$O(n^2)$ sorting algorithms

- Selection sort and insertion sort are both $O(n^2)$
- $O(n^2)$ sorting is infeasible for n over 5000

A different strategy?

- Divide array in two equal parts
- Separately sort left and right half
- Combine the two sorted halves to get the full array sorted

Combining sorted lists

- Given two sorted lists A and B, combine into a sorted list C
  - Compare first element of A and B
  - Move it into C
  - Repeat until all elements in A and B are over
- Merging A and B

Merging two sorted lists

32 74 89

21 55 64

Merging two sorted lists

~~32~~ ~~74~~ 89

~~21~~ ~~55~~ ~~64~~

21 32 55 64 74

Merge Sort

43 32 22 78 63 57 91 13

43 32 22 78 | 63 57 91 13

43 32 | 22 78 | 63 57 | 91 13

43 | 32 | 22 | 78 | 63 | 57 | 91 | 13

Divide and conquer

- Break up problem into disjoint parts
- Solve each part separately
- Combine the solutions efficiently

Merging sorted lists

Combine two sorted lists A and B into C

- If A is empty, copy B into C
- If B is empty, copy A into C
- Otherwise, compare first element of A and B and move the smaller of the two into C
- Repeat until all elements in A and B have been moved

Merging

```
def merge(A,B): # Merge A[0:m],B[0:n]
  (C,m,n) = ([],len(A),len(B))
  (i,j) = (0,0) # Current positions in A,B
  while i+j < m+n: # i+j is number of elements merged so far
    if i == m:  # Case 1: A is empty
      C.append(B[j])
      j = j+1
    elif j == n: # Case 2: B is empty
      C.append(A[i])
      i = i+1
    elif A[i] <= B[j]: # Case 3: Head of A is smaller
      C.append(A[i])
      i = i+1
    elif A[i] > B[j]: # Case 4: Head of B is smaller
      C.append(B[j])
      j = j+1
  return(C)
```

Divide and conquer

- Break up problem into disjoint parts
- Solve each part separately
- Combine the solutions efficiently

