Comparing DNA via Longest Common Subsequence (LCS)

```
AGGACAT
ATTACGAT

>>> LCS(“AGGACAT”, “ATTACGAT”)
5
>>> LCS(“spam”, “sam!”)
3
>>> LCS(“spam”, “xsam”)
3
```

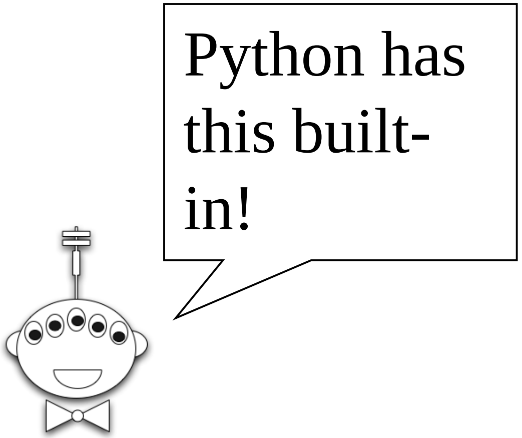
Simpler examples first, please

Investigating code

```
def mystery(n):
  return m_help(n, 0)

def m_help(n, r):
  if n == 0:
    return r
  return m_help(n // 10, r * 10 + n % 10)

print(mystery(452))   # TRACE THIS
```

Computing the length of a list

```
>>> len([1, 42, "spam"])
3
>>> len([1, [2, [3, 4]]])
```



Python has this built-in!

```
def len(lst):
  """returns the length of lst"""
```

Hint: view the list recursively, as `[first] + rest`

Reversing a list

```
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]

def reverse(lst):
  """returns a new list that is the
     reverse of the input list"""
```

member

```
>>> member(42, [1, 3, 5, 42, 7])
True
>>> member(42, [“spam”, “is”, “yummy”, 2])
False

Hint:  view L as L[0] + L[1:]
def member(x, L):
```

Worksheet please!

member

```
>>> member(42, [1, 3, 5, 42, 7])
True
>>> member(42, [“spam”, “is”, “yummy”])
False


def member(x, L):
```

Thinking about L recursively:
We can check whether it’s [ ].
We can refer to L[0] (the first element).
We can refer to L[1:] (all but first element).
That’s all!

Writing `map` and `reduce`

```
>>> map(dbl, [0, 1, 2, 3])
[0, 2, 4, 6]
```

The shortest possible list has length 0

```
def map(f, L):
```

```
>>> reduce(add, [1, 2, 3])
6
```

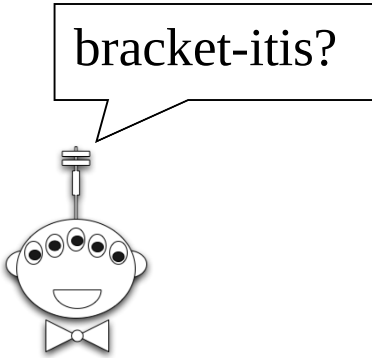
The shortest allowed list has length 1

```
def reduce(g, L):
```

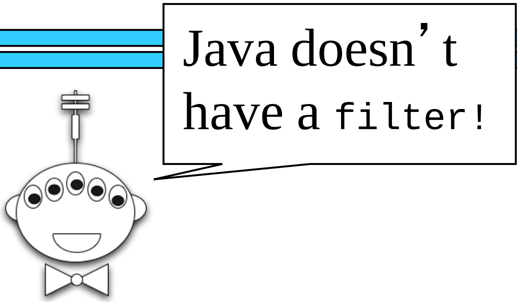
Writing map

```
>>> map(dbl, [0, 1, 2, 3])
[0, 2, 4, 6]
```

```
def map(f, L):
  # Recursive view of L is L[0] and L[1:] only

  if L==[]:
    return []
  else:
    return [f(L[0])] + map(f, L[1:])
```



filter

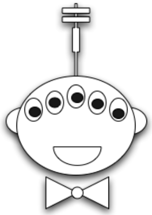


```
def even(x):
   ’’’returns True iff x is even’’’
   return x % 2 == 0

>>> filter(even, range(100))
[0, 2, 4, 6, …, 98]
```

A function that returns either `True` or `False` Is called a *predicate*

filter

```
def short (List):
   '''returns True iff List has len <= 2'''
   return len(List) <= 2

>>> filter(short, [ ["spam", "yum"], [42], [1, 2, 3]])
```



`filter` can be written from scratch using recursion.

