

Lecture 7: Algorithms

CS442: Great Insights in Computer Science
Michael L. Littman, Spring 2006

Python Tutorial

- Although you don't need to learn to program in this class, I'd like you to be able to read a simple program to see what it does.
- I had been assuming you'd learn Python by osmosis.
- Last time I got a lot of good questions about Python, so I thought you deserved a more complete description.

Variables and Strings

<code>print "hello"</code>	<code>hello</code>
<code>it = "hello"</code>	
<code>print it</code>	<code>hello</code>
<code>print "it"</code>	<code>it</code>
<code>print 'it' + " " + it</code>	<code>it hello</code>
<code>x = "tuna"</code>	
<code>y = "fish"</code>	
<code>print x</code>	<code>tuna</code>
<code>print x + y</code>	<code>tunafish</code>
<code>print x + " " + y</code>	<code>tuna fish</code>
<code>print "x + y"</code>	<code>x + y</code>

Subroutines

```
def d(x):  
    print x + " are delicious"
```

<code>d("salmon")</code>	<code>salmon are delicious</code>
<code>d(x)</code>	<code>tuna are delicious</code>
<code>d(y)</code>	<code>fish are delicious</code>
<code>d(x+y)</code>	<code>tunafish are delicious</code>
<code>d("x+y")</code>	<code>x+y are delicious</code>

Functions

```
def s(y):  
    return "fried " + y
```

```
print s("potatoes")    fried potatoes  
print s(x)             fried tuna  
d(s("eggs"))          fried eggs are delicious  
print s(x) + s(y)     fried tunafried fish
```

Lists

```
z = ["Paul", "George",  
     "Ringo", "John"]  
print z                ['Paul', 'George', 'Ringo', 'John']  
print z[0]             Paul  
print z[3]             John  
print z[1:]            ['George', 'Ringo', 'John']  
print z + ["Stuart", "Billy"] ['Paul', 'George', 'Ringo', 'John',  
                                'Stuart', 'Billy']  
print len(z)           4  
print range(4)         0, 1, 2, 3
```

Strings as Lists

```
print x          tuna
print x[0]      t
print x[1:]     una
print (s(y))[1:8] ried fi
def reverse(x):
    if x == "": return ""
    return reverse(x[1:])+x[0]

reverse("swordfish")  hsifdrows
del z[2]
print z               ['John', 'Paul', 'Ringo']
```

Numbers

```
print 1+1, 2-2, 3*3, 4/4, 10/3  2 0 9 1 3
print 1 + x                      <error>
print str(1) + x                  1tuna
def frac(x,y):
    print x/y, x-y*(x/y), "/", y

frac(1,3)                         0 1 / 3
frac(14,3)                        4 2 / 3
```

Loops

```
for b in z:
```

```
    print b + " was a Beatle."
```

Paul was a Beatle.

George was a Beatle.

Ringo was a Beatle.

John was a Beatle.

```
x = 1000; y = 1
```

```
while x > 1:
```

```
    y = y + 1; x = x / 2
```

```
print y
```

10

Today's Goal

- Last time we looked at different ways of writing programs to produce the same output (Macdonald #1, #2, and #3, for example).
- None was definitively better, except aesthetically.
- We'll look at another way of comparing programs...

Sock Matching

- Hillis begins Chapter 5 with an example.
- We've got a basketful of mixed up pairs of socks.
- We want to pair them up reaching into the basket as few times as we can.



Sock 'Ops

- `getSock()`: pulls a sock out of the basket and provides its value.
- `match(sock1, sock2)`: takes two socks and returns True if they match (and pairs them) and False otherwise.
- `replaceSock(sock)`: puts the given sock back in the laundry basket.
- `emptyBasket()`: returns True if the basket is empty and False if there are still more socks.

Sock Sorter #1

- Grab two socks.
- If they don't match, toss them back in the basket.
- Will this procedure ever work?
- Will it *always* work?

```
def sorter1():  
    x = getSock()  
    y = getSock()  
    if not match(x,y):  
        replaceSock(x)  
        replaceSock(y)
```

Measuring Performance

- Hillis asserts that the time-consuming part of this operation is reaching into the basket: `getSock()`.
- Let's say we have 50 pairs of socks.
- How many `getSock()` operations does `sorter1()` do?
- Min, max, average?
- 100 experiments:
 - mean: 5051.36
 - max: 7354
 - min: 2978

Sock Sorter #2

- Grab two socks.
- If they don't match, put one back and grab a replacement.
- Repeat until a match is found.
- Ever? Always? Min, max, average? Better/worse/same?

```
def sorter2():  
    x = getSock()  
    y = getSock()  
    while not match(x,y):  
        replaceSock(y)  
        y = getSock()
```

Analysis

- Roughly the same number of matching operations, but since we always hold onto one sock, roughly half the number of getSocks().
- When might this approach fail in the real world?
- Does sorter1() suffer from this difficulty?
- 100 experiments:
 - mean: 2571.77
 - max: 3779
 - min: 1606

Sock Sorter #3

- Grab two socks.
- If they don't match, toss one into a separate pile and get a new one.
- When a match is found, put the pile back into the basket.
- Min/Max/Mean?

```
def sorter3():  
    x = getSock()  
    y = getSock()  
    pile = []  
    while not match(x,y):  
        pile = pile + [y]  
        y = getSock()  
    for sock in pile:  
        replaceSock(sock)
```

Analysis

- Again, roughly half of the previous one.
- In both, we grab a random sock and go through the basket looking for its mate.
- This time, we never check the same sock twice.
- Once it's been checked, we can set it aside temporarily.
- 100 experiments:
 - mean: 1313.10
 - max: 1723
 - min: 994

Sock Sorter #4

- Make a pile.
- Grab a sock.
- Look for its mate in the pile.
- If found, shrink pile.
- If not, add to the pile.
- Min/Max/Mean?

```
def sorter4():  
    pile = []  
    while not emptyBasket():  
        x = getSock()  
        matched = False  
        for i in range(len(pile)):  
            if not matched and match(x,pile[i]):  
                matched = True  
                del pile[i]  
        if not matched:  
            pile = pile + [ x ]
```

Analysis

- Gets every sock exactly once!
- A bit of extra work keeping the pile in proper shape.
- Always precisely 100 getSocks()!
- How might this approach be considered less good than the previous approaches?

Lessons Learned

- If we have a notion of “time” (getSock() or number of statements executed), we can compare different algorithms based on the time they take.
- They really are different, so use good algorithms.
- I once redesigned a colleague’s algorithm and it ran in seconds where it used to take an hour.
- Hard to believe they solved the same problem...

Next Time

- Knowing which routine works best for 50 pairs of socks is nice, but not terrible general.
- Next, how do algorithms differ as the size of the input grows?
- read: Hillis Chapter 5 (“algorithms”)