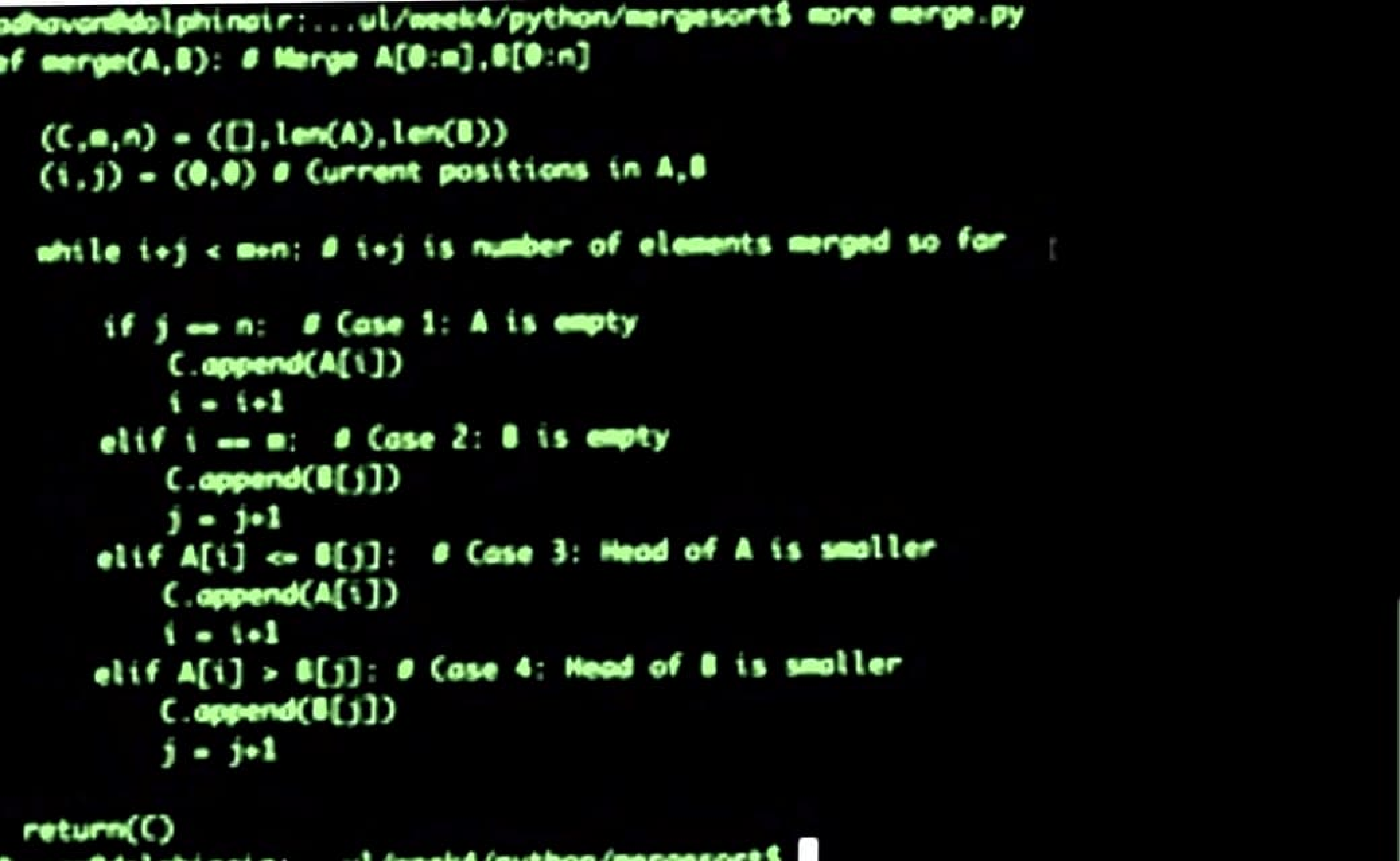
Merging sorted lists

Combine two sorted lists A and B into C

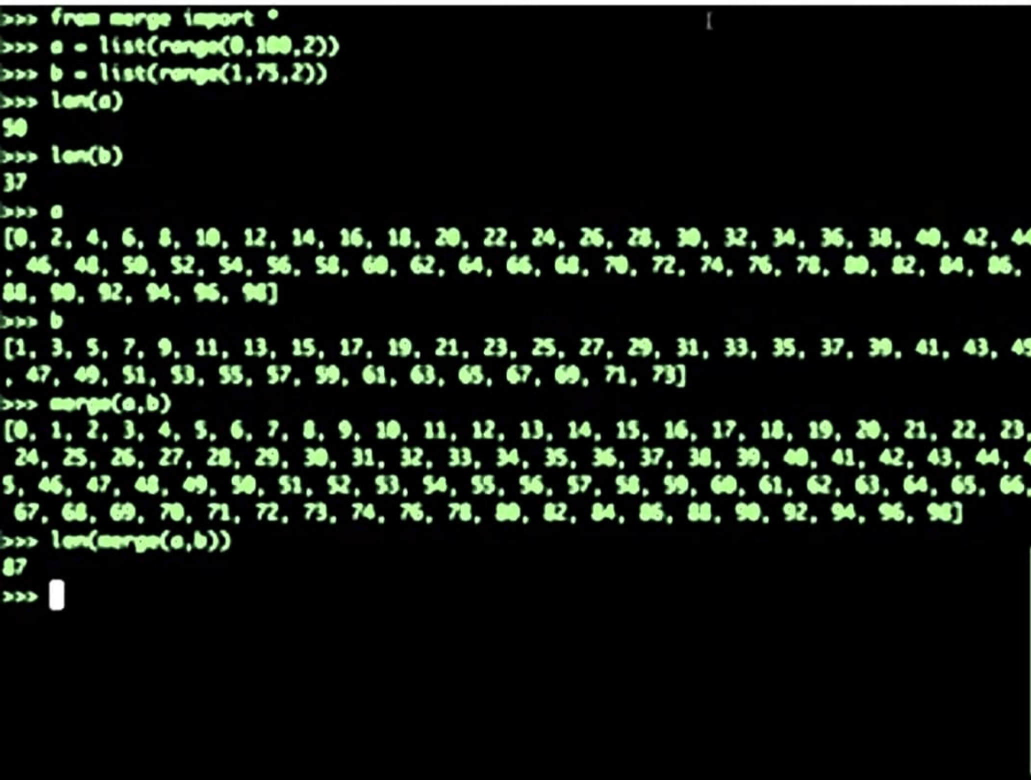
- If A is empty, copy [illegible]
- If B is empty, copy [illegible]
- Otherwise, compare first element of A and B and move the smaller of the two into C
- Repeat until all elements in A and B have been moved

Merging

```
def merge(A,B): # Merge A[0:m],B[0:n]
  (C,m,n) = ([],len(A),len(B))
  (i,j) = (0,0) # Current positions in A,B
  while i+j < m+n: # i+j is number of elements merged so far
    if i == m:  # Case 1: A is empty
      C.append(B[j])
      j = j+1
    elif j == n: # Case 2: B is empty
      C.append(A[i])
      i = i+1
    elif A[i] <= B[j]: # Case 3: Head of A is smaller
      C.append(A[i])
      i = i+1
    elif A[i] > B[j]: # Case 4: Head of B is smaller
      C.append(B[j])
      j = j+1
  return(C)
```



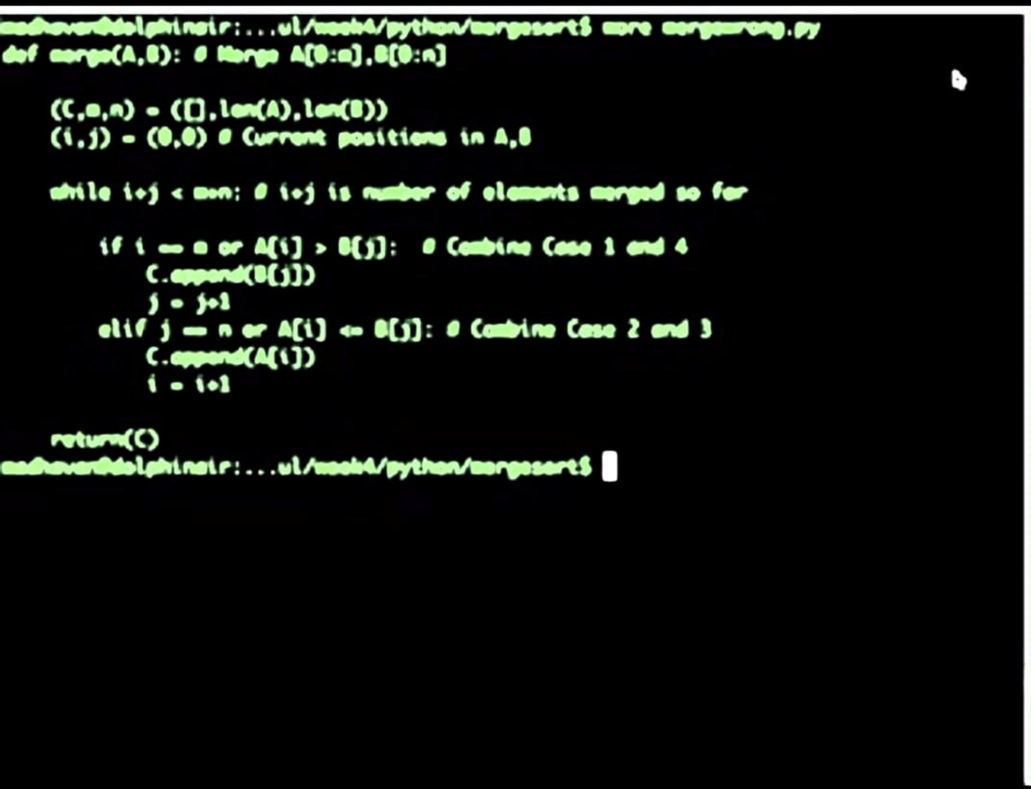
```
adhavan@dolphinoir:...ul/week4/python/mergesort$ more merge.py
ef merge(A,B): # Merge A[0:m],B[0:n]

    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B

    while i+j < m+n: # i+j is number of elements merged so far

        if j == n:  # Case 1: A is empty
            C.append(A[i])
            i = i+1
        elif i == m:  # Case 2: B is empty
            C.append(B[j])
            j = j+1
        elif A[i] <= B[j]:  # Case 3: Head of A is smaller
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]: # Case 4: Head of B is smaller
            C.append(B[j])
            j = j+1

    return(C)
adhavan@dolphinoir:...ul/week4/python/mergesort$
```



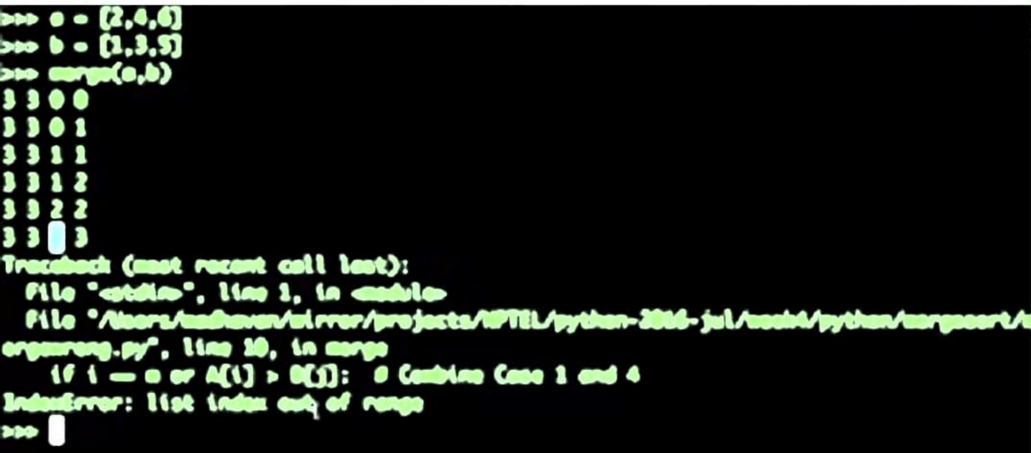
```
>>> from merge import *
>>> a = list(range(0,100,2))
>>> b = list(range(1,75,2))
>>> len(a)
50
>>> len(b)
37
>>> a
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44
, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
88, 90, 92, 94, 96, 98]
>>> b
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45
, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73]
>>> merge(a,b)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 4
5, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
 67, 68, 69, 70, 71, 72, 73, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
>>> len(merge(a,b))
87
>>> 
```