See this week's lab.

Functions are data

```
def divides(n):
  def div(k):
      return n % k == 0
  return div

>>> f = divides(10)
>>> f
<function f at 0x661f0>
>>> f(2)

>>> listOfFunctions = [divides(10), divides(20)]
>>> listOfFunctions[0](2)
```

*my*thon (a “pure” functional language)

```
myNumber = 42
myFood = “spam”

def dbl(x):
  return 2 * x

dbl =  (x):
  return 2 * x

dbl = lambda(x):
  return 2 * x

>>> dbl(21)
42
```

Alonzo Church
1903 - 1995

Python (an "impure" functional language)

Alonzo Church
1903 - 1995

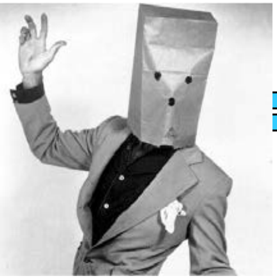
```
myNumber = 42
myFood = "spam"

def dbl(x):
  return 2 * x

dbl =  ~~lambda (x):~~
~~   return 2 * x~~

dbl = lambda x: 2 * x
```

One line
no parentheses on the input variable
`return` is implicit

```
>>> dbl(21)
42
```

lambda

aka “anonymous functions”



```
>>> filter(lambda x: x%2 == 0, range(100))

>>> filter(lambda List: len(List) <= 2,
        [[“spam”, “yum”], [42], [1, 2, 3]])
```

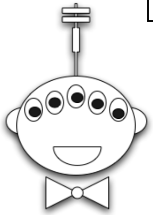
lambda

```
even = lambda X: X%2 == 0

def even(x):
    ’’’returns True iff x is even’’’
    return x % 2 == 0
```

```
short = lambda List: len(List) <= 2

def short(List):
    ’’’returns True iff List has len <= 2’’’
    return len(List) <= 2
```

lambda

```
def mystery(item, L):
  NewL = map(lambda X : X == item, L)
  return sum(NewL) > 0
```



This is exploiting the fact that `True==1` and `False==0`.

Another Prime Example

Write a function called `prime(n)` that returns `True` if `n` is prime and `False` otherwise by testing all possible divisors from 2 to `n-1` (or sqrt of n)

```
def prime(n):
   possibleDivisors = range(2, n)
   divisors = filter(                                      )
   return ???
```

A Prime Example

Write a function called `prime(n)` that returns `True` if `n` is prime and `False` otherwise by testing all possible divisors from 2 to `n-1` (or sqrt of n)

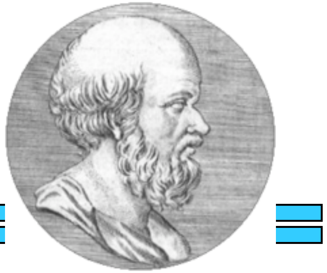
```
def prime(n):
   possibleDivisors = range(2, n)
   divisors = filter(lambda X: n % X == 0, possibleDivisors)
   return len(divisors) == 0
```

Alternatively, which of these works?…

```
def divides(X):
     return n % X == 0

def prime(n):
  possibleDivisors = range(2, n)
  divisors = filter(divides,
                    possibleDivisors)
  return len(divisors) == 0
```

```
def prime(n):

  def divides(X):
     return n % X == 0

  possibleDivisors = range(2, n)
  divisors = filter(divides,
                    possibleDivisors)
  return len(divisors) == 0
```

Listing Primes…



Eratosthenes
200 BCE

Objective: Find all primes less than or equal to some given n.

