More about range()

- `range(i,j)` produces the sequence `i,i+1,…,j-1`
- `range(j)` automatically starts from `0`; `0,1,…,j-1`
- `range(i,j,k)` increments by `k`; `i,i+k,…,i+nk`
  - Stops with `n` such that `i+nk < j <= i+(n+1)k`
- Count down? Make `k` negative!
  - `range(i,j,-1)`, `i > j`, produces `i,i-1,…,j+1`

More about `range()`

- General rule for `range(i,j,k)`
  - Sequence starts from `i` and gets as close to `j` as possible without crossing `j`

More about range()

- General rule for `range(i,j,k)`
  - Sequence starts from `i` and gets as close to `j` as possible without crossing `j`
- If `k` is positive and `i >= j`, empty sequence
  - Similarly if `k` is negative and `i <= j`
- If `k` is negative, stop "before" `j`
  - `range(12,1,-3)` produces `12,9,6,3`

More about range()

- Why does `range(i,j)` stop at `j-1`?
  - Mainly to make it easier to process lists
  - List of length `n` has positions `0,1,..,n-1`
  - `range(0,len(l))` produces correct range of valid indices
    - Easier than writing `range(0,len(l)-1)`

range() and lists

- Compare the following
  - `for i in [0,1,2,3,4,5,6,7,8,9]:`
  - `for i in range(0,10):`

range() and lists

- Compare the following
  - `for i in [0,1,2,3,4,5,6,7,8,9]:`
  - `for i in range(0,10):`
- Is `range(0,10) == [0,1,2,3,4,5,6,7,8,9]`?
  - In Python2, yes
  - In Python3, no!

range() and lists

- Can convert `range()` to a list using `list()`
  - `list(range(0,5)) == [0,1,2,3,4]`
- Other type conversion functions using type names
  - `str(78) = "78"`
  - `int("321") = 321`

range() and lists

- Can convert `range()` to a list using `list()`
  - `list(range(0,5)) == [0,1,2,3,4]`
- Other type conversion functions using type names
  - `str(78) = "78"`
  - `int("321") = 31`
    - But `int("32x")` yields error

Summary

- `range(n)` has is implicitly from `0` to `n-1`
- `range(i,j,k)` produces sequence in steps of `k`
  - Negative `k` counts down
- Sequence produced by `range()` is not a list
  - Use `list(range(..))` to get a list

Lists

- Lists are mutable
  - `list1 = [1,3,5,6]`
    `list2 = list1`
    `list1[2] = 7`
  - `list1` is now `[1,3,7,6]`
  - So is `list2`

Lists

- On the other hand
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1 = list1[0:2] + [7] + list1[3:]
    ```
  - `list1` is now `[1,3,7,6]`
  - `list2` remains `[1,3,5,6]`
    ```

Lists

- On the other hand
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1 = list1[0:2] + [7] + list1[3:]
    ```
  - `list1` is now `[1,3,7,6]`
  - `list2` remains `[1,3,5,6]`
- Concatenation produces a new list
    ```

```
...then-2016-jul/week3/python3 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.
>>> list1 = [1,3,5,6]
>>> list2 = list1
>>> list1[2] = 7
>>> list1
[1, 3, 7, 6]
>>> list2
[1, 3, 7, 6]
>>> list1 = [1,3,5,6]
>>> list2 = list1
>>> list1 = list1[0:2] + [7] + list[3:]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'type' object is not subscriptable
>>> list1 = list1[0:2] + [7] + list1[3:]
>>>
```

```
...then-2016-jul/week3/python3 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.
>>> list1 = [1,3,5,6]
>>> list2 = list1
>>> list1[2] = 7
>>> list1
[1, 3, 7, 6]
>>> list2
[1, 3, 7, 6]
>>> list1 = [1,3,5,6]
>>> list2 = list1
>>> list1 = list1[0:2] + [7] + list[3:]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'type' object is not subscriptable
>>> list1 = list1[0:2] + [7] + list1[3:]
>>> list1
[1, 3, 7, 6]
>>> list2
[1, 3, 5, 6]
>>>
```

Extending a list

- **Adding an element to a list, in place**
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1.append(12)
    ```
    ```

Extending a list

- Adding an element to a list, in place
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1.append(12)
    ```
  - `list1` is now `[1,3,5,6,12]`
  - `list2` is also `[1,3,5,6,12]`
    ```

