relatively prime, the number of trial divisions is exactly min(k,l). If the minimum is an n bit number, the running time is $O(2^n)$. Hence this method is applicable only to relatively small inputs.

Slow GCD Code

```
def slow_gcd (x,y):
    """ greatest common divisor of two integers –
        naive inefficient method """
    assert isinstance (x, int) and isinstance (y, int )
        # type checking : x and y both integers
    x,y = abs (x), abs (y) # simultaneous assignment
                           #  to x and y
         # gcd invariant to abs . Both x,y now non negative
    if x<y:
        x,y = y,x # switch x and y if x < y. Now y <= x
     for g in range (y, 0, – 1): # from y downward to 1
        if x%g == y%g == 0: # does g divide both x and y?
            return g
     return None # should never get here , 1 divides all
```

assert evaluates its boolean argument and aborts if false.

Running Slow GCD Code

```
>>> from clock import elapsed
>>> elapsed ("slow_gcd(2**50 ,2**23+1)")
2.294258
>>> elapsed ("slow_gcd(2**50 ,2**27+1)")
36.838267
```

Computing GCD - Euclid's Algorithm

- Euclid, maybe the best known early Greek mathematician, lived in Alexandria in the 3rd century BC. His book, Elements, lays the foundation to so called Euclidean geometry, including an axiomatic treatment. The book also deals with number theory and describes an efficient gcd algorithm.

(drawing from Wikipedia)

- Euclid's gcd algorithm is iterative, and is based on the following invariant

  (an invariant is a property that remains true, or a value that is unchanged, before and after applying some transformation):

Suppose $0 < l < k$, then $\gcd(k,l) = \gcd(k \bmod l, l)$.

Euclid's Algorithm (cont.)

Invariant:

Suppose $0 < l < k$, then $\gcd(k,l) = \gcd(k \bmod l, l)$.

- The algorithm replaces the pair (k,l) by (l,k (mod l)), and keeps iterating till the smaller of the two reaches zero. Then it uses the identity gcd(h, 0) = h.

- Notice that after taking the remainder, k (mod l) is strictly smaller than l. Thus one iteration of this operation reduces both numbers to be no larger than the original minimum.

Euclid's Algorithm: an example:

```
>>> k,l = 6438 , 1902
>>> k,l = l,k%l ; print (k,l)# simultaneous assignment ; then print
1902   732
>>> k,l = l,k%l ; print (k,l)
732   438
>>> k,l = l,k%l ; print (k,l)
438   294
>>> k,l = l,k%l ; print (k,l)
294   144
>>> k,l = l,k%l ; print (k,l)
144   6
>>> k,l = l,k%l ; print (k,l)
6   0
```

The gcd of 6438 and 1902 is 6.

Euclid's Algorithm (cont.)

- It can be shown (proof omitted) that two iterations of this operation make both numbers smaller than half the original maximum.
- Example:

  k0=4807526976, l0=2971215073
  k1=2971215073, l1=1836311903
  k2=1836311903, l2=1134903170
  k3=1134903170, l3=701408733
  k4=701408733, l4=433494437
  ...

- Suppose that originally k is an n bit number, namely $2^{n-1} \leq k < 2^n$. On every second iteration, the maximum number is halved. So in terms of bits, the length of the maximum becomes at least one bit shorter. Therefore, the number of iterations is at most 2n.

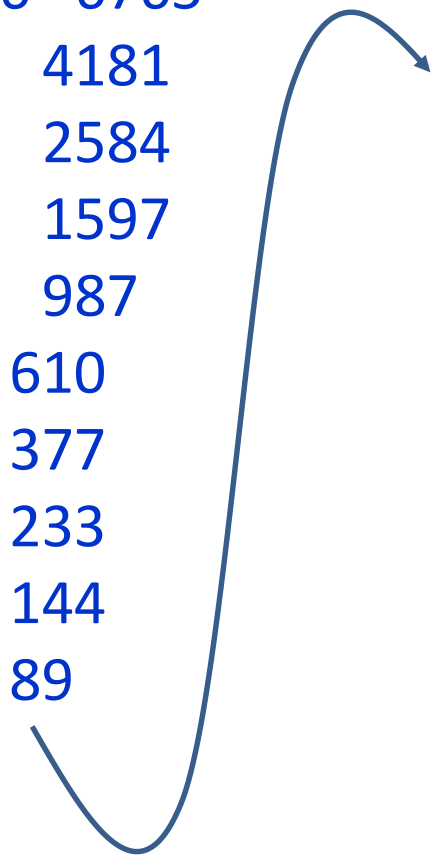
Python Code -- Euclid's Algorithm, Displaying