Approach 1: Test 2, 3, …, n for primality

Approach 2: The Sieve of Eratosthenes…

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,…

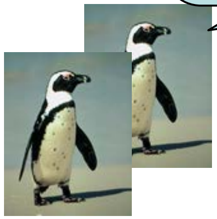
2, **3**, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,…

2, **3**, 4, **5**, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,…

Nifty Sifty...

```
>>> sieve([2, 3, 4, 5, 6, 7, 8, 9])
[2, 3, 5, 7]

def primes(n):
   ’’’returns the list of primes <= n’’’
   return sieve(range(2, n+1))

def sieve(L):
   if L == []: return []
   else: return ???
```

This is a fun challenge. Try it in your notes!



Nifty Sifty…

```
>>> sieve([2, 3, 4, 5, 6, 8, 9])
[2, 3, 5, 7]

def primes(n):
   ’’’returns the list of primes <= n’’’
   return sieve(range(2, n+1))

def sieve(L):
   if L == []: return []
   else: return [L[0]] + ???
```

Filter it!

```
>>> sieve([2, 3, 4, 5, 6, 7, 8, 9])
[2, 3, 5, 7]

def sieve(L):
   if L == []: return []
   else: return [L[0]] +
       filter(lambda X: X % L[0] != 0, L[1:])
```

This is a good start, but we’re not quite done!

Is this sweet or what!?

```
>>> sieve([2, 3, 4, 5, 6])
[2, 3, 5]

def sieve(L):
   if L == []: return []
   else: return [L[0]] +
     sieve(filter(lambda X: X % L[0] != 0, L[1:]))
```

```
def primes(n):
   return sieve(range(2, n+1))
```

demo!

Power Set!

```
>>> powerset([1, 2])
[[], [2], [1], [1, 2]]

>>> powerset([1, 2, 3])
[[], [3], [2], [2, 3], [1], [1, 3],
  [1, 2], [1, 2, 3]]
>>> powerset([1])


>>> powerset([])
```

This really demonstrates the power of functional programming!

The order in which the subsets are presented is unimportant but within each subset, the order should be consistent with the input set. So maybe it should be called "powerlist".

Power Set!

```
def powerset(L):
```

In your notes…

The Packing Problem

```
>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
```

Two inputs means two base cases!

The Packing Problem

```
>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
   if target == 0: return True
```

Two inputs means two base cases!

The Packing Problem

```
>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False
```

```
def subset(target, L):
   if target == 0: return True
   elif L == []: return False
```

What if we switched the order of these?

The Packing Problem

```
>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False
```

```
def subset(target, L):
   if target == 0: return True
   elif L == []: return False
   elif L[0] > target: return subset(target, L[1:])
```

The Packing Problem

```
>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
   if target == 0: return True
   elif L == []: return False
   elif L[0] > target: return subset(target, L[1:])
   else:
      useIt = subset(target – L[0], L[1:])
      loseIt = subset(target, L[1:])
```

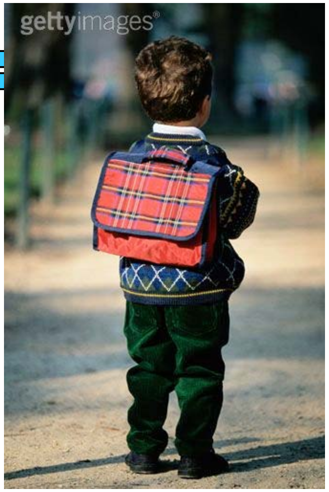
The Packing Problem

```
>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False
```

```
def subset(target, L):
   if target == 0: return True
   elif L == []: return False
   elif L[0] > target: return subset(target, L[1:])
   else:
      useIt = subset(target – L[0], L[1:])
      loseIt = subset(target, L[1:])
      return useIt or loseIt
```