Merging, wrong

```
def mergewrong(A,B): # Merge A[0:m],B[0:n]
  (C,m,n) = ([],len(A),len(B))
  (i,j) = (0,0) # Current positions in A,B
  while i+j < m+n:
  # i+j is number of elements merged so far
    # Combine Case 1, Case 4
    if i == m or A[i] > B[j]:
      C.append(B[j])
      j = j+1
    # Combine Case 2, Case 3:
    elif j == n or A[i] <= B[j}:
      C.append(A[i])
      i = i+1
  return(C)
```



```
madhavan@dolphinair:...ul/week4/python/mergesort$ more mergewrong.py
def merge(A,B): # Merge A[0:m],B[0:n]

    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B

    while i+j < m+n: # i+j is number of elements merged so far

        if i == m or A[i] > B[j]:  # Combine Case 1 and 4
            C.append(B[j])
            j = j+1
        elif j == n or A[i] <= B[j]: # Combine Case 2 and 3
            C.append(A[i])
            i = i+1

    return(C)
madhavan@dolphinair:...ul/week4/python/mergesort$ 
```

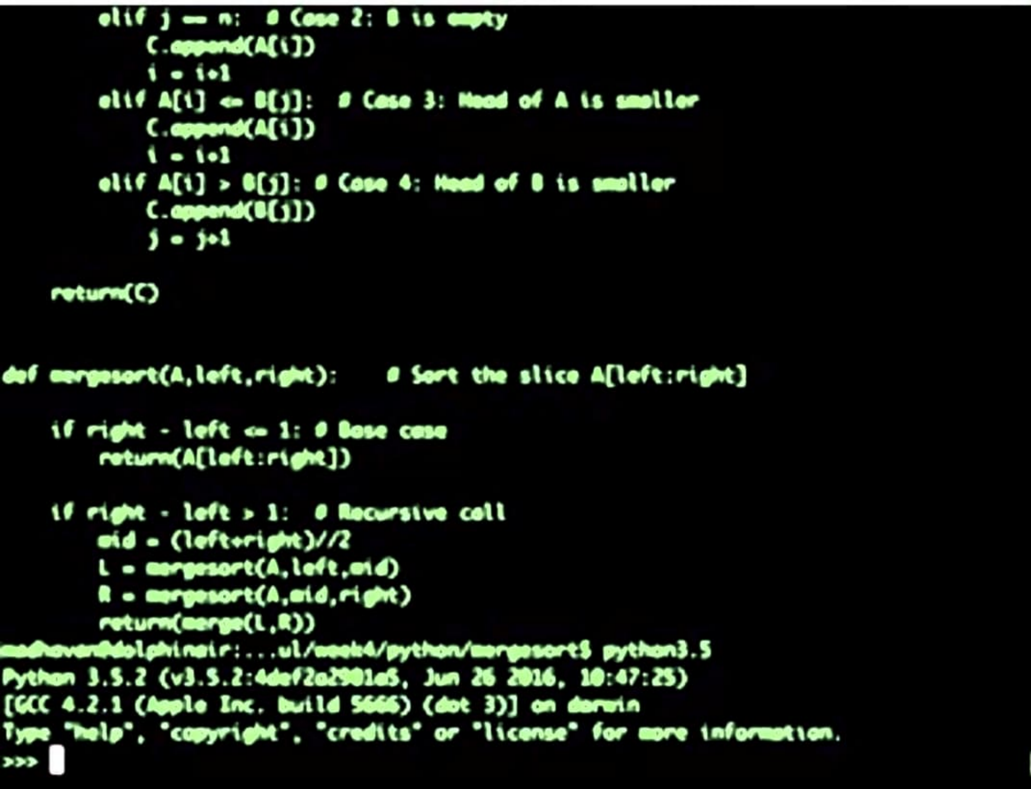


```
>>> a = [2,4,6]
>>> b = [1,3,5]
>>> merge(a,b)
3 3 0 0
3 3 0 1
3 3 1 1
3 3 1 2
3 3 2 2
3 3 █ 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/mergesort/m
ergewrong.py", line 10, in merge
    if i == m or A[i] > B[j]:  # Combine Case 1 and 4
IndexError: list index out of range
>>> █
```

Merge Sort

To sort `A[0:n]` into `B[0:n]`

- If n is 1, nothing to be done
- Otherwise
    - Sort `A[0:n//2]` into `L` (left)
    - Sort `A[n//2:n]` into `R` (right)
    - Merge `L` and `R` into `B`



```
        elif j == n:  # Case 2: B is empty
            C.append(A[i])
            i = i+1
        elif A[i] <= B[j]:  # Case 3: Head of A is smaller
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]: # Case 4: Head of B is smaller
            C.append(B[j])
            j = j+1

    return(C)


def mergesort(A,left,right):    # Sort the slice A[left:right]

    if right - left <= 1: # Base case
        return(A[left:right])

    if right - left > 1:  # Recursive call
        mid = (left+right)//2
        L = mergesort(A,left,mid)
        R = mergesort(A,mid,right)
        return(merge(L,R))
madhavan@dolphinair:...ul/week4/python/mergesort$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Analysis of Merge

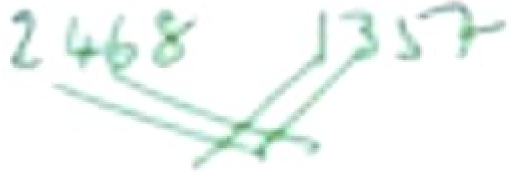
How much time does Merge take?

- Merge A of size m, B of size n into C
- In each iteration, we add one element to C
  - Size of C is $m+n$
  - $m+n \leq 2 \max(m,n)$
- Hence $O(\max(m,n)) = O(n)$ if $m \approx n$

Merge Sort: Shortcomings

- Merging A and B cr [illegible] w array C
  - No obvious way t [illegible] ly merge in place
- Extra storage can be costly
- Inherently recursive
  - Recursive call and return are expensive

Alternative approach



- Extra space is required to merge
- Merging happens because elements in left half must move right and vice versa
- Can we divide so that everything to the left is smaller than everything to the right?
  - No need to merge!

Divide and conquer without merging

- Suppose the median value in A is m
- Move all values $\leq m$ to left half of A
  - Right half has values $> m$
  - This shifting can be done in place, in time $O(n)$

Divide and conquer without merging

- Suppose the median value in A is m
- Move all values $\leq m$ to left half of A
  - Right half has values $> m$
  - This shifting can be done in place, in time $O(n)$
- Recursively sort left and right halves
- A is now sorted! No need to merge
  - $T(n) = 2T(n/2) + n = O(n \log n)$

Quicksort in Python

