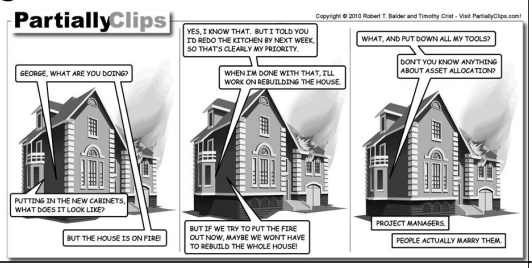
Sort Procedures and Quicker Sorting

One-Slide Summary

- **g is in O(f)** iff there exist positive constants c and n_0 such that $g(n) \leq cf(n)$ for all $n \geq n_0$.
- If g is in O(f) we say that f is an **upper bound** for g.
- We use **Omega Ω** for lower bounds and **Theta Θ** for tight bounds.
- Knowing a running time is in O(f) tells you that the running time is *not worse than f*. This can only be good news.
- Some way to **sort** have different running times.

Outline

- Finish up "Magic"
- Administrivia: your views, voting
- Sorting: timing and costs
- Insertion Sort
- Better sorting?
- End early?



Exam 1

- Handed out at end of class on **Thu Oct 04**, due at the beginning of class **Thu Oct 11**
  - You have one week, should take 2-4 hours
- Open Book - ***No PyCharm / Udacity Python***
- Open TAs & Profs - ***No Friends***
- Covers everything through Tue Oct 02 including:
  - Lectures 1-11, Book Chapters 1-8 (etc.), PS 1-4
- Post on the forum if you want a review session

Time On Problem Sets: The Bad

- *"I believe this PS3 has taken me over 10 hours to complete, not including reading for class."*
- *"I'd say this lab took around a total of just over 3 hours for me, which is not too bad I suppose."*

- Some people mentioned that they found the PS long: 10 hours was the max mentioned time.
  - "One credit of laboratory work can equal one to four hours per week." - UVA Registrar http://www.virginia.edu/registrar/about.html
  - "a 3 hour course requires about 10 hours/week for the entire semester." - UVA Kinesiology http://records.uva.acalog.com/preview_program.php?catoid=11&poid=1052&bc=1
  - "a ratio of 4 clock hours per credit hour per week." - UVA Clinical Nursing http://www.nursing.virginia.edu/media/NEW%20Student%20Handbook%20CNL%2007-08.pdf
  - "Total contact hours for a course should account for readings, online time, outside preparation and study. Total contact hours required per credit hour are as follows: 135 hours for a 3-credit course [9 hours a week for 15 weeks]." - UVA Syllabus Template http://www.faculty.virginia.edu/bbcp/documents/Final_Syllabus_Template.doc

Time On Problem Sets: The Good

- *"At least, personally, I could not have done this PS without their help. Is that really what the problem sets are supposed to be?"*

- PS3 is one of the two hardest problem sets. Remember, you are not expected to know or do it all.
  - 86% of you had perfect coding scores on PS3. 89% on PS2. You may be working too hard!
- PS Design: **Open-Ended Grading, not Rote!**
  - Final problems allow us to distinguish between superstars: currently you are all superstars!
  - Example: Skipping convert_lcommands_to_curvelist on PS3: 21/23
  - **Course curve**: An "A" does not require perfect PS

Tutoring and Hints

- *"Is there any way to get one on one tutoring for this type of problem set?"*

- In the past, the ACM and ACM-W have offered one-on-one tutoring. Send me (or the course staff) email if you are interested; I will try to set something up.
  - *"More hints written into PS if possible please? This way I can work on it independently of TAs"*
- I will add more hints on a optional links for PS4 on. On your honor!

Recall: Asymptotic Complexity

g is in $O(f)$ iff: There are positive constants c and n_0 such that $g(n) \leq cf(n)$ for all $n \geq n_0$.

g is in $\Omega(f)$ iff: There are positive constants c and n_0 such that $g(n) \geq cf(n)$ for all $n \geq n_0$.

g is in $\Theta(f)$ iff: g is in $O(f)$ and g is in $\Omega(f)$.

Is our sort good enough?

Takes over 1 second to sort 1000-length list. How long would it take to sort 1 million items?

1s = time to sort 1000
4s ~ time to sort 2000

1M is 1000 * 1000

Sorting time is n^2
so, sorting 1000 times as many items will take
1000^2 times as long = 1 million seconds ~ 11 days

Note: there are 800 Million VISA cards in circulation.
It would take 20,000 years to process a VISA transaction at this rate.

Which of these is true?

- Our sort procedure is too slow for VISA because its running time is in $O(n^2)$
- Our sort procedure is too slow for VISA because its running time is in $\Omega(n^2)$
- Our sort procedure is too slow for VISA because its running time is in $\Theta(n^2)$

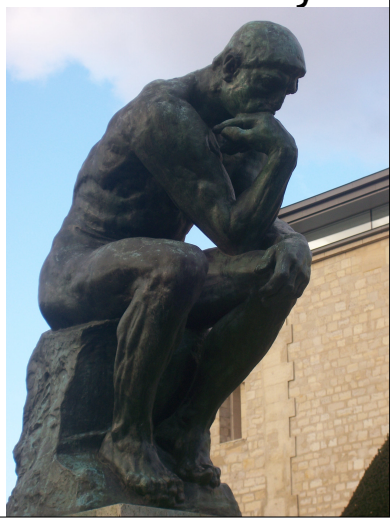
Which of these is true?