- The following code computes gcd(x,y) using Euclid's algorithm. In addition, it prints all intermediate pairs.

```
def display_gcd(x,y):
    """ greatest common divisor of two integers , Euclid 's algorithm .
        This function prints all intermediate results along the way . """
    assert isinstance(x, int) and isinstance(y, int)
              # type checking : x and y both integers
    x,y = abs(x), abs(y) # simultaneous assignment to x and y
                  # gcd invariant to abs . Both x,y now non - negative
    if x<y:
        x,y = y,x # switch x and y if x < y. Now y <= x
    print(x,y)
    while y >0:
        x,y = y,x%y
        print(x,y)
    return x
```

Python Code -- Euclid's Algorithm, Displaying (Execution)

```
>>> display_gcd(10946, 6765)
10946   6765
6765   4181
4181   2584
2584   1597
1597   987
987   610
610   377
377   233
233   144
144   89
89   55
55   34
34   21
21   13
13   8
8   5
5   3
3   2
2   1
1   0
1 # final outcome --   gcd(10946,6765)
```



Python Code -- Euclid's Algorithm, Displaying (Execution)

```
>>> display_gcd (6774 ,4227)
6774   4227
4227   2547
2547   1680
1680   867
867   813
813   54
54   3
3   0
3 # final outcome : gcd (6774 ,4227)=3
```

- Non trivial question: Which pairs of n bit integers, x,y, cause a worst case performance (maximal number of iterations) for Euclid's gcd?

Python Code -- Euclid's vs. Slow

- As noted before, on n bit numbers, Euclid's algorithm takes at most 2n iterations, while slow gcd takes up to 2^n iterations.

- Let us put this theoretical analysis to the ultimate test -- the test of the clock. We note that we now consider a version of Euclid's algorithm which does not display intermediate results (code omitted).

```
>>> elapsed("gcd(2**50 ,2**27+1)",number =1000000)
# million runs
26.885727999999997
>>> elapsed("slow_gcd(2**50 ,2**27+1)",number =1)
37.548531000000004
>>> slow_gcd(2**50 ,2**27+1) # sanity check
1
>>> gcd(2**50 ,2**27+1)
1
```

- Euclid's algorithm is almost 1.4 million times faster than the naïve one for this input. So theory and practice do agree here.

Euclid's gcd: Proof of Correctness Using an Invariant (for reference only)

- Suppose $0 < l \le k$.
- We first show that $\gcd(k, l) = \gcd(l, k - l)$.
  - Denote $g = \gcd(k, l)$, $h = \gcd(l, k - l)$
  - Since g divides both k and l, it also divides $k - l$.
  - Thus it divides both l and $k - l$.
  - Since h is the greatest common divisor of l and $k - l$, every divisor of l and $k - l$ divides h (think of primes' powers).
  - As g is a divisor of both, we conclude that g divides h.
  - A similar argument shows that any divisor of l and $k - l$ is also a divisor of k.
  - Thus h divides the gcd of k and l, which is g.
  - So g divides h and h divides g.
  - They are both positive, therefore g equals h.
- QED

- Note: a (different) invariant was used to prove correctness of the iterated squaring algorithm to compute a^b.

Euclid's gcd: Proof of Correctness Using an Invariant (cont.) (for reference only)

- Suppose $0 < l \leq k$. We just showed that $\gcd(k, l) = \gcd(k - l, l)$.
- If $k - l < l$, then $k \pmod l = k - l$, and we are done.
- Otherwise, $k - l \geq \gcd(k, l) = \gcd(k - l, l)$.
- Repeating the previous argument, $\gcd(k - l, l) = \gcd(k - 2 \cdot l, l)$.
- There is a unique $m \geq 1$ such that $k \pmod l = k - ml$.
- By the argument above,
  $\gcd(k, l) = \gcd(l, k - l) = \gcd(l, k - 2 \cdot l) = \gcd(l, k - 3 \cdot l) =$
  $\ldots = \gcd(l, k - m \cdot l) = \gcd(l, k \bmod l)$

QED

Proof of Correctness Using an Invariant: Conclusion (for reference only)

- In every iteration of Euclid's algorithm, we replace (k, l) by (l, k mod l), until the smaller number equals zero.
- The claim above means that at each iteration, the gcd is invariant.
- At the final stage, when we have (g, 0), we return their gcd, which equals g.
- By this invariance, g indeed equals the gcd of the original (k, l)

QED

Relative Primality and Multiplicative Inverses (for reference only)

gcd(28,31)=1

gcd(12,35)=1

gcd(527,621)=1

gcd(1002,973)=1

If gcd(k,m) = 1, we say that k,m are relatively prime.