List functions

- `list1.append(v)` — extend `list1` by a single value `v`
- `list1.extend(list2)` — extend `list1` by a list of values
  - In place equivalent of `list1 = list1 + list2`

List functions

- `list1.append(v)` — extend `list1` by a single value `v`
- `list1.extend(list2)` — extend `list1` by a list of values
  - In place equivalent of `list1 = list1 + list2`
- `list1.remove(x)` — removes first occurrence of `x`
  - Error if no copy of `x` exists in `list1`

```
>>> list1 = list(range(10))
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list1.append(12)
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12]
>>> list1.extend([13,14])
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14]
>>> list2 = list1 + list1
>>> list2
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14]
>>> list2.remove(5)
>>> list2
[0, 1, 2, 3, 4, 6, 7, 8, 9, 12, 13, 14, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14]
>>> list2.remove(5)
>>> list2
[0, 1, 2, 3, 4, 6, 7, 8, 9, 12, 13, 14, 0, 1, 2, 3, 4, 6, 7, 8, 9, 12, 13, 14]
>>> list2.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

A note on syntax

- `list1.append(x)` rather than `append(list1,x)`
  - `list1` is an object
  - `append()` is a function to update the object
  - `x` is an argument to the function
- Will return to this point later

Further list manipulation

- Can also assign to a slice in place
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1[2:] = [7,8]
    ```
  - `list1` and `list2` are both `[1,3,7,8]`
- Can expand/shrink slices, but be sure you know what you are doing!
  - `list1[2:] = [9,10,11]` produces `[1,3,9,10,11]`
    ```

List membership

- `x in l` **returns** `True` **if value** `x` **is found in list** `l`

```
# Safely remove x from l
if x in l:
    l.remove(x)
```

Other functions

- `l.reverse()` — reverse `l` in place
- `l.sort()` — sort `l` in ascending order
- `l.index(x)` — find leftmost position of `x` in `l`
  - Avoid error by checking if `x` in `l`
- `l.rindex(x)` — find rightmost position of `x` in `l`
- Many more … see Python documentation!

Initialising names

- A name cannot be used before it is assigned a value

```
y = x + 1 # Error if x is unassigned
```

- May forget this for lists where update is implicit

```
l.append(v)
```

- Python needs to know that `l` is a list

Initialising names ...

```
def factors(n):

  for i in range(1,n+1):
    if n%i == 0:
      flist.append(i)

  return(flist)
```

Initialising names

- A name cannot be used before it is assigned a value

```
y = x + 1 # Error if x is unassigned
```

- May forget this for lists where update is implicit

```
l.append(v)
```

- Python needs to know that `l` is a list

Initialising names ...

```
def factors(n):

  flist = []

  for i in range(1,n+1):
    if n%i == 0:
      flist.append(i)

  return(flist)
```

Summary

- To extend lists in place, use `l.append()`, `l.extend()`
  - Can also assign new value, in place, to a slice
- Many built in functions for lists — see documentation
- Don't forget to assign a value to a name before it is first used

Loops revisited

- `for i in l:`

  `. . .`

  - Repeat body for each item in list `l`

- `while condition:`

  `. . .`

  - Repeat body till `condition` becomes `False`

Search for value in a list

```
def findpos(l,v):
    # Return first position of v in l
    # Return -1 if v not in l
    (found,i) = (False,0)
    while i < len(l):
        if l[i] == v:
            (found,pos) = (True,i)
    if not found:
        pos = -1
    return(pos)
```

Search for value in a list

l.index(v)

```
def findpos(l,v):
    # Return first position of v in l
    # Return -1 if v not in l
    (found,i) = (False,0)
    while i < len(l):
        if l[i] == v:
            (found,pos) = (True,i)
        i = i + 1
    if not found:
        pos = -1
    return(pos)
```

0, 1, . . , n-1

-1

Search for value in a list ...

- A more natural strategy
  - Scan list for value
  - Stop scan as soon as we find the value
  - If the scan completes without success, report -1

Search for value in a list ...

- A more natural strategy

```
def findpos(l,v):

  for x in l:
    if x == v:
      # Exit and report position of x

  # Loop over, report -1 if we did not see x
```

Search for value in a list ...

- A more natural strategy

```
def findpos(l,v)
  (pos,i) = (-1,0)
  for x in l:
    if x == v: # Exit, report position of x
      pos = i
      break
    i = i+1

