Algorithms, programming

- Algorithm: how to systematically perform a task
- Write down as a sequence of steps
  - "Recipe", or program
- Programming language describes the steps
  - What is a step? Degrees of detail
  - "Arrange the chairs" vs "Make 8 rows with 10 chairs in each row"

Our focus

- Algorithms that manipulate information
  - Compute numerical functions — $f(x,y) = x^y$
  - Reorganize data — arrange in ascending order
  - Optimization — find the shortest route
  - And more ...
    - Solve Sudoku, play chess, correct spelling ...

Greatest common divisor

- `gcd(m,n)`
  - Largest `k` such that `k` divides `m` and `k` divides `n`
  - `gcd(8,12) = 4`
  - `gcd(18,25) = 1`
- 1 divides every number
- At least one common divisor for every `m, n`

Computing gcd(m,n)

- List out factors of m
- List out factors of n
- Report the largest number that appears on both lists
- Is this a valid algorithm?
  - Finite presentation of the "recipe"
  - Terminates after a finite number of steps

Computing gcd(m,n)

- Factors of m must be between 1 and m
  - Test each number in this range
  - If it divides m without a remainder, add it to list of factors
- Example: gcd(14,63)
- Factors of 14

1 2 3 4 5 6 7 8 9 10 11 12 13 14

Computing gcd(14,63)

- Factors of 14 1 2 7 14
- Factors of 63 1 3 7 9 21 63
- Construct list of common factors
  - For each factor of 14, check if it is a factor of 63

1 7

- Return largest factor in this list: 7

gcd

Computing gcd(14,63)

- Factors of 14 1 2 7 14
- Factors of 63 1 3 7 9 21 63
- Construct list of common factors
  - For each factor of 14, check if it is a factor of 63

1 7

- Return largest factor in this list: 7

gcd

An algorithm for gcd(m,n)

- Use fm, fn for list of factors of m, n, respectively
- For each i from 1 to m, add i to fm if i divides m
- For each j from 1 to n, add j to fn if j divides n
- Use cf for list of common factors
- For each f in fm, add f to cf if f also appears in fn

[1,2,7,14] [1,3,7,9,21,63]

An algorithm for `gcd(m,n)`

- Use `fm`, `fn` for list of factors of `m`, `n`, respectively
- For each `i` from 1 to `m`, add `i` to `fm` if `i` divides `m`
- For each `j` from 1 to `n`, add `j` to `fn` if `j` divides `n`
- Use `cf` for list of common factors
- For each `f` in `fm`, add `f` to `cf` if `f` also appears in `fn`
- Return largest (rightmost) value in `cf`

Our first Python program

```
def gcd(m,n):
  fm = []
  for i in range(1,m+1):
    if (m%i) == 0:
      fm.append(i)

  fn = []
  for j in range(1,n+1):
    if (n%j) == 0:
      fn.append(j)

  cf = []
  for f in fm:
    if f in fn:
      cf.append(f)

  return(cf[-1])
```

Some points to note

- Use names to remember intermediate values
  - `m, n, fm, fn, cf, i, j, f`
- Values can be single items or collections
  - `m, n, i, j, f` are single numbers
  - `fm, fn, cf` are lists of numbers
- Assign values to names
  - Explicitly, `fn = []`, and implicitly, `for f in cf:`
- Update them, `fn.append(i)`

Some points to note …

- Program is a sequence of steps
- Some steps are repeated
  - Do the same thing for each item in a list
- Some steps are executed conditionally
  - Do something if a value meets some requirement

```
if (m%i) == 0
```

An algorithm for gcd(m,n)

- Use `fm`, `fn` for list of factors of `m`, `n`, respectively
- For each `i` from `1` to `m`, add `i` to `fm` if `i` divides `m`
- For each `j` from `1` to `n`, add `j` to `fn` if `j` divides `n`
- Use `cf` for list of common factors
- For each `f` in `fm`, add `f` to `cf` if `f` also appears in `fn`
- Return largest (rightmost) value in `cf`

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`

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`

A shorter Python program

```
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])
```

A shorter Python program

```
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])
```

Do we need lists at all?

- We only need the largest common factor
- 1 will always be a common factor
- Each time we find a larger common factor, discard the previous one
- Remember the largest common factor seen so far and return it
  - `mrcf` — most recent common factor

No lists!

```
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)
```

Scan backwards?

- 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!

No lists!

for i in range(1, min(m,n)+1)

```
def gcd(m,n):

    i = min(m,n)

    while i > 0:
        if (m%i) == 0 and (n%i) == 0:
            return(i)
        else:
            i = i-1
```

A new kind of repetition

```
while condition:
    step 1
    step 2
    . . .
    step k
```

- Don't know in advance how many times we will repeat the steps
- Should be careful to ensure the loop terminates—eventually the condition should become false!

Summary

- With a little thought, we have dramatically simplified our naive algorithm
- Though the newer versions are simpler, they still take time proportional to the values `m` and `n`
- A much more efficient approach is possible

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!