```
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1:  # Base case
    return ()

  # Partition with respect to pivot, a[l]
  yellow = l+1

  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1

  # Move pivot into place
  (A[l],A[yellow-1]) = (A[yellow-1],A[l])

  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
```

Merge Sort: Shortcomings

- Merging A and B creates a new array C
  - No obvious way to efficiently merge in place
- Extra storage can be costly
- Inherently recursive
  - Recursive call and return are expensive

Alternative approach

- Extra space is required to merge
- Merging happens because elements in left half must move right and vice versa
- Can we divide so that everything to the left is smaller than everything to the right?
    - No need to merge!

Divide and conquer without merging

- Suppose the median value in A is m
- Move all values $\leq m$ to left half of A
  - Right half has values $> m$
  - This shifting can be done in place, in time $O(n)$
- Recursively sort left and right halves

Divide and conquer without merging

- Suppose the median value in A is m
- Move all values $\leq$ m to left half of A
  - Right half has values $>$ m
  - This shifting can be done in place, in time $O(n)$
- Recursively sort left and right halves
- A is now sorted! No need to merge
  - $T(n) = 2T(n/2) + n = O(n \log n)$

Quicksort

- Choose a pivot element
  - Typically the first value in the array
- Partition A into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition

Quicksort

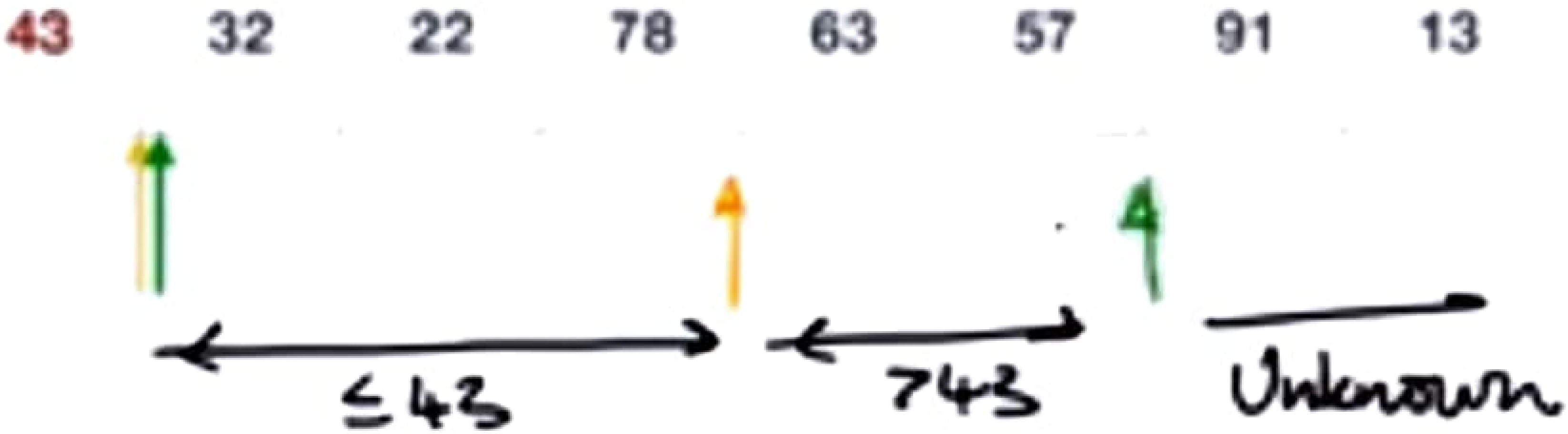
- Choose a pivot element
    - Typically the first value in the array
- Partition A into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions

Quicksort

- High level view

13	32	22	43	63	57	91	78

Quicksort: Partitioning

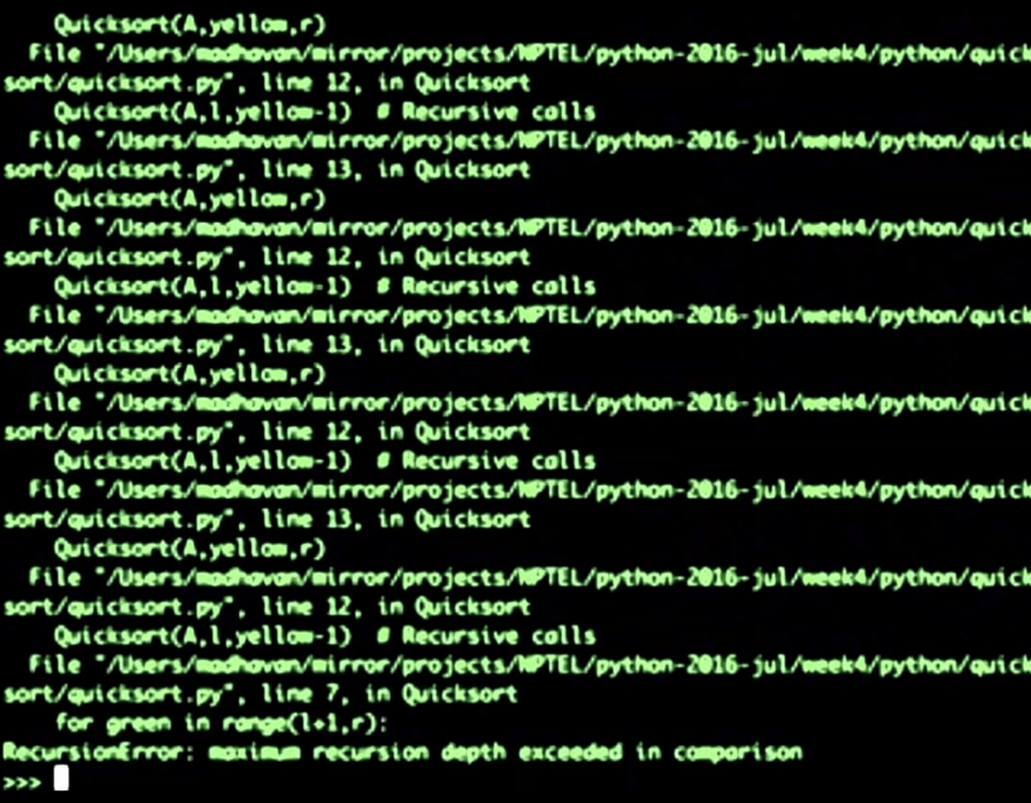


43 32 22 78 63 57 91 13

Quicksort in Python

```
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1:  # Base case
    return ()

  # Partition with respect to pivot, a[l]
  yellow = l+1

  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1

  # Move pivot into place
  (A[l],A[yellow-1]) = (A[yellow-1],A[l])

  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
```

```
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 10
104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 11
120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 13
136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 15
152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 16
168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 18
184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 19
200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 21
216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 23
232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 24
248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 26
264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 27
280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 29
296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 31
312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 32
328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 34
344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 35
360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 37
376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 39
392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 4
408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 4
424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 4
440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 4
456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 4
472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 4
488, 489, 490, 491, 492, 493, 494, 495, 496
```