  # If pos not reset in loop, pos is -1
  return(pos)
```

Search for value in a list ...

- A more natural strategy

```
def findpos(l,v)
    (pos,i) = (-1,0)
    for x in l:
        if x == v: # Exit, report position of x
            pos = i
            break
        i = i+1

    # If pos not reset in loop, pos is -1
    return(pos)
```

Search for value in a list ...

- A more natural strategy

```
def findpos(l,v)

  pos = -1
  for i in range(len(l)):
    if l[i] == v: # Exit, report position
      pos = i
      break

  # If pos not reset in loop, pos is -1
  return(pos)
```

Search for value in a list ...

- A loop can also have an `else:` — signals normal termination

```
def findpos(l,v)
  for i in range(len(l)):
    if l[i] == v: # Exit, report position
      pos = i
      break
  else:
    pos = -1 # No break, v not in l
  return(pos)
```

Summary

- Can exit prematurely from loop using `break`
  - .Applies to both for and while
- Loop also has an `else:` clause
  - Special action for normal termination

Sequences of values

- Two basic ways of storing a sequence of values
  - Arrays
  - Lists
- What's the difference?

Arrays

- Single block of memory, elements of uniform type
  - Typically size of sequence is fixed in advance
- Indexing is fast
  - Access `seq[i]` in constant time for any `i`
  - Compute offset from start of memory block
- Inserting between `seq[i]` and `seq[i+1]` is expensive

Arrays

- Single block of memory, elements of uniform type
  - Typically size of sequence is fixed in advance
- Indexing is fast
  - Access `seq[i]` in constant time for any `i`
  - Compute offset from start of memory block
- Inserting between `seq[i]` and `seq[i+1]` is expensive
- Contraction is expensive

Lists

- Values scattered in memory
  - Each element points to the next—"linked" list
  - Flexible size

Lists

- Values scattered in memory
  - Each element points to the next — "linked" list
  - Flexible size
- Follow `i` links to access `seq[i]`
  - Cost proportional to `i`

Lists

- Values scattered in memory
  - Each element points to the next—"linked" list
  - Flexible size
- Follow i links to access `seq[i]`
  - Cost proportional to `i`
- Inserting or deleting an element is easy
  - "Plumbing"

Operations

- Exchange `seq[i]` and `seq[j]`
  - Constant time in array, linear time in lists
- Delete `seq[i]` or Insert `v` after `seq[i]`
  - Constant time in lists (if we are already at `seq[i]`)
  - Linear time in array
- Algorithms on one data structure may not transfer to another
  - Example: Binary search

Search problem

- Is a value `v` present in a collection `seq`?
- Does the structure of `seq` matter?
  - Array vs list
- Does the organization of the information matter?
  - Values sorted/unsorted

The unsorted case

```
def search(seq,v)
  for x in seq:
    if x == v:
      return(True)
  return(False)
```

Worst case

- Need to scan the entire sequence `seq`
  - Time proportional to length of sequence
- Does not matter if `seq` is array or list

Search a sorted sequence

- What if `seq` is sorted?
  - Compare `v` with midpoint of `seq`
  - If midpoint is `v`, the value is found
  - If `v` < midpoint, search left half of `seq`
  - If `v` > midpoint, search right half of `seq`
- Binary search

Binary search ...