Suppose k,m are relatively prime, and k < m.

Then there is a positive integer, a, a < m, such that

$$a \cdot k = 1 \pmod{m}$$

Such a is called a multiplicative inverse of k modulo m.

Relative Primality and Multiplicative Inverses, (cont.) (for reference only)

Suppose k,m are relatively prime, and k < m.
The there is a positive integer, a, a < m, such that

$$a \cdot k = 1 \pmod{m}$$

Such multiplicative inverse, a, can be found efficiently, using an extended version of Euclid's algorithm (details not elaborated upon in class).

$$10 \cdot 28 = 1 \pmod{31}$$

$$3 \cdot 12 = 1 \pmod{35}$$

$$218 \cdot 527 = 1 \pmod{621}$$

$$817 \cdot 937 = 1 \pmod{1002}$$

Extended GCD (for reference only)

Claim: if gcd(x, y) = g, then there are two integers a, b such that ax + by = g. For example,
gcd(1001, 973) = 7, and indeed 35 · 1001 – 36 · 973 = 7,
gcd(100567, 97328) = 79. Indeed 601 · 100567 - 621 · 97328 = 79.
gcd(10^7, 10^6 + 1) = 1, and indeed $10^5 \cdot 10^7 - 999999 \cdot (10^6 + 1) = 1$.
A simple modification of Euclid's gcd algorithm enables to compute these coefficients a, b efficiently. This algorithm is termed extended Euclidian gcd.
If p is a prime and 1 ≤ x ≤ p – 1 (this is also denoted $x \in Z_p^*$), then gcd(x, p) = 1 (why?).
Therefore there are integers a, b such that ax + bp = 1. In particular, we have ax = 1 mod p. Therefore a is the multiplicative inverse of x modulo p. (This establishes that Z_p^* with multiplication modulo p is a group, and has consequences in several cryptographic applications.)

And Now to Something Completely Different:

OOP

Object Oriented Programming (OOP)

- OOP is a major theme in programming language design, starting with Simula, a language for discrete simulation, in the 1960s. Then Smalltalk in the late 1970s (out of the legendary Xerox Palo Alto Research Center, or PARC, where many other ideas used in today's computer environment were invented). Other "OOP languages" include Eiffel, C++, Java, C#, and Scala.

- Entities in programs are modeled as objects. They represent encapsulations that have their own:
  1) attributes (also called fields), that represent their state
  2) methods, which are functions or operations that can be performed on them. Creation and manipulation of objects is done via their methods.

Object Oriented Programming (OOP), cont.

- The object oriented approach enables modular design. It facilitates software development by different teams, where each team works on its own object, and communication among objects is carried out by well defined methods' interfaces.

- Python supports object oriented style programming (maybe not up to the standards of OOP purists). We'll describe some facets, mostly via concrete examples. A more systematic study of OOP will be presented in Tochna 1, using Java.

Classes and Objects

- We already saw that classes represent data types. In addition to the classes/types that are provided by python (e.g. str, list, int), programmers can write their own classes.

- A class is a template to generate objects. The class is a part of the program text. An object is generated as an instance of a class.

- As we indicated, a class includes data attributes (fields) to store the information about the object, and methods to operate on them.

Building Class Student

```
class Student:
    def __init__(self, name, surname, ID):
        self.name = name
        self.surname = surname
        self.id = ID
        self.grades = dict()

    def __repr__(self): #must return a string
        return "<" + self.name + ", " + str(self.id) + ">"

    def update_grade(self, course, grade):
        self.grades[course] = grade


    def avg(self):
        s = sum([self.grades[course] for course in self.grades])
        return s / len(self.grades)
```

__init__and __repr__
are special standard methods,
with pre-allocated names.
More on this coming soon.

Student Class (cont.)

- The Student class has 4 fields: name, surname, id and a dictionary of grades in courses. These fields can be accessed directly, and values can be assigned to them directly.