The Knapsack Problem…

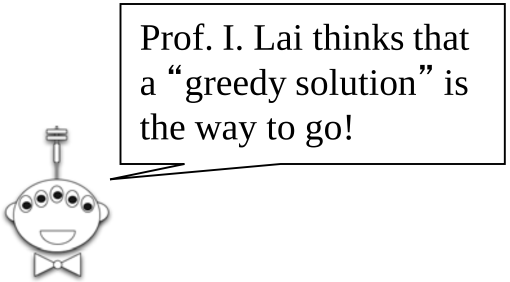
Kingdom of Shmorbodia



Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

```
>>> knapsack(7, [ [2, 100], [3, 112], [4, 125] ])
237
```

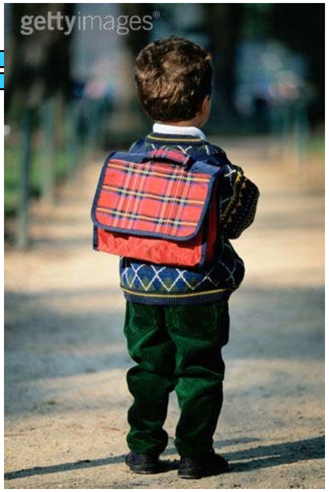

Prof. I. Lai thinks that a "greedy solution" is the way to go!

The Knapsack Revisited…

Kingdom of Shmorbodia

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?



```
>>> knapsack(7, [ [2, 100], [3, 112], [4, 125] ])
[237, [ [3, 112], [4, 125] ] ]
```

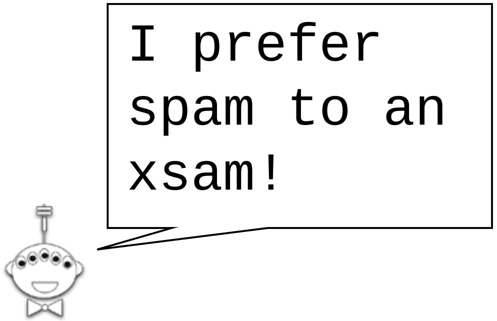
Comparing DNA via Longest Common Subsequence (LCS)

AGGACAT
ATTACGAT

```
>>> LCS(“AGGACAT”, “ATTACGAT”)
5
>>> LCS(“can”, “man!”)
2
```

Comparing DNA via Longest Common Subsequence (LCS)

```
AGGACAT
ATTACGAT

>>> LCS(“AGGACAT”, “ATTACGAT”)
5
>>> LCS(“spam”, “sam!”)
3
>>> LCS(“spam”, “xsam”)
3
```



Recursive Approach…

```
def LCS(S1, S2):
    if BASE CASE
    else:
```

LCS(“spam”, “sam!”)

Try this in your notes!

Solution follows

Recursive Approach…

```
def LCS(S1, S2):
    if S1 == "" or S2 == "": return 0
    else:
```

LCS(“spam”, “sam!”)

Recursive Approach...

```
def LCS(S1, S2):
    if S1 == "" or S2 == "": return 0
    else:
        if S1[0] == S2[0]:  # DO THE FIRST SYMBOLS MATCH?
            return 1 + ???
        else:
```

LCS(“spam”, “sam!”)

Recursive Approach...

```
def LCS(S1, S2):
    if S1 == "" or S2 == "": return 0
    else:
        if S1[0] == S2[0]:  # DO THE FIRST SYMBOLS MATCH?
            return 1 + LCS(S1[1:], S2[1:])
        else:
```

LCS(“spam”, “sam!”)

Recursive Approach...

```
def LCS(S1, S2):
    if S1 == "" or S2 == "": return 0
    else:
        if S1[0] == S2[0]:  # DO THE FIRST SYMBOLS MATCH?
            return 1 + LCS(S1[1:], S2[1:])
        else:
            return max(LCS(S1, S2[1:]), LCS(S1[1:], S2))
```

LCS(“spam”, “sam!”)