```
def bsearch(seq,v,l,r):
// search for v in seq[l:r], seq is sorted
  if (r - l == 0):
    return(False)
  mid = (l + r) // 2    // integer division
  if (v == seq[mid]):
    return (True)
  if (v < seq[mid]):
    return (bsearch(seq,v,l,mid))
  else:
    return (bsearch(seq,v,mid+1,r))
```

Binary Search ...

- How long does this take?
  - Each step halves the interval to search
  - For an interval of size 0, the answer is immediate
- T(n): time to search in an array of size n
  - T(0) = 1
  - T(n) = 1 + T(n/2)

Binary Search ...

- T(n): time to search in a list of size n
  - T(0) = 1
  - T(n) = 1 + T(n/2)

recurrence

Binary Search …

- T(n): time to search in a list of size n
  - T(0) = 1
  - T(n) = 1 + T(n/2)
- Unwind the recurrence
  - $T(n) = 1 + T(n/2) = 1 + 1 + T(n/2^2) = \ldots$
    $= 1 + 1 + \ldots + 1 + T(n/2^k)$
    $= 1 + 1 + \ldots + 1 + T(n/2^{\log n}) = O(\log n)$

Binary Search ...

$2^{10} = 1024$

- Works only for arrays
  - Need to look up `seq[i]` in constant time
- By seeing only a small fraction of the sequence, we can conclude that an element is not present!

Python lists

- Are built in lists in Python lists or arrays?
- Documentation suggests they are lists
  - Allow efficient expansion, contraction
- However, positional indexing allows us to treat them as arrays
  - In this course, we will "pretend" they are arrays
  - Will later see explicit implementation of lists

Efficiency

- Measure time taken by an algorithm as a function $T(n)$ with respect to input size n
- Usually report worst case behaviour
  - Worst case for searching in a sequence is when value is not found
  - Worst case is easier to calculate than "average" case or other more reasonable measures

O( ) notation

- Interested in broad relationship between input size and running time
- Is $T(n)$ proportional to $\log n$, n, $n \log n$, n^2, …, 2^n?
- Write $T(n) = O(n)$, $T(n) = O(n \log n)$, … to indicate this
  - Linear scan is $O(n)$ for arrays and lists
  - Binary search is $O(\log n)$ for sorted arrays

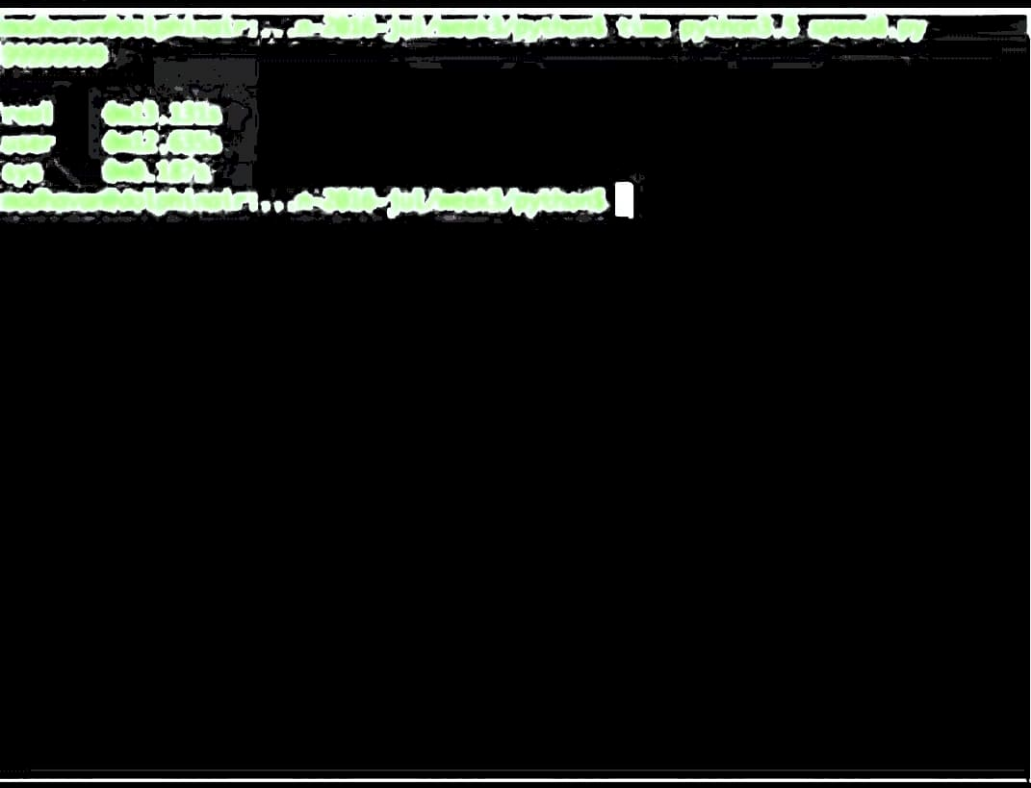
Typical functions T(n)...

Input	log n	n	n log n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

Input	log n	n	n log n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

Python can do about 10^7 steps in a second