- The methods (operations) of the class are:
  - __init__ used to create and initialize an object in this class
  - __repr__ used to describe how an object is represented (when printing such an object).
  - update_grade used to insert a new grade or update an existing one
  - avg returns the average of the student in all the courses

Student Class - Executions

```
>>> s1 = Student("Hillary", "Clinton", 123456789)
>>> s1
<Hillary, 123456789>
>>> s1.update_grade("CS1001", 91)
>>> s1.grades
{'CS1001': 91}
>>> s1.update_grade("HEDVA", 90)
>>> s1.update_grade("CS1001", 98) #she appealed
>>> s1.grades, s1.avg()
{'HEDVA': 90, 'CS1001': 98}
>>> s1.avg()
94.0
>>> s2 = Student("Angela", "Merkel", 888888888)
>>> s2.update_grade("Algebra", 95)
>>> s2.update_grade("CS1001", 100)
>>> s2, s2.grades, s2.avg()
<Angela, 888888888>,   {'CS1001': 100, 'Algebra': 95},   97.5
```

Special Methods

- There are various special methods, whose names begin and end with __ (double-underscore). These methods are invoked (called) when specific operators or expressions are used.
- Following is a partial list. The full list and more details can be found at: http://getpython3.com/diveintopython3/special-method-names.html

You Want…	So You Write…	And Python Calls…
to initialize an instance of class MyClass	x = MyClass()	x.__*init*__()
the "official" representation as a string	*print*(x)	x.__*repr*__()
addition	x + y	x.__*add*__(*y*)
subtraction	x - y	x.__*sub*__(*y*)
multiplication	x * y	x.__*mul*__(*y*)
equality	x == y	x.__*eq*__(*y*)
less than	x < y	x.__*lt*__(*y*)
for collections: to know whether it contains a specific value	k in x	x.__*contains*__(k)
for collections: to know the size	len(x)	x.__len__()
…		

Building Class Rational

Rational is another example of a class. (By convention, class names start with an upper case letter).

An object of the class Rational has to represent a rational number. This can be done in (at least) 2 ways:

1) Storing nominator (MONE') and denominator (MECHANE')
2) Storing quotient (MANA), remainder (SHE'ERIT) and denominator.

We will see both options, and ask which is better for specific operations.

The method that is used to create and initialize an object has a special name, __init__ .

Initializing Class Rational

```
from gcd import *

class Rational():

    def __init__(self, n, d):
        assert isinstance(n,int)  and isinstance(d,int)
        gcd_nd = gcd(n,d)
        self.n = n//gcd_nd #nominator
        self.d = d//gcd_nd #denominator
```

assert evaluates its boolean argument and aborts if false.

Initializing Class Rational

```
>>> r1 = Rational(3,5) # calls __init__ of class Rational
>>> r1.n      # accessing the field n of the object r1
3
>>> r1.d
5
>>> r2 = Rational(3,6)
>>> r2.n
1
>>> r2.d
2
>>> r2.y
Traceback (most recent call last):
File "<pyshell#34>", line 1, in <module>
r1.y
AttributeError: 'Rational' object has no attribute 'y'
```

Presenting Class Rational

__repr__ is another special method of classes (starts and ends with two _ symbols). It is used to describe how an instance of the class is represented (when printing such an object).

Special methods have specific roles in any class in which they appear. We will see how they are called.

```
def __repr__(self):
  if self.d == 1:
    return "<Rational " + str(self.n) + ">"
  else:
    return "<Rational " + str(self.n) + "/" + str(self.d) + ">"
```

Presenting Class Rational

```
>>> r1 = Rational(3,6)
>>> r1        # calls __repr__ of class Rational on the object r1
<Rational 1/2>
>>> r2 = Rational(12,6)
>>> r2        # calls __repr__ of class Rational on the object r2
<Rational 2>
```

Rational Class, Additional Methods (cont)

```
def is_int(self):
    return self.d == 1

def floor(self):
    return self.n // self.d
```

```
>>> r1 = Rational(3,5)
>>> r2 = Rational(12,6)
>>> r1.is_int()
False
>>> r2.is_int()
True
>>> r1.floor()
0
>>> r2.floor()
2
```

Calling methods

- We have seen that we call a function by its full name, preceded by an object of the appropriate class, for example r1.floor().

- But we can also call it using the name of the class (rather than a specific object). In this case the first parameter will be the calling object:

```
>>> r1 = Rational(3,5)
>>> Rational.is_int(r1)
False
>>> Rational.floor(r1)
0
```

Rational Class, Defining Equality

```
>>> r1 = Rational(3,5)
>>> r2 = Rational(6,10)
>>> r1==r2
False #Hah?

>>> r3 = Rational(3,5)
>>> r1==r3
False #WHAT??
```

Rational Class, Defining Equality (2)

- Unless otherwise defined, Python compares objects by their memory address.
- __eq__ is a special method that determines when two objects (in this case) lines are equal.

```
def __eq__(self, other):
    return self.n == other.n and self.d == other.d
```

```
>>> r1 = Rational(3,5)
>>> r2 = Rational(6,10)
>>> r1==r2   # __eq__ is called , same as r1.__eq__(r2)
True #☺
```

Rational Class, Defining Equality (3)

```
>>> r1 = Rational(6,3)
>>> r1==2
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    r1==2
  File "D...", line 27, in __eq__
    return self.n == other.n and self.d == other.d
AttributeError: 'int' object has no attribute 'n'
```

This should not surprise you.