- ~~Our sort procedure is too slow for VISA because its running time is in $O(n^2)$~~
- Our sort procedure is too slow for VISA because its running time is in $\Omega(n^2)$
- Our sort procedure is too slow for VISA because its running time is in $\Theta(n^2)$

Knowing a running time is in $O(f)$ tells you the running time is not *worse* than f. This can *only* be good news. It doesn't tell you anything about how bad it is. *(Lots of people and books get this wrong.)*

Liberal Arts Trivia: Art History

- Name the work shown and its sculptor. The artist is generally considered the progenitor of modern sculpture: he departed from mythology and allegory and modeled the human body with realism, celebrating individual character and physicality.



Liberal Arts Trivia: Chinese History

- This period of Chinese history roughly corresponds to the Eastern Zhou dynasty (8th century BCE to 5th century BCE). China was feudalistic, with Zhou kings controlling only the capital (Luoyang) and granting the rest as fiefdoms to several hundred nobles (including the Twelve Princes). As the era unfolded, powerful states annexed smaller ones until a few large principalities controlled China. By 6th century BCE, the feudal system had crumbled and the Warring States period had begun.

Sorting Cost

```
def bf_sort(lst, cf): # simple sort
  if not lst: return []
  best = find_best(lst, cf)
  return [best] + bf_ort(remove(lst, best), cf)
def find_best(things, better):
   if len(things) == 1: return things[0]
   return pick_better(better, things[0], \
     find_best(things[1:], better)
```

The running time of best first sort is in $\Theta(n^2)$ where n is the number of elements in the input list.

Assuming the comparison function passed as *cf* has constant running time.

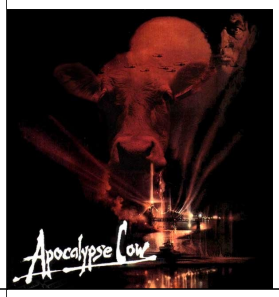
Divide and Conquer sorting?

- **Best first sort**: find the lowest in the list, add it to the front of the result of sorting the list after deleting the lowest.

- **Insertion sort**: insert the first element of the list in the right place in the sorted rest of the list.
  - Let's write this together on the next slide!
  - Hint: use/write helper function insert_one
  - insert_one(2, [1,3,4], ascend) --> [1,2,3,4]

insert-sort

```
def insert_sort(lst, cf):
  if not lst: return []
  return insert_one(lst[0], insert_sort(lst[1:], cf))
```



Try writing insert_one.
def **insert_one**(elt, lst, cf):
    ...

>>> insert_one(2, [1,3,5], ascend)
[1,2,3,5]

insert_one

```
def insert_one(elt, lst, cf):
  if not lst: return [elt] # careful!
  if cf(elt, lst[0]):      # are we there yet?
    return [elt] + lst     # yes!
  else:                    # no, keep going!
    return [lst[0]] + insert_one(elt, lst[1:], cf)
```

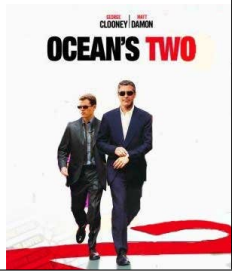
insert_one(3, [1,2,4,5], cf) ->
[1] + insert_one(3, [2,4,5], cf) ->
[1] + [2] + insert_one(3, [4,5], cf) ->
[1] + [2] + [3] + [4,5] ->
[1,2,3,4,5]

How much work is insert_sort?

```
def insert_sort(lst, cf):
  if not lst: return []
  return insert_one(lst[0], insert_sort(lst[1:], cf))

def insert_one(elt, lst, cf):
  if not lst: return [elt]
  if cf(elt, lst[0]):
    return [elt] + lst
  else:
    return [lst[0]] + \
      insert_one(elt, lst[1:], cf)
```

How many times does insert_sort evaluate insert_one?



How much work is insert-sort?

```
def insert_sort(lst, cf):
  if not lst: return []
  return insert_one(lst[0], insert_sort(lst[1:], cf))

def insert_one(elt, lst, cf):
  if not lst: return [elt]
  if cf(elt, lst[0]): return [elt] + lst
  return [lst[0]] + insert_one(elt, lst[1:], cf)
```

How many times does insert-sort evaluate insert-one?

n times (once for each element)

running time of insert-one is ?

How much work is insert-sort?

```
def insert_sort(lst, cf):
  if not lst: return []
  return insert_one(lst[0], insert_sort(lst[1:], cf))

def insert_one(elt, lst, cf):
  if not lst: return [elt]
  if cf(elt, lst[0]): return [elt] + lst
  return [lst[0]] + insert_one(elt, lst[1:], cf)
```

How many times does insert-sort evaluate insert-one?

n times (once for each element)

running time of insert-one is in $\Theta(n)$

How much work is insert-sort?

```
def insert_sort(lst, cf):
  if not lst: return []
  return insert_one(lst[0], insert_sort(lst[1:], cf))

def insert_one(elt, lst, cf):
  if not lst: return [elt]
  if cf(elt, lst[0]): return [elt] + lst
  return [lst[0]] + insert_one(elt, lst[1:], cf)
```

How many times does insert-sort evaluate insert-one?