```
    Quicksort(A,yellow,r)
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 12, in Quicksort
    Quicksort(A,l,yellow-1)  # Recursive calls
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 13, in Quicksort
    Quicksort(A,yellow,r)
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 12, in Quicksort
    Quicksort(A,l,yellow-1)  # Recursive calls
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 13, in Quicksort
    Quicksort(A,yellow,r)
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 12, in Quicksort
    Quicksort(A,l,yellow-1)  # Recursive calls
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 13, in Quicksort
    Quicksort(A,yellow,r)
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 12, in Quicksort
    Quicksort(A,l,yellow-1)  # Recursive calls
  File "/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week4/python/quick
sort/quicksort.py", line 7, in Quicksort
    for green in range(l+1,r):
RecursionError: maximum recursion depth exceeded in comparison
>>> 
```

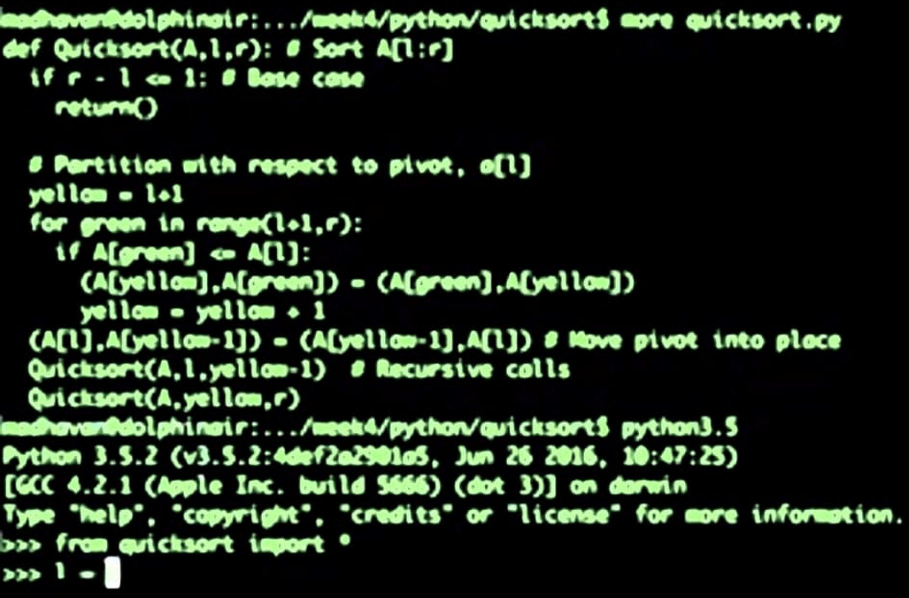
Quicksort

- Choose a pivot element
  - Typically the first value in the array
- Partition A into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions

Analysis of Quicksort

Worst case

- Pivot is either maximum or minimum
  - One partition is empty
  - Other has size n-1
  - $T(n) = T(n-1) + n = T(n-2) + (n-1) + n$
    $= \ldots = 1 + 2 + \ldots + n = O(n^2)$



```
madhavan@dolphinair:.../week4/python/quicksort$ more quicksort.py
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1: # Base case
    return()

  # Partition with respect to pivot, a[l]
  yellow = l+1
  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1
  (A[l],A[yellow-1]) = (A[yellow-1],A[l]) # Move pivot into place
  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
madhavan@dolphinair:.../week4/python/quicksort$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from quicksort import *
>>> l =
```

Analysis of Quicksort

But …

- Average case is O(n log n)
  - All permutations of n values, each equally likely
  - Average running time across all permutations
- Sorting is a rare example where average case can be computed

Quicksort

- Choose a pivot element
  - Typically the first value in the array
- Partition A into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions

Quicksort in Python

```
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1:  # Base case
    return ()

  # Partition with respect to pivot, a[l]
  yellow = l+1

  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1

  # Move pivot into place
  (A[l],A[yellow-1]) = (A[yellow-1],A[l])

  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
```

Analysis of Quicksort

Worst case

- Pivot is either maximum or minimum
  - One partition is empty
  - Other has size n-1
  - $T(n) = T(n-1) + n = T(n-2) + (n-1) + n$
    $= \ldots = 1 + 2 + \ldots + n = O(n^2)$

Analysis of Quicksort

Worst case

- Pivot is either maximum or minimum
  - One partition is empty
  - Other has size n-1
  - $T(n) = T(n-1) + n = T(n-2) + (n-1) + n$
    $= \ldots = 1 + 2 + \ldots + n = O(n^2)$

Analysis of Quicksort

But …

- Average case is O(n log n)
  - All permutations of n values, each equally likely
  - Average running time across all permutations
- Sorting is a rare example where average case can be computed

Quicksort: randomization

- Worst case arises because of fixed choice of pivot
  - We chose the first element
  - For any fixed strategy (last element, midpoint), can work backwards to construct $O(n^2)$ worst case
- Instead, choose pivot randomly
  - Pick any index in `range(0,n)` with uniform probability
- Expected running time is again $O(n \log n)$

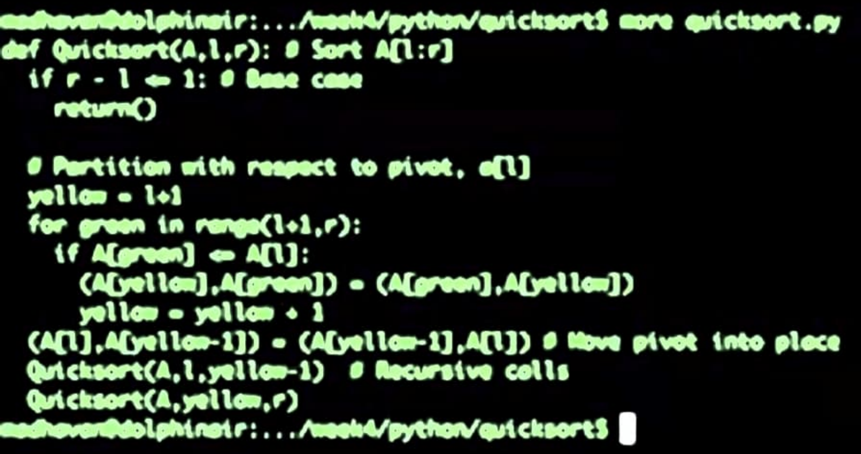
Quicksort in practice

- In practice, Quicksort is very fast
    - Typically the default algorithm for in-built sort functions
        - Spreadsheets
        - Built in sort function in programming languages

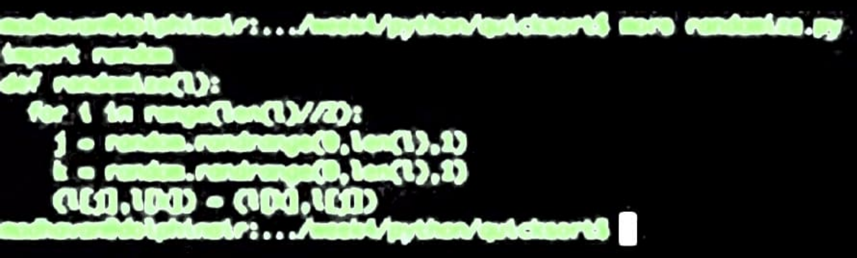
Quicksort in practice

l.sort()

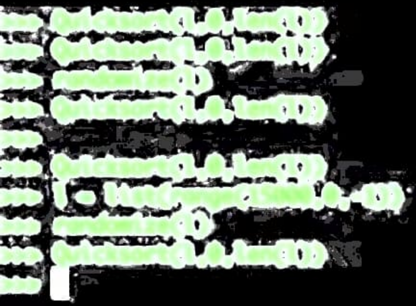
- In practice, Quicksort is very fast
    - Typically the default algorithm for in-built sort functions
        - Spreadsheets
        - Built in sort function in programming languages