But why not allow comparing a Rational type object to an int?

Rational Class, Defining Equality (4)

- Why not allow comparing a Rational type object to an int?

```
def __eq__(self, other):
        assert isinstance(other, (Rational,int))
        if isinstance(other, Rational):
                return self.n == other.n and self.d == other.d
        else:
                return self.n == other and self.d == 1
```

type safety assertion

```
>>> r1 = Rational(6,3)
>>> r1==2
True #☺
```

Alternative design of class Rational

Perhaps it is better to store the quotient, remainder and denominator?

```
class Rational2():

    def __init__(self, n, d):
        assert isinstance(n,int)  and isinstance(d,int)
        gcd_nd = gcd(n,d)
        n = n//gcd_nd
        d = d//gcd_nd
        self.q = n//d #quotient (MANA)
        self.r = n%d  #remainder (SHE'ERIT)
        self.d = d      #denominator (MECHANE')
```

Alternative design of class Rational (2)

- The methods should change accordingly, while keeping the interface intact:

```
    ...
def __repr__(self):
    if self.r == 0:
        return "<Rational " + str(self.q) + ">"
    else:
        n = self.q * self.d + self.r
        return "<Rational " + str(n) + "/" + str(self.d) + ">"

def is_int(self):
    return self.r == 0

def floor(self):
    return self.q
```

Compare to:

```
def floor(self):
    return self.n // self.d
```

Alternative design of class Rational (3)

```
class Rational2():

    …
  def __eq__(self,other):
      assert isinstance(other, (Rational2,int))
      if isinstance(other,Rational2):
         return self.q == other.q and \
                self.r == other.r and \
                self.d == other.d
      else:
         return self.q == other and self.r == 0
```

Note on fields and parameters

- The first parameter of every method represents the current object (an object of the class which includes the method). By convention, we use the name self for this parameter.

- So the variable self.n is the field named n in the (current) object. Unlike other languages (eg. Java), we do not explicitly declare the names of the fields of the class we are defining. They exist because they are mentioned – initialized in the __init__ method. other.n is the nominator of the object pointed to by other.

Privacy

```
>>> r1 = Rational(3,5)
>>> r1.n = 10        # do we want to allow this?
>>> r1
<Rational 10/5>     # the object has changed
```

Allowing access to fields is a source for trouble. An the following is even more scary:

```
>>> r1.y
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    r1.y
AttributeError: 'Rational' object has no attribute 'y'
>>> r1.y = 77
>>> r1
<Rational 10/5>
>>> r1.y
77
```

Information hiding (for reference only)

One of the principles of OOP is information hiding: The designer of a class should be able to decide what information is known outside the class, and what is not. In most OOP languages this is achieved by declaring fields and methods as either public or private.

In python, a field whose name starts with two _ symbols, will be private. It will be known inside the class, but not outside.

A private field cannot be written (assigned) outside the class, and its value cannot be read (inspected), because its name is not known. The class then provides methods to access and modify the state of the object in the "legal" way.

OOP and python

- Python provides the basic ingredients for OOP, including inheritance (that we will not discuss).
- However, we do not have the full safety that strict OOP languages have . “Private” fields are accessible with mangled names, a client may add a field to an object, etc. In short, there is no way to enforce data hiding in python, it is all based on convention.
- The language puts more emphasis on flexibility.
- In this course we will not use private fields to simplify the code, rather than adhere to OOP. This is the common style in python.
- The course Software 1 (in Java) places OOP at the center.

Designing classes in OOP

The recommended way to design a class is to first decide what operations (methods) the class should support. This would be the API (or contract) between the class designer and the clients.

We often distinguish between:

- Queries – return a value, do not change the state
- Commands – change the state, do not return a value
- Constructors – create the object and initialize it

Then decide how to represent the state of objects (which fields), and make the fields private.

Then implement (write code for) the methods.

This way we can later change the representation (eg. change from Cartesian to Polar representation of points), while the client code is unchanged.