```
mashaven@dolphinair:...on-2016-jul/week3/python$ ls
speed.py        speed4.py       speed5.py       speed6.py       speed7.py
speed8.py       speed9.py
mashaven@dolphinair:...n-2016-jul/week3/python$ more speed4.py
for i in range(10000):
  m = i
print(m)
mashaven@dolphinair:...n-2016-jul/week3/python$ more speed5.py
for i in range(100000):
  m = i
print(m)
mashaven@dolphinair:...n-2016-jul/week3/python$ more speed6.py
for i in range(1000000):
  m = i
print(m)
mashaven@dolphinair:...n-2016-jul/week3/python$ more speed7.py
for i in range(10000000):
  m = i
print(m)
mashaven@dolphinair:...n-2016-jul/week3/python$
```



Typical functions T(n)...

$2^{10} = 1024$
$2^{20} = 10^6$
$2^{30} = 10^9$

Input	log n	n	n log n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

Python can do about
10^7 steps in a second

Efficiency

- Theoretically $T(n) = O(n^k)$ is considered efficient
  - Polynomial time
- In practice even $T(n) = O(n^2)$ has very limited effective range
  - Inputs larger than size 5000 take very long

Sorting

- Searching for a value
  - Unsorted array — linear scan, O(n)
  - Sorted array — binary search, O(log n)
- Other advantages of sorting
  - Finding median value: midpoint of sorted list
  - Checking for duplicates
  - Building a frequency table of values

How to sort?

- You are a Teaching Assistant for a course
- The instructor gives you a stack of exam answer papers with marks, ordered randomly
- Your task is to arrange them in descending order

Strategy 1

- Scan the entire stack and find the paper with minimum marks
- Move this paper to a new stack
- Repeat with remaining papers
  - Each time, add next minimum mark paper on top of new stack
- Eventually, new stack is sorted in descending order

Strategy 1 ...

21 **32** **55** **64** **74** **89**

Strategy 1 ...

Selection Sort

- Select the next element in sorted order
- Move it into its correct place in the final sorted list

Selection Sort

- Avoid using a second list
  - Swap minimum element with value in first position
  - Swap second minimum element to second position
  - ...

Selection Sort

Selection Sort

```
def SelectionSort(l):

    # Scan slices l[0:len(l)], l[1:len(l)], ...
    for start in range(len(l)):

        # Find minimum value in slice . . .
        minpos = start
        for i in range(start,len(l)):
            if l[i] < l[minpos]:
                minpos = i

        # . . . and move it to start of slice
        (l[start],l[minpos]) = (l[minpos],l[start])
```

Analysis of Selection Sort

- Finding minimum in unsorted segment of length k requires one scan, k steps
- In each iteration, segment to be scanned reduces by 1
- $T(n) = n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2 = O(n^2)$

```
madhavan@dolphinair:...eek3/python/selectionsort$ more selectionsort.py
def SelectionSort(l):
  # Scan slices l[0:len(l)], l[1:len(l)], ...
  for start in range(len(l)):
    # Find minimum value in slice . . .  minpos = start
    for i in range(start,len(l)):
        minpos = start
        if l[i] < l[minpos]:
            minpos = i
            # .. and move it to start of slice
            (l[start],l[minpos]) = (l[minpos],l[start])
madhavan@dolphinair:...eek3/python/selectionsort$ 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 selectionsort import *
>>> l = [3,7,2]
>>> SelectionSort(l)
>>> l
[2, 3, 7]
>>> l = list(range(500,0.-1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
>>> l = list(range(500,0.-1))
```

How to sort?

- You are a Teaching Assistant for a course
- The instructor gives you a stack of exam answer papers with marks, ordered randomly
- Your task is to arrange them in descending order

Strategy 2

- First paper: put in a new stack
- Second paper:
  - Lower marks than first? Place below first paper
    Higher marks than first? Place above first paper
- Third paper
  - Insert into the correct position with respect to first two papers

Strategy 2

- First paper: put in a new stack
- Second paper:
  - Lower marks than first? Place below first paper
    Higher marks than first? Place above first paper
- Third paper
  - Insert into the correct position with respect to first two papers
- Do this for each subsequent paper:
  insert into correct position in new sorted stack

Strategy 2 …

21 **32** **55** **64** **74** **89**

Strategy 2 ...

Insertion Sort

- Start building a sorted sequence with one element
- Pick up next unsorted element and insert it into its correct place in the already sorted sequence

Insertion Sort

```
def InsertionSort(seq):

  for sliceEnd in range(len(seq)):
    # Build longer and longer sorted slices
    # In each iteration seq[0:sliceEnd] already sorted