```
madhavan@dolphinair:.../week4/python/quicksort$ more quicksort.py
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1: # Base case
    return()

  # Partition with respect to pivot, a[l]
  yellow = l+1
  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1
  (A[l],A[yellow-1]) = (A[yellow-1],A[l]) # Move pivot into place
  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
madhavan@dolphinair:.../week4/python/quicksort$
```



```
madhavan@dolphinair:.../week4/python/quicksort$ more randomize.py
import random
def randomize(l):
  for i in range(len(l)//2):
    j = random.randrange(0,len(l),1)
    k = random.randrange(0,len(l),1)
    (l[j],l[k]) = (l[k],l[j])
madhavan@dolphinair:.../week4/python/quicksort$
```



```
>>> Quicksort(l,0,len(l))
>>> Quicksort(l,0,len(l))
>>> randomize()
>>> Quicksort(l,0,len(l))
>>>
>>> Quicksort(l,0,len(l))
>>> l = list(range(15000,0,-1))
>>> randomize()
>>> Quicksort(l,0,len(l))
>>> 
```

Stable sorting

- Sorting on multiple criteria
- Assume students are listed in alphabetical order
- Now sort students by marks
    - After sorting, are students with equal marks still in alphabetical order?
- Stability is crucial in applications like spreadsheets
    - Sorting column B should not disturb previous sort on column A

Stable sorting …

- Quicksort, as described, is not stable
  - Swap operation during partitioning disturbs original order
- Merge sort is stable if we merge carefully
  - Do not allow elements from right to overtake elements from left
  - Favour left list when breaking ties

Tuples

- Simultaneous assignments

```
(age,name,primes) = (23,"Kamal",[2,3,5])
```

- Can assign a "tuple" of values to a name

```
point = (3.5,4.8)
date = (16,7,2013)
```

- Extract positions, slices

```
xcoordinate = point[0]
monthyear = date[1:]
```

- Tuples are immutable

`date[1] = 8` is an error

Generalizing lists

- `l = [13, 46, 0, 25, 72]`
- View `l` as a function, associating values to positions
  - `l : {0,1,...,4} → integers`
  - `l(0) = 13, l(4) = 72`
- `0,1,...,4` are keys
- `l[0],l[1],...,l[4]` are corresponding values

Dictionaries

- Allow keys other than `range(0,n)`
- Key could be a string

```
test1["Dhawan"] = 84
test1["Pujara"] = 16
test1["Kohli"] = 200
```

- Python dictionary Ass
    - Any immutable value can be a key

Dictionaries

- Allow keys other than `range(0,n)`
- Key could be a string

```
test1["Dhawan"] = 84
test1["Pujara"] = 16
test1["Kohli"] = 200
```

- Python dictionary
  - Any immutable value can be a key
  - Can update dictionaries in place —mutable, like lists

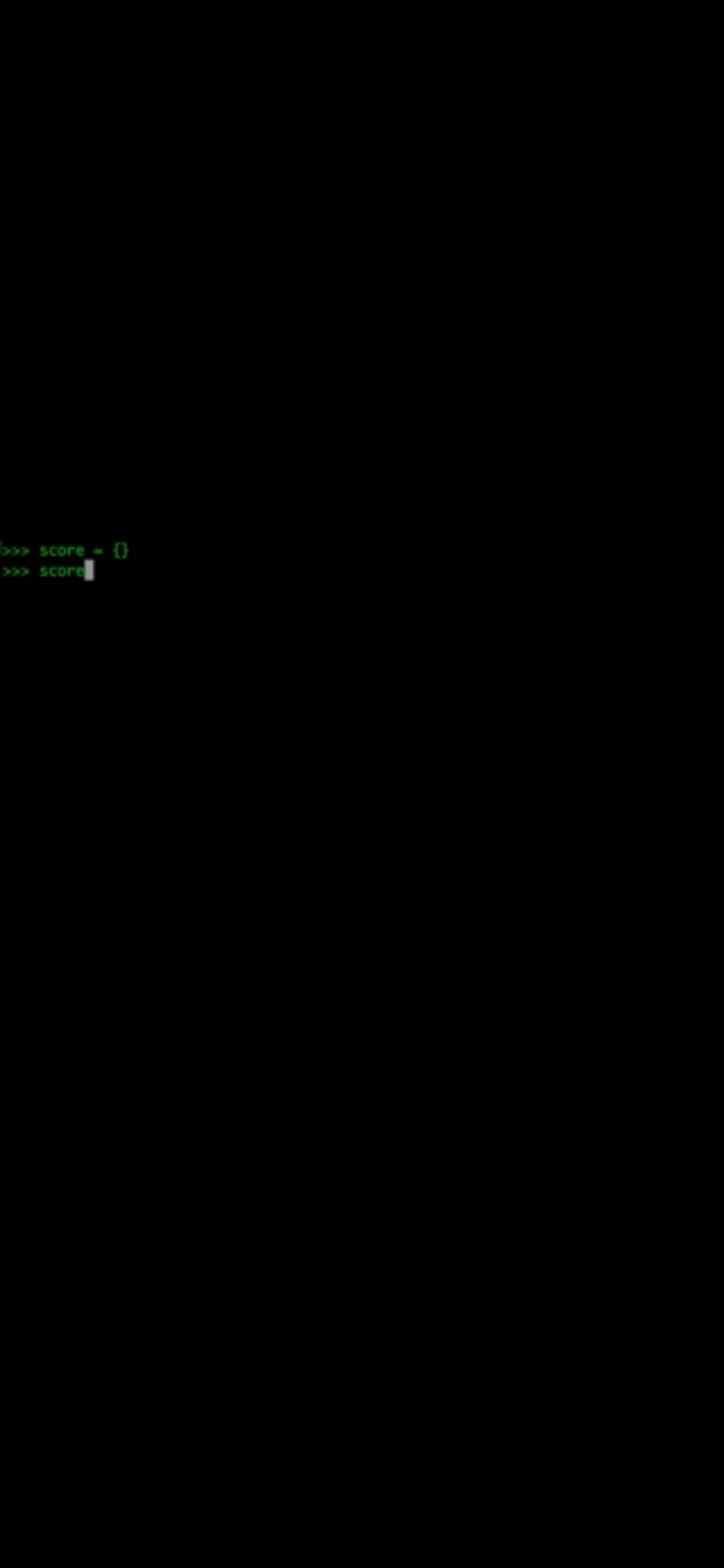
Dictionaries

- Can nest dictionaries