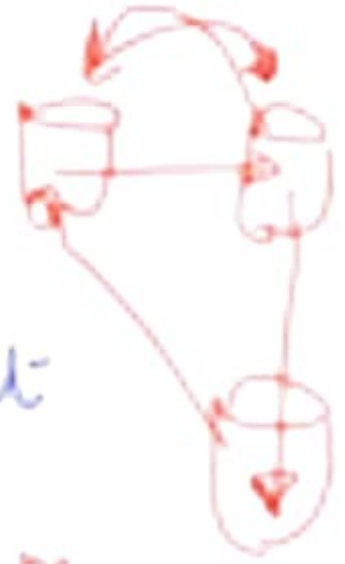
Euclid's algorithm

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

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))
```

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))
```

Comment

m → m
n → n

m = 97
n = 2

gcd(n, m-n)

diff = 95

Recursion

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)
```

Euclid's algorithm, again

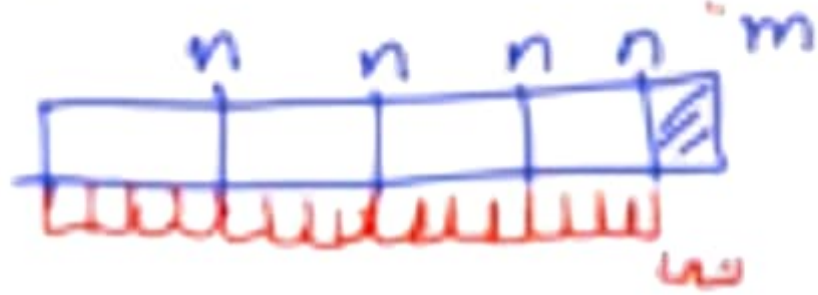
```
def gcd(m,n):

    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)
```

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 $m = ad$, $n = bd$

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



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)
```

Efficiency

- 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

gcd (101, 2)

↳ r = 1

Efficiency

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

Installing Python

- Python is available on all platforms: Linux, MacOS and Windows
- Two main flavours of Python
  - Python 2.7
  - Python 3+ (currently 3.5.x)
- We will work with Python 3+

Python 2.7 vs Python 3

- Python 2.7 is a "static" older version
  - Many libraries for scientific and statistical computing are still in Python 2.7, hence still "alive"
- Python 3 is mostly identical to Python 2.7
  - Designed to better incorporate new features
  - Will highlight some differences as we go along

Downloading Python 3.5

- Any Python 3 version should be fine, but the latest is 3.5.x
- On Linux, it should normally be installed by default, else use the package manager
- For MacOS and Windows, download and install from https://www.python.org/downloads/release/python-350/
- If you have problems installing Python, search online or ask someone!

Downloading Python 3.5

- Any Python 3 version should be fine, but the latest is 3.5.x
- On Linux, it should normally be installed by default, else use the package manager
- For MacOS and Windows, download and install from https://www.python.org/downloads/release/python-350/
- If you have problems installing Python, search online or ask someone!

Interpreters vs compilers

- Programming languages are "high level", for humans to understand
- Computers need "lower level" instructions
- Compiler: Translates high level programming language to machine level instructions, generates "executable" code
- Interpreter: Itself a program that runs and directly "understands" high level programming language

Interpreters vs compilers

- Programming languages are "high level", for humans to understand
- Computers need "lower level" instructions
- Compiler: Translates high level programming language to machine level instructions, generates "executable" code
- Interpreter: Itself a program that runs and directly "understands" high level programming language

Python interpreter

- Python is basically an interpreted language
  - Load the Python interpreter
  - Send Python commands to the interpreter to be executed
  - Easy to interactively explore language features
  - Can load complex programs from files
    - `>>> from filename import *`

```
$ ls
__pycache__/    [illegible].py    [illegible].py    [illegible].py    [illegible].py
[illegible].py    [illegible].py    [illegible].py    [illegible].py
$ python3.5
Python 3.5.2 (default, Jun 27 2016, [illegible])
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 - 5
>>> 1
>>> 5

>>> 1+5
>>> 6

>>> 2*3 + 5
>>> 11

>>> def twice(x):
>>> 
```

```
$ ls
__pycache__/   protocol1x0.py   protocol1x0a.py   protocol1x0b.py   protocol1x0c.py
profiler.py    protocol.py      protocolv4.py     protocol.py
protocol.5
Python 3.5.2 (default, Jun 27 2016, 00:00:32)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 = 0
>>> 1
>>> 5

>>> 1+1
>>> 0

>>> 2*3 + 5
>>> 11

>>> def value():
...     y = 2*1
... 
```

```
$
$
$
$ ls
__pycache__/    gcdeuclid[illegible].py   gcdeuclid[illegible].py  gcdeuclid[illegible].py   gcdeuclid[illegible].py
gcdfour.py      gcdone.py        gcdthree.py     gcdtwo.py
$ emacs gcdone.py
$ python3.5
Python 3.5.2 (default, Jun 27 2016, [illegible])
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from gcdone import *
>>> gcd(14,63)
>>> 7

>>> gcd([illegible],[illegible])
>>> 1

>>> gcd(
```