    # Move first element after sorted slice left
    # till it is in the correct place
    pos = sliceEnd
    while pos > 0 and seq[pos] < seq[pos-1]:
      (seq[pos],seq[pos-1]) = (seq[pos-1],seq[pos])
      pos = pos-1
```

Insertion Sort


```
def InsertionSort(seq):

 for sliceEnd in rang                    )):
   # Build longer and l                    slices
   # In each iteration                     nd] already sorted

   # Move first element aft                orted slice left
   # till it is in the correct place
   pos = sliceEnd
   while pos > 0 and seq[pos] < seq[pos-1]:
     (seq[pos],seq[pos-1]) = (seq[pos-1],seq[pos])
     pos = pos-1
```

Insertion Sort

Analysis of Insertion Sort

- Inserting a new value in sorted segment of length k requires upto k steps in the worst case
- In each iteration, sorted segment in which to insert increased by 1
- $T(n) = 1 + 2 + \ldots + n-1 = n(n-1)/2 = O(n^2)$

```
>>> l = list(range(0,5000))
>>> InsertionSort(l)
>>> l = list(range(0,100000))
>>>
```

Inductive definitions

Many arithmetic functions are naturally defined inductively

- Factorial
  - $0! = 1$
  - $n! = n \times (n-1)!$

Inductive definitions

Many arithmetic functions are naturally defined inductively

- Factorial
  - $0! = 1$
  - $n! = n \times (n-1)!$
- Multiplication — repeated addition
  - $m \times 1 = m$
  - $m \times (n+1) = m + (m \times n)$

Inductive definitions …

- Define one or more base cases
- Inductive step defines f(n) in terms of smaller arguments

Inductive definitions …

- Define one or more base cases
- Inductive step defines f(n) in terms of smaller arguments

Fib : 1, 1, 2, 3, 5, 8

F$_i$:

Recursive computation

- **Inductive definitions naturally give rise to recursive programs**

```
def factorial(n):
    if n == 0:
        return(1)
    else:
        return(n * factorial(n-1))
```

Recursive computation

- **Inductive definitions naturally give rise to recursive programs**

```
def multiply(m,n):
    if n == 1:
        return(m)
    else:
        return(m + multiply(m,n-1))
```

$m \cdot 1 = m$

$m \cdot n = m + m(n-1)$

Inductive definitions for lists

- Lists can be decomposed as
  - First (or last) element
  - Remaining list with one less element

Inductive definitions for lists

- Lists can be decomposed as
  - First (or last) element
  - Remaining list with one less element
- Define list functions inductively
  - Base case: empty list or list of size 1
  - Inductive step: f(l) in terms of smaller sublists of l

Inductive definitions for lists

- **Length of a list**

```
def length(l):
    if l == []:
        return(0)
    else:
        return(1 + length(l[1:])
```

Inductive definitions for lists

- Sum of a list of numbers

$[x_1, x_2, \ldots x_n]$

$x_1 + [x_2, \ldots x_n]$

```
def sumlist(l):
    if l == []:
        return(0)
    else:
        return(l[0] + sumlist(l[1:]))
```

Recursive insertion sort

- **Base case**: if list has length 1 or 0, return the list
- Inductive step:
  - Inductively sort slice `l[0:len(l)-1]`
  - Insert `l[len(l)]` into this sorted slice

Recursive insertion sort

```
def InsertionSort(seq):
  isort(seq,len(seq))

