Quiz 2 problems

- 1. short answers

```
>>> a = [2]
>>> b = [a, a]
>>> a.append(3)
>>> b
```

____

- 2. sorting --
  translate between 3 ways of sorting and compare them
- 3. number of interleavings -- memoization
- 4. weird quicksort again -- binary search tree operations
- 5. related to the DFS in HW1

Basic Sorting

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> print a
[1, 2, 3, 4, 5]

>>> a = [5, 2, 3, 1, 4]
>>> a.sort(reverse=True)
>>> a
[5, 4, 3, 2, 1]

>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a.reverse()
>>> a
[5, 4, 3, 2, 1]
```

sort() is in-place,
but sorted() returns new copy

```
>>> a = [5, 2, 3, 1, 4]
>>> sorted(a)
[1, 2, 3, 4, 5]
>>> a
[5, 2, 3, 1, 4]
```

Built-in and custom cmp

```
>>> a = [5, 2, 3, 1, 4]
>>> def mycmp(a, b):
        return b-a

>>> sorted(a, mycmp)
[5, 4, 3, 2, 1]

>>> sorted(a, lambda x,y: y-x)
[5, 4, 3, 2, 1]

>>> a = zip([1,2,3], [6,4,5])
>>> a.sort(lambda (_,y1), (__, y2): y1-y2)
>>> a
[(2, 4), (3, 5), (1, 6)]
>>> a.sort(lambda (_,y1), (_, y2): y1-y2)
SyntaxError: duplicate argument '_' in function definition
```

demo

Sorting by Keys or Key mappings

```
>>> a = "This is a test string from Andrew".split()
>>> a.sort(key=str.lower)
>>> a
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']

>>> import operator
>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3), ('b', 1)]

>>> L.sort(key=operator.itemgetter(1))
>>> L
[('d', 1), ('b', 1), ('c', 2), ('b', 3), ('a', 4)]
```

demo

```
>>> sorted(L, key=operator.itemgetter(1, 0))
[('b', 1), ('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

sort by two keys

```
>>> operator.itemgetter(1,0)((1, 2, 3))
(2, 1)
```

lambda for key mappings

- you can use lambda for both custom cmp and key map

```
>>> a = "This is a test string from Andrew".split()
>>> a.sort(lambda x, y: cmp(x.lower(), y.lower()))
>>> a
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']

>>> a.sort(key=lambda x: x.lower())

>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3), ('b', 1)]

>>> L.sort(key=lambda (_, y): y)
>>> L
[('d', 1), ('b', 1), ('c', 2), ('b', 3), ('a', 4)]

>>> sorted(L, key=lambda (x, y): (y, x))
[('b', 1), ('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

Decorate-Sort-Undecorate

```
>>> words = "This is a test string from Andrew.".split()

>>> deco = [ (word.lower(), i, word) for i, word in \
... enumerate(words) ]

>>> deco.sort()

>>> new_words = [ word for _, _, word in deco ]

>>> print new_words
['a', 'Andrew.', 'from', 'is', 'string', 'test', 'This']
```

demo

- Most General
- Faster than custom cmp (or custom key map) -- why?
- stable sort (by supplying index)

Sorting: Summary

- 3 ways: key mapping, custom cmp function, decoration
- decoration is most general, key mapping least general
- decoration is faster than key mapping & cmp function
  - decoration only needs O(n) key mappings
  - other two need O(nlogn) key mappings -- or O(n^2) for insertsort
  - real difference when key mapping is slow
- decoration is stable

Memoized Recursion v1

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib (n-1) + fib (n-2)
```

```
fibs = {0:1, 1:1}
def fib(n):
    if n in fibs:
        return fibs[n]
    fibs[n] = fib(n-1) + fib(n-2)
    return fibs[n]
```

can we get rid of the global variable?

Memoized Recursion v2

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib (n-1) + fib (n-2)
```

```
def fib(n, fibs={0:1, 1:1}):
    if n not in fibs:
        fibs[n] = fib(n-1, fibs) + fib(n-2, fibs)
    return fibs[n]
```

Memoized Recursion v3

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
>>> fib(3)
1 {1: 1}
0 {0: 1, 1: 1}
2 {0: 1, 1: 1, 2: 2}
3 {0: 1, 1: 1, 2: 2, 3: 3}
3
>>> fib(2)
2
>>> print fibs
Error!
```

draw the tree!

```
def fib(n, fibs={0:1, 1:1}):
    if n not in fibs:
        fibs[n] = fib(n-1) + fib(n-2)
#       print n, fibs
    return fibs[n]
```

the **fibs** variable has a weird closure!! feature or bug?

most people think it’s a bug, but Python inventor argues it’s a feature.