Installing Python

- Python is available on all platforms: Linux, MacOS and Windows
- Two main flavours of Python
  - Python 2.7
  - Python 3+ (currently 3.5.x)
- We will work with Python 3+

Python 2.7 vs Python 3

- Python 2.7 is a "static" older version
  - Many libraries for scientific and statistical computing are still in Python 2.7, hence still "alive"
- Python 3 is mostly identical to Python 2.7
  - Designed to better incorporate new features
  - Will highlight some differences as we go along

Downloading Python 3.5

- Any Python 3 version should be fine, but the latest is 3.5.x
- On Linux, it should normally be installed by default, else use the package manager
- For MacOS and Windows, download and install from https://www.python.org/downloads/release/python-350/
- If you have problems installing Python, search online or ask someone!

Interpreters vs compilers

- Programming languages are "high level", for humans to understand
- Computers need "lower level" instructions
- Compiler: Translates high level programming language to machine level instructions, generates "executable" code
- Interpreter: Itself a program that runs and directly "understands" high level programming language

Python interpreter

- Python is basically an interpreted language
  - Load the Python interpreter
  - Send Python commands to the interpreter to be executed
  - Easy to interactively explore language features
  - Can load complex programs from files
    - `>>> from filename import *`

```
$ ls
__pycache__/   [illegible].py   [illegible].py   [illegible].py   [illegible].py
[illegible].py   [illegible].py   [illegible].py   [illegible].py
$ python3.5
Python 3.5.2 (default, Jun 27 2016, [illegible])
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 - 5
>>> 1
>>> 5

>>> 1+5
>>> 6

>>> 2*3 + 5
>>> 11

>>> def twice(x):
>>> 
```

```
$ ls
__pycache__/    gcdeuclid1.py   gcdeuclid1a.py  gcdeuclid2.py   gcdeuclid2a.py
gcdfour.py      gcdone.py       gcdthree.py     gcdtwo.py
$ python3.5
Python 3.5.2 (default, Jun 27 2016, 08:10:38)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 5
>>> 1
>>> 5

>>> 1+5
>>> 6

>>> 2*3 + 5
>>> 11

>>> def twice(x):
>>>   y = 2*x
...   return(y)
...
...
>>> twice(7)
>>> 14

>>> twice(512)
>>> 1024

>>> 
```

```
def gcd(m,n):

    fm = []
    for i in range(1,m+1):
        if (m%i) == 0:
            fm.append(i)

    fn = []
    for j in range(1,n+1):
        if (n%j) == 0:
            fn.append(j)

    cf = []
    for f in fm:
        if f in fn:
            cf.append(f)

    return(cf[-1])
```

```
__pycache__   gcdeuclid1.py   gcdeuclid1a.py   gcdeuclid1b.py   gcdeuclid2.py   gcdeuclid2a.py
four.py       gcdone.py       gcdthree.py      gcdtwo.py
macs gcdone.py
ython3.5
hon 3.5.2 (default, Jun 27 2016, 03:10:38)
 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
 "help", "copyright", "credits" or "license" for more information.
from gcdone import *
gcd(14,63)
7

gcd(999999,100000)


cd(9999999,1000000)


cd(9999999,1000000)
```

```
def gcd(m,n):

    if m < n:
        (m,n) = (n,m)

    if (m % n) == 0:
        return(n)
    else:
        return (gcd(n,m%n))
```

U-:----F1 gcdeuclid2.py   All L1   (Python)

```
$
$
$ ls
__pycache__/    gcdeuclid1.py   gcdeuclid1o.py  gcdeuclid2.py   gcdeuclid2o.py
gcdfour.py      gcdone.py       gcdthree.py     gcdtwo.py
$ emacs gcdeuclid2.py
$ python3.5
Python 3.5.2 (default, Jun 27 2016, 08:10:36)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from gcdeuclid1o import *
>>> gcd(9999999,1000000)
>>> 1

>>> gcd(9999999999,1000000000)
>>> 1

>>> bleep(7)
>>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bleep' is not defined

>>> 7 <> 5
>>>   File "<stdin>", line 1
    7 <> 5
      ^
SyntaxError: invalid syntax
```

Some resources

- The online Python tutorial is a good place to start: https://docs.python.org/3/tutorial/index.html
- Here are some books, again available online:
  - *Dive into Python 3*, Mark Pilgrim
    http://www.diveintopython3.net/
  - *Think Python*, 2nd Edition, Allen B. Downey
    http://greenteapress.com/wp/think-python-2e/

Learning programming

- Programming cannot be learnt theoretically
- Must write and execute your code to fully appreciate the subject
- Python syntax is light and is relatively easy to learn
- Go for it!