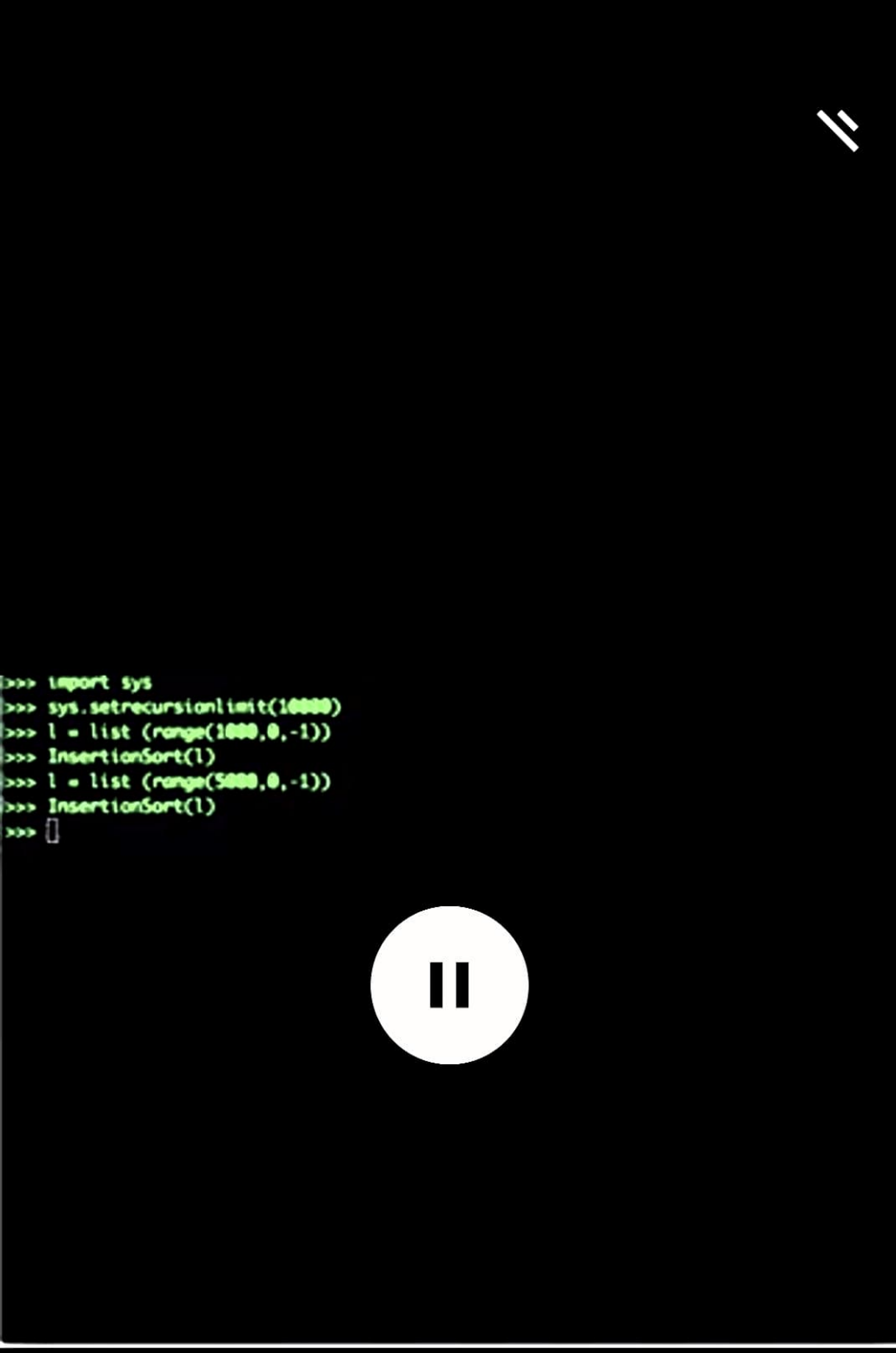
def isort(seq,k): # Sort slice seq[0:k]
  if k > 1:
    isort(seq,k-1)
    insert(seq,k-1)

def insert(seq,k): # Insert seq[k] into sorted seq[0:k-1]
  pos = k
  while pos > 0 and seq[pos] < seq[pos-1]:
    (seq[pos],seq[pos-1]) = (seq[pos-1],seq[pos])
    pos = pos-1
```

```
[illegible]:...jul/week3/python/isortrec$ more isortrec.py
def InsertionSort(seq):
    isort(seq,len(seq))

def isort(seq,k):
    if k > 1:
        isort(seq,k-1)
        insert(seq,k-1)

def insert(seq,k):
    pos = k
    while pos > 0 and seq[pos] < seq[pos-1]:
        (seq[pos],seq[pos-1]) = (seq[pos-1],seq[pos])
        pos = pos-1
[illegible]:...jul/week3/python/isortrec$
```


>>> import sys
>>> sys.setrecursionlimit(10000)
>>> l = list (range(1000,0,-1))
>>> InsertionSort(l)
>>> l = list (range(5000,0,-1))
>>> InsertionSort(l)
>>>

14:49 / 17:58

Recursion limit in Python

- Python sets a recursion limit of about 1000

```
>>> l = list(range(1000,0,-1))
>>> InsertionSort(l)
. . .
RecursionError: maximum recursion depth
exceeded in comparison
```

- Can manually raise the limit

```
>>> import sys
>>> sys.setrecursionlimit(10000)
```