Memoized Recursion v4

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
>>> fib(4)
{0: 1, 1: 1, 2: 2}
{0: 1, 1: 1, 2: 2, 3: 3}
{0: 1, 1: 1, 2: 2, 3: 3, 4: 5}
5
>>> fib(3)
{0: 1, 1: 1, 2: 2}
{0: 1, 1: 1, 2: 2, 3: 3}
3
```

```
def fib(n, fibs=None):
    if fibs is None:
        fibs = {0:1, 1:1}
    if n not in fibs:
        fibs[n] = fib(n-1, fibs) + fib(n-2, fibs)
#       print n, fibs
    return fibs[n]
```

this is so far the cleanest way to avoid this bug.

HW 1 - interleave in 6, 4, 3 lines

```
def inter(a, b, c=[]):
  if a == []:
    return [c + b]
  if b == []:
    return [c + a]
  return inter(a[1:], b, c+[a[0]]) + inter(a, b[1:], c+[b[0]])
```

(Ezq)

```
def inter(a, b, c=[]):
  if a == [] or b == []:
    return [c + a + b]
  return inter(a[1:], b, c+[a[0]]) + inter(a, b[1:], c+[b[0]])
```

```
def inter(a, b, c=[]):
  return [c + a + b] if a == [] or b == [] else \
    inter(a[1:], b, c+[a[0]]) + inter(a, b[1:], c+[b[0]])
```

Improve it! Memoized interleave

```
def inter2(a, b, c=[], inters={}):
   key = tuple(a), tuple(b)
   if key not in inters:
     inters[key] = [c + a + b] if a == [] or b == [] else \
       inter2(a[1:], b, c+[a[0]]) + inter2(a, b[1:], c+[b[0]])
   return inters[key]
```

lists, being mutable, can not be hashed! (even recursively)

```
>>> {(1, [2]): 2}
TypeError: unhashable type: 'list'
```

Thomas: # of interleavings
= C(n+m, n)
= C(n+m, m)

```
>>> len(inter2(range(13),range(13)))
10400600
>>> len(inter(range(13),range(13)))
...
```

```
but still exponential time and space complexities!
what if we only want the number of interleavings?
how fast could that be?
```

Mutable types are not hashable

- mutables: list, dict, set
- immutables: tuple, string, int, float, frozenset, ...
  - only recursively immutable objects are hashable
- your own class objects are hashable (but be careful...)

```
>>> {{1}: 2}
TypeError: unhashable type: 'set'

>>> {{1:2}: 2}
TypeError: unhashable type: 'dict'

>>> {frozenset([1]): 2}
{frozenset([1]): 2}

>>> {frozenset([1, [2]]): 2}
TypeError: unhashable type: 'list'
```

HW 1 -- introduce

```
def dfs(v, vertices, adjacencies):
    vertices[v] = True
    for w in sorted(adjacencies[v]):
        if not vertices[w]:
            print "%s introduces %s." % (v, w)
            dfs(w, vertices, adjacencies)

if __name__ == "__main__":
    lines = stdin.readlines()

    vertices = dict([(v, False) for v in lines[0].split()])
    adjacencies = defaultdict(list)

    for line in lines[1:]:
        line = line.split()
        adjacencies[line[0]].append(line[1])
        adjacencies[line[1]].append(line[0])

    totalgroups = singletons = 0
    for v in sorted(vertices):
        if not vertices[v]:
            totalgroups += 1
            print "The instructor introduces %s." % v
            if not adjacencies[v]:
                singletons += 1
            else:
                dfs(v, vertices, adjacencies)
```

(Ezq)

```
if not w in visited:
```

```
visited = {}  # set!
```

```
x, y = line.split()
adjacencies[x].append(y)
adjacencies[y].append(x)
```

```
if not adjacencies[v ] == []:
```

HW 1 -- word frequencies

(Kareem)