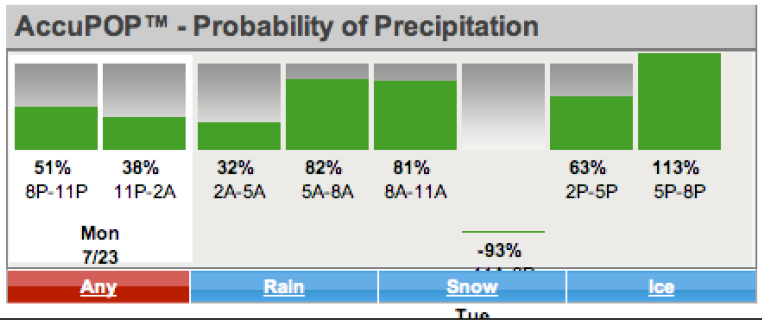
n times (once for each element)

running time of insert-one is in $\Theta(n)$

insert-sort has running time in $\Theta(n^2)$ where n is the number of elements in the input list

Which is better?

- Is insert_sort faster than best_first_sort?




```
>>> def reverse(lst): return lst[::-1]
>>> insert_sort(reverse(intsto(20), ascending)
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Requires 190 applications of <

```
>>> insert_sort(intsto(20), ascending)
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Requires 19 applications of <

```
>>> insert_sort(random_order_int_list_20, ascending)
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Requires 104 applications of <

```
>>> best_first_sort(intsto(20), ascending)
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Requires 210 applications of <

```
>>> best_first_sort(random_order_int_list_20,
ascending)
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Requires 210 applications of <

best-first-sort vs. insert-sort

- Both are $\Theta(n^2)$ worst case (reverse list)
- Both are $\Theta(n^2)$ when sorting a randomly ordered list
  - But insert-sort is about twice as fast
- insert-sort is $\Theta(n)$ best case (ordered input list)

Can we do better?

insert_one(88, [1,2,3,5,6,22,63,77,89,90], ascending)

Suppose we had procedures
first_half(lst)
second_half(lst)
that *quickly* divided the list in two halves?

quicker-insert using halves

```
def quicker_insert(elt, lst, cf):
  if not lst: return [elt]     # just like insert_one
  if len(lst) == 1:            # handle 1 element by hand
    return [elt]+lst   if cf(elt, lst[0])   else lst+[elt]
  front = first_half(lst)
  back = second_half(lst)
  if cf(elt, back[0]):         # insert into front half
    return quicker_insert(elt, front, cf) + back
  Else:                        # insert into back half
    return front + quicker_insert(elt, back, cf)
```


Evaluating quicker-sort

```
>>> quicker_insert(3, [1,2,4,5,7,8,9,10], ascend)
Front = [1,2,4,5]
Back = [7,8,9,10]
Is 3 < 7? Yes. So
return quicker_insert(3,[1,2,4,5],ascend) + [7,8,9,10]
Front = [1,2]
Back = [4,5]
Is 3 < 4? Yes. So
return quicker_insert(3,[1,2],ascend) + [4,5]
Front = [1]
Back = [2]
Is 3 < 2? No. So
Return [1] + quicker_insert(3,[2],ascend)
One element. Compare 3 and 2, return [2,3]

So final result is:
[1] + [2,3] + [4,5] + [7,8,9,10]
```

```
def quicker_insert(elt, lst, cf):
  if not lst: return [elt]          # just like insert_one
  if len(lst) == 1:                 # handle 1 element
    return [elt]+lst   if cf(elt, lst[0])   else lst+[elt]
  front = first_half(lst)
  back = second_half(lst)
  if cf(elt, back[0]):    # insert into front half
    return quicker_insert(elt, front, cf) + back
  else:                   # insert into back half
    return front + quicker_insert(elt, back, cf)
```

Every time we call quicker-insert, the length of the list is approximately **halved**!

How much work is quicker-sort?

Each time we call quicker-insert, the size of lst halves. So doubling the size of the list only increases the number of calls by 1.

```
def quicker_insert(elt, lst, cf):
  if not lst: return [elt]          # just like insert_one
  if len(lst) == 1:                 # handle 1 element
    return [elt]+lst   if cf(elt, lst[0])   else lst+[elt]
  front = first_half(lst)
  back = second_half(lst)
  if cf(elt, back[0]):    # insert into front half
    return quicker_insert(elt, front, cf) + back
  else:                   # insert into back half
    return front + quicker_insert(elt, back, cf)
```

List Size	# quicker_insert applications
1	1
2	2
4	3
8	4
16	5
32	6

Homework

- **Problem Set 4**
- **Read Chapter 8 or Udacity 5**
- Exam 1 Out Soon