```
score["Test1"]["Dhawan"] = 84
score["Test2"]["Kohli"] = 200
score["Test2"]["Dhawan"] = 27
```

- Directly assign values to a dictionary

```
score = {"Dhawan":84, "Kohli":200}
score = {"Test1":{"Dhawan":84,
  "Kohli":200}, "Test2":{"Dhawan":50}}
```



```
>>> score = {}
>>> score
```

Operating on dictionaries

- d.keys() returns sequence of keys of dictionary d

```
for k in d.keys():
  # Process d[k]
```

- d.keys() is not in any predictable order

```
for k in sorted(d.keys()):
  # Process d[k]
```

Operating on dictionaries

- `d.keys()` returns sequence of keys of dictionary `d`

```
for k in d.keys():
  # Process d[k]
```

- `d.keys()` is not in any predictable order

```
for k in sorted(d.keys()):
  # Process d[k]
```

- `sorted(l)` returns sorted copy of `l`, `l.sort()` sorts `l` in place

Dictionaries

- Can nest dictionaries

```
score["Test1"]["Dhawan"] = 84
score["Test2"]["Kohli"] = 200
score["Test2"]["Dhawan"] = 27
```

Operating on dictionaries

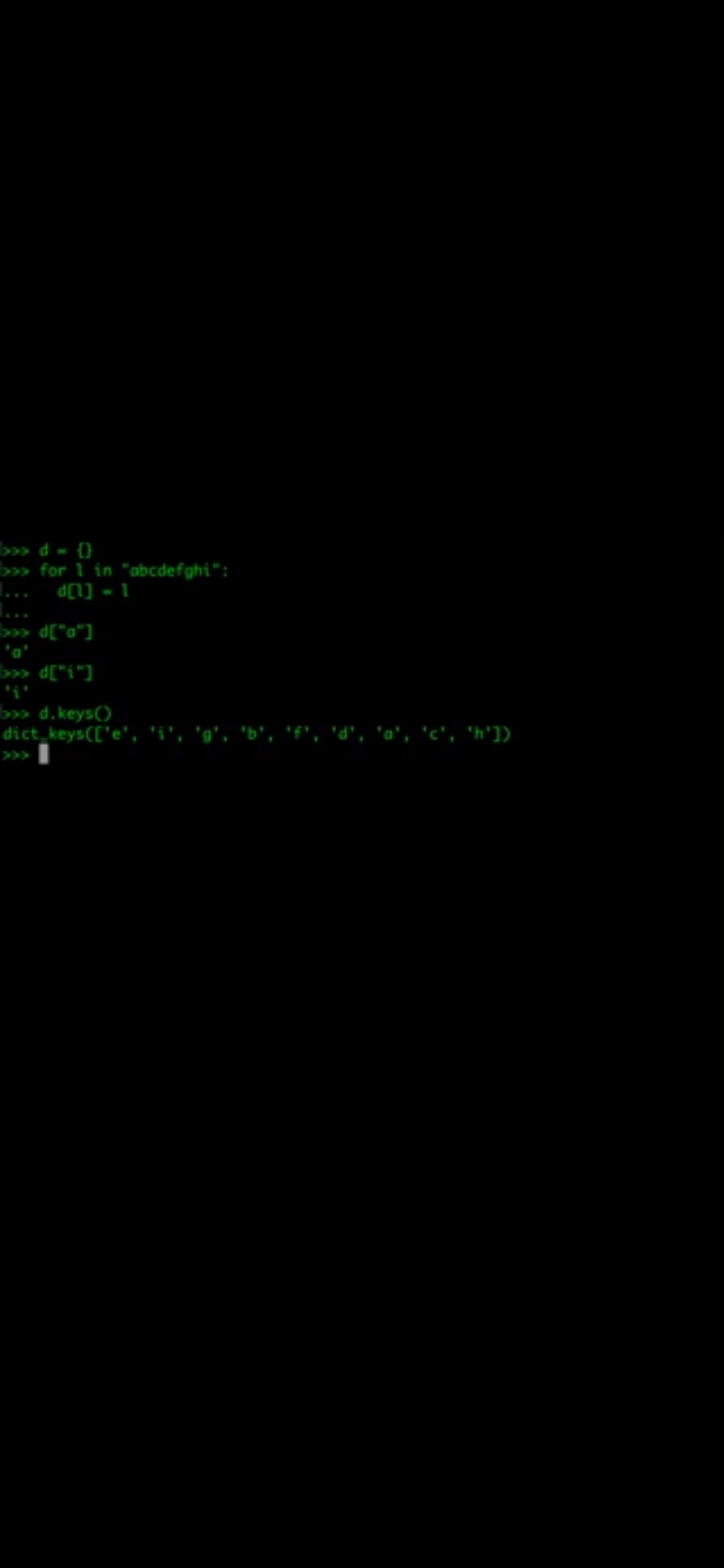
- `d.keys()` returns sequence of keys of dictionary `d`

```
for k in d.keys():
  # Process d[k]
```

- `d.keys()` is not in any predictable order

```
for k in sorted(d.keys()):
  # Process d[k]
```

- `sorted(l)` returns sorted copy of `l`, `l.sort()` sorts `l` in place
- `d.keys()` is not a list —use `list(d.keys())`



```
>>> d = {}
>>> for l in "abcdefghi":
...     d[l] = l
...
>>> d["a"]
'a'
>>> d["i"]
'i'
>>> d.keys()
dict_keys(['e', 'i', 'g', 'b', 'f', 'd', 'a', 'c', 'h'])
>>> 
```

Operating on dictionaries

- Similarly, `d.values()` is sequence of values in *d*

```
total = 0
for s in test1.values():
  total = total + test1
```

- Test for key using `in`, like list membership

```
for n in ["Dhawan","Kohli"]:
  total[n] = 0
  for match in score.keys():
    if n in score[match].keys():
      total[n] = total[n] + score[match][n]
```

Operating on dictionaries

- Similarly, `d.values()` is sequence of values in *d*

```
total = 0
for s in test1.values():
  total = total + test1
```

- Test for key using `in`, like list membership

```
for n in ["Dhawan","Kohli"]:
  total[n] = 0
  for match in score.keys():
    if n in score[match].keys():
      total[n] = total[n] + score[match][n]
```

Dictionaries vs lists

- Assigning to an unknown key inserts an entry

```
d = {}
d[0] = 7  # No problem, d == {0:7}
```

Dictionaries vs lists

- Assigning to an unknown key inserts an entry

```
d = {}
d[0] = 7  # No problem, d == {0:7}
```

- … unlike a list

```
l = []
l[0] = 7  # IndexError!
```

Summary

- Dictionaries allow a flexible association of values to keys
    - Keys must be immutable values
- Structure of dictionary is internally optimized for key-based lookup
    - Use `sorted(d.keys())` to retrieve keys in predictable order
- Extremely useful for manipulating information from text files, tables … — use column headings as keys

Passing values to functions

- Argument value is substituted for name