```
#!/usr/bin/env python
__author__ = "Kareem Francis"
import sys
from collections import defaultdict

def words(text_file=sys.stdin):
    '''
    Given a file object (stdin by default), computes the frequency and line
    numbers on which each word in the file has occured.
    Format: '<frequency> <word> <line appearances>'
    '''
    word_count = defaultdict(int)
    line_appearances = defaultdict(list)
    for i, line in enumerate(text_file, 1):
        row = line.strip().split()
        for word in row:
            word = word.lower()
            word_count[word]+=1
            if not i in line_appearances[word]:
                line_appearances[word].append(i)
    for word, _ in sorted(word_count.items(), key=lambda(k, v): (-v, k)):
        print word_count[word], word, ' '.join(map(str,line_appearances[word]))

if __name__ == '__main__':
    words()
```

should combine into one hash

no need to strip before split()

slow! should use set not list!

my solution using defaultdict

- Counting Word Frequencies
  - read in a text file, count the frequencies of each word, and print in descending order of frequency

```
import sys
from collections import defaultdict

if __name__ == '__main__':
    wordlist = defaultdict(set)
    for i, line in enumerate(sys.stdin, 1):
        for word in line.split():
            wordlist[word].add(i)

    sortedlist = sorted([(-len(lines), word, lines) \
                 for (word, lines) in wordlist.items()])

    for freq, word, lines in sortedlist:
        print -freq, word, " ".join(map(str, sorted(lines)))
```

input

```
Python is a cool language but OCaml

is even cooler since it is purely functional
```

output

```
3 is 1 2
1 a 1
1 but 1
1 cool 1
1 cooler 2
1 even 2
1 functional 2
1 it 2
1 language 1
1 OCaml 1
1 purely 2
1 Python 1
1 since 2
```

shorter, but... wrong!

my corrected solution

- Counting Word Frequencies
  - read in a text file, count the frequencies of each word, and print in descending order of frequency

```
import sys
from collections import defaultdict

if __name__ == '__main__':
    wordlist = defaultdict(list)
    for i, line in enumerate(sys.stdin, 1):
        for word in line.split():
            wordlist[word].append(i)

    sortedlist = sorted([(-len(lines), word, lines) \
                  for (word, lines) in wordlist.items()])

    for freq, word, lines in sortedlist:
        print -freq, word, " ".join(map(str, sorted(set(lines))))
```

input

```
Python is a cool language but OCaml

is even cooler since it is purely functional
```

output

```
3 is 1 2
1 a 1
1 but 1
1 cool 1
1 cooler 2
1 even 2
1 functional 2
1 it 2
1 language 1
1 OCaml 1
1 purely 2
1 Python 1
1 since 2
```

HW 1 -- word frequencies

(Ezq)

```
#!/usr/bin/env python
from sys import stdin
from operator import itemgetter
from collections import defaultdict

if __name__ == "__main__":
    d = defaultdict(lambda : {
        "count": 0,
        "lines": set()
    })

    for line, words in enumerate(stdin, 1):
        for word in words.lower().split():
            d[word]["count"] += 1
            d[word]["lines"].add(str(line))

    items = [(str(d[word]["count"]), word, \
        " ".join(sorted(d[word]["lines"]))) for word in d]

    items.sort(key = itemgetter(1))
    items.sort(key = itemgetter(0), reverse = True)

    print "\n".join([" ".join([count, word, lines]) \
        for count, word, lines in items])
```

good design: readable

shoud not sort twice!

HW 1 - mergesort & quickselect

```
def mergelists2(a, b):
    if a == [] or b == []:
        return b if a == [] else a
    if a[0] <= b[0]:
        return [a[0]] + mergelists2(a[1:], b)
    else:
        return [b[0]] + mergelists2(a, b[1:])
```

stable!

(Ezq)

```
from random import randint
def quickselect(a, k):
    if a == []:                            # do not write “not a”
        return a
    pivot = a[randint(0, len(a) - 1)]
    left = [i for i in a if i < pivot]
    ith = len(left) + 1
    if k < ith:
        return quickselect(left, k)
    if k > ith:
        return quickselect([i for i in a if i > pivot], k - ith)
    return pivot
```

Implementation

- lists, tuples, and strings
  - random access: O(1)
  - insertion/deletion/in: O(n)
- dict
  - in/random access: almost O(1)
  - insertion/deletion: almost O(1)
  - but no linear ordering!

Pythonic Styles

do not write ...	when you can write ...
`for key in d.keys():`	`for key in d:`
`if d.has_key(key):`	`if key in d:`

`i = 0`
`for x in a:`
`    ...`
`    i += 1`
	`for i, x in enumerate(a):`

`a[0:len(a) - i]`	`a[:-i]`

`for line in \`
`    sys.stdin.readlines():`
	`for line in sys.stdin:`

`for x in a:`
`    print x,`
`print`
	`print " ".join(map(str, a))`

`s = ""`
`for i in range(lev):`
`    s += " "`
`print s`
	`print " " * lev`