```
def power(x,n):
  ans = 1
  for i in range(0,n):
    ans = ans*x
  return(ans)
```

```
power(3,5)

x = 3
n = 5
ans = 1
for i in range..
```

- Like an implicit assignment statement

Pass arguments by name

```
def power(x,n):
  ans = 1
  for i in range(0,n):
    ans = ans*x
  return(ans)
```

- Call `power(n=5,x=4)`

Default arguments

- Recall `int(s)` that converts string to integer
  - `int("76")` is `76`

int("76", 10)

  - `int("A5")` generates an error
- Actually `int(s,b)` takes two arguments, string s and base b
  - b has default value `10`

Default arguments

- Recall `int(s)` that converts string to integer
    - `int("76") is 76`
    - `int("A5") generates an error`
- Actually `int(s,b)` takes two arguments, string s and base b
    - b has default value `10`
    - `int("A5",16)` is `165` (10 x 16 + 5)

Default arguments

```
def int(s,b=10):
    . . .
```

- Default value is provided in function definition
- If parameter is omitted, default value is used
  - Default value must be available at definition time
  - `def Quicksort(A,l=0,r=len(A)):` does not work

Default arguments

```
def f(a,b,c=14,d=22):
    . . .
```

- `f(13,12)` is interpreted as `f(13,12,14,22)`
- `f(13,12,16)` is interpreted as `f(13,12,16,22)`
- Default values are identified by position, must come at the end

Default arguments

```
def f(a,b,c=14,d=22):
    . . .
```

- `f(13,12)` is interpreted as `f(13,12,14,22)`
- `f(13,12,16)` is interpreted as `f(13,12,16,22)`
- Default values are identified by position, must come at the end
  - Order is important

Function definitions

- def associates a function body with a name
- Flexible, like other value assignments to name
- Definition can be conditional

```
if condition:
  def f(a,b,c):
    . . .
else:
  def f(a,b,c):
    . . .
```

Function definitions

- Can assign a function to a new name

```
def f(a,b,c):
    . . .

g = f
```

- Now *g* is another name for f

Can pass functions

- Apply f to x n times

```
def apply(f,x,n):
  res = x
  for i in range(n):
    res = f(res)
  return(res)
```

Can pass functions

- Apply f to x n times

```
def apply(f,x,n):
  res = x
  for i in range(n):
    res = f(res)
  return(res)
```

```
def square(x):
  return(x*x)

apply(square,5,2,1)

square(square(5))

625
```

Passing functions

- Useful for customizing functions such as sort
- Define `cmp(x,y)` that returns `-1` if `x < y`, `0` if `x == y` and `1` if `x > y`
    - `cmp("aab","ab")` is `-1` in dictionary order
    - `cmp("aab","ab")` is `1` if we compare by length
- `def sortfunction(l,cmpfn=defaultcmpfn):`

Summary

- Function definitions behave like other assignments of values to names
- Can reassign a new definition, define conditionally

  ...
- Can pass function names to other functions

Built in function map()

- map(f,l) applies f to each element of l
- Output of map(f,l) is not a list!
    - Use list(map(f,l)) to get a list
    - Can be used directly in a for loop

```
for i in ~~list~~(map(f,l)~~)~~:
```

    - Like range(i,j), d.keys()

Selecting a sublist

- Extract list of primes from list `numberlist`

```
primelist = []
for i in numberlist:
  if isprime(i):
    primelist.append(i)
return(primelist)
```

Selecting a sublist

- In general

```
def select(property,l):
  sublist = []
  for x in l:
    if property(x):
      sublist.append(x)
  return(sublist)
```

Combining map and filter

- Sum of squares of even numbers from 0 to 99

```
list(map(square,filter(iseven,range(100)))

def square(x):
  return(x*x)

def iseven(x):
  return(x%2 == 0)
```

Combining map and filter

- Sum of squares of even numbers from 0 to 99

```
list(map(square,filter(iseven,range(100)))

def square(x):
  return(x*x)

def iseven(x):
  return(x%2 == 0)
```

List comprehension

- Pythagorean triple: $x^2 + y^2 = z^2$
- All Pythagorean triples (x,y,z) with values below n

$$\{\ \underline{(x,y,z)}\ \boxed{|}\ \underline{1 \leq x,y,z \leq n},\ x^2 + y^2 = z^2\ \}$$

List comprehension

- Pythagorean triple: $x^2 + y^2 = z^2$
- All Pythagorean triples (x,y,z) with values below n

$$\{ (x,y,z) \mid 1 \leq x,y,z \leq n,\ x^2 + y^2 = z^2 \}$$

- In set theory, this is called set comprehension
  - Building a new set from existing sets

List comprehension

- Squares of even numbers below 100

```
[square(x) for i in range(100) if iseven(x)]
```

map generator filter

Multiple generators

- Pythagorean triples with x,y,z below 100

```
[(x,y,z) for x in range(100)
          for y in range(100)
           for z in range(100)
            if x*x + y*y == z*z]
```

- Order of x,y,z is like nested for loop   0 ,

```
for x in range(100): 0
  for y in range(100): 0
    for z in range(100): 0
```

Multiple generators

- Later generators can depend on earlier ones
- Pythagorean triples with x,y,z below 100, no duplicates

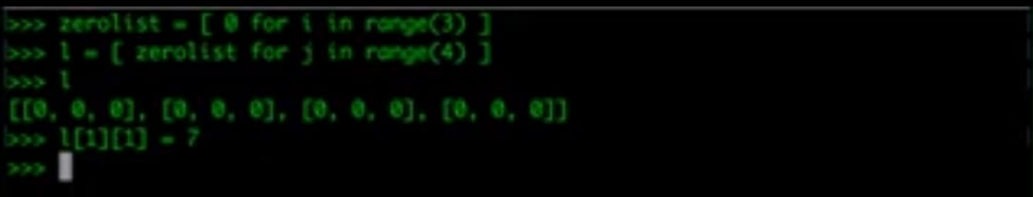
```
[(x,y,z) for x in range(100)
          for y in range(x,100)
           for z in range(y,100)
            if x*x + y*y == z*z]
```

Useful for initialising lists

- Initialise a 4 x 3 matrix
  - 4 rows, 3 columns
  - Stored row-wise

```
l = [ [ 0 for i in range(3) ]
          for j in range(4)]
```

for each row



```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]
>>> l
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> l[1][1] = 7
>>> 
```

Warning

- What's happening here?

```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]

>>> l[1][1] = 7
```

Warning

- What's happening here?

```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]

>>> l[1][1] = 7

>>> l
[[0,7,0],[0,7,0],[0,7,0],[0,7,0]]
```

- Each row in `l` points to same list `zerolist`

Warning

- What's happening here?

```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]

>>> l[1][1] = 7

>>> l
[[0,7,0],[0,7,0],[0,7,0],[0,7,0]]
```

- Each row in `l` points to same list `zerolist`

Summary

- `map` and `filter` are useful functions to manipulate lists
- List comprehension provides a useful notation for combining `map` and `filter`