CS221 Section 1: Extra Material

Ice Pasupat

September 28, 2014

1 Coin Payment Problem

Problem Suppose you have an unlimited supply of coins with values 2, 3, and 5 cents. How many ways can you pay for an item costing 12 cents?

1.1 What if the order matters?

Let $a(n)$ be the number of ways to write 12 as an ordered sum of 2, 3, or 5. If the first number is k, then the remaining numbers must sum to $n-k$, and there are $a(n-k)$ ways to do that. So we get the recurrence relation

$$a(n) = a(n-2) + a(n-3) + a(n-5)$$

The boundary cases are $a(n) = 0$ for $n < 0$ and $a(0) = 1$. So we can write a Python code like this:

```
In [1]: def a(n):
            if n < 0:
                return 0
            if n == 0:
                return 1
            return a(n-2) + a(n-3) + a(n-5)

In [2]: print a(12)
```

```
27
```

However, running $a(120)$ will take a very long time since we are repeatedly recalculating $a(n)$ for the same value of n. To make it run faster, we use **memoization**: keeping the computed results in a Python `dict` (a lookup table, similar to C++'s STL `map` / Java's `Map` / MATLAB's `containers.Map`)

```
In [3]: cache = {}
        def a(n):
            if n < 0:
                return 0
            if n == 0:
                return 1
            if n not in cache:
                cache[n] = a(n-2) + a(n-3) + a(n-5)
            return cache[n]

In [4]: print a(120)
```

```
1424300270363583033
```

Bonus Running $a(12000)$ will results in an error since Python only allows around 1000 layers of recursion. One hack is to cache the values of smaller n's first. When we do large n, we can just look for the answer in the cache without having to call the function recursively. (You don't have to worry about this in the homework, though.)

```
In [5]: for n in xrange(12000):
            a(n)
        print a(12000)
```

```
21259576275262079042706440175862153183787319671374714576436336611381643941924729344496668081213561556790
```

1.2 What if the order does not matter?

Since $5 + 5 + 2$ and $2 + 5 + 5$ are now the same, we should decide on a canonical representation of the sum. Let's enforce that the numbers must be in increasing order (so $5 + 5 + 2$ must be rewritten as $2 + 5 + 5$).

Let's consider the first number. It can be 2, 3, or 5. If we choose 2, then the remaining numbers can be anything from 2, 3, and 5. But if we choose, say, 3, then the remaining numbers must be 3 or 5, not 2. This means the choice of the first number restricts the set of numbers we can use. If $b(n, \{2, 3, 5\})$ is the number of ways to write n as a sum of 2, 3, or 5 in increasing order, we get

$$b(n, \{2, 3, 5\}) = b(n - 2, \{2, 3, 5\}) + b(n - 3, \{3, 5\}) + b(n - 5, \{5\})$$

Remember Percy's comment on "using the indices instead of the actual object"? We will use it here to simplify the code. Let $c = [2, 3, 5]$ be the list of all coin values. We redefine $b(n, i)$ to be the number of ways to write n as a sum of $c[i], c[i + 1], \ldots$ in increasing order. We get

$$b(n, i) = b(n - c[i], i) + b(n - c[i + 1], i + 1) + \cdots$$

The code is as follows:

```
In [6]: c = [2,3,5]
        cache = {}
        def b(n, i):
            if n < 0:
                return 0
            if n == 0:
                return 1
            if (n,i) not in cache:
                cache[(n,i)] = sum(b(n-c[j], j) for j in xrange(i, len(c)))
            return cache[(n,i)]
```

```
In [7]: print b(12, 0)
```

```
5
```

Indeed, the actual sums are

- $2 + 2 + 2 + 2 + 2 + 2$
- $2 + 2 + 2 + 3 + 3$
- $2 + 2 + 3 + 5$
- $2 + 5 + 5$
- $3 + 3 + 3 + 3$

Note that there are other, more efficient solutions as well.

2 Python

2.1 Basic Data Structures

The basic data structures include:

- **numbers** (`int` = integers and `float` = decimals)
- **strings** (`str` = strings and `unicode` = international strings)
  - Unlike C, you cannot change a character in strings.
  - Unlike Java, `a == b` actually works.
- **lists** (`list` = mutable array; you can change its content)
- **tuples** (`tuple` = immutable array; you cannot change its content)
- **sets** (`set` = set of unrepeated values)
- **dicts** (`dict` = lookup table mapping keys to values)
- `True`, `False` and `None`

See https://docs.python.org/2/library/stdtypes.html for more details.

2.2 Syntactic Sugar

2.2.1 List comprehension

```
In [8]: a = "Can I skip a CS221 homework? No dice!!"

In [9]: b = a.split()
        print b

['Can', 'I', 'skip', 'a', 'CS221', 'homework?', 'No', 'dice!!']

In [10]: c = [len(_) for _ in b]
         print c

[3, 1, 4, 1, 5, 9, 2, 6]
```

Note that _ is a valid variable name (usually for one-time use). An equivalent for-loop would be

```
In [11]: c = []
         for _ in b:
             c.append(len(_))
         print c

[3, 1, 4, 1, 5, 9, 2, 6]
```

You can also filter the results:

```
In [12]: c = [len(_) for _ in b if len(_) % 2 == 0]    # % is modulus (remainder after division)
         print c

[4, 2, 6]
```

2.2.2 List Slicing

```
In [13]: c = [len(_) for _ in b]
         print c

[3, 1, 4, 1, 5, 9, 2, 6]

In [14]: print c[2]    # The index starts from 0

4

In [15]: print c[-1]   # You can also count backward from the right: c[-1] is the last element
```

```
6
```

```
In [16]: print c[1:4]  # Include c[1] but Exclude c[4]
```

```
[1, 4, 1]
```

```
In [17]: print c[:4]   # = c[0:4]
```

```
[3, 1, 4, 1]
```

```
In [18]: print c[4:]   # = c[4:len(c)]
```

```
[5, 9, 2, 6]
```

```
In [19]: print c[:-1]
```

```
[3, 1, 4, 1, 5, 9, 2]
```

2.2.3 Generator

A **list** is concrete and you can use it many times. A list also uses up memory depending on how long the list is.

```
In [20]: c = [len(_) for _ in b]
         print c
```

```
[3, 1, 4, 1, 5, 9, 2, 6]
```

```
In [21]: print max(c)
```

```
9
```

```
In [22]: print min(c)
```

```
1
```

However, a **generator** will generate data for only one round. The values are computed on-the-fly, so nothing is stored in the memory.

```
In [23]: c = (len(_) for _ in b)   # Use parentheses
         print c
```

```
<generator object <genexpr> at 0x7fc56c39c5f0>
```

```
In [24]: print c.next()
```

```
3
```

```
In [25]: print c.next()
```

```
1
```

```
In [26]: c = (len(_) for _ in b)   # Use parentheses
         print max(c)
```

```
9
```

```
In [27]: print min(c)    # Cannot do; the max function already read the computed values
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-27-a62e960570c1> in <module>()
----> 1 print min(c)    # Cannot do; the max function already read the computed values


ValueError: min() arg is an empty sequence
```

Generators are great for one-time use:

```
In [28]: print max(len(_) for _ in b)   # Do this if we do not want to store the list

9
```

You can also create lists or generators using built-in functions:

```
In [29]: print range(3,9)    # a list
         print xrange(3,9)   # not a generator but behaves similar to one
         print list(xrange(3,9))    # convert to a list

[3, 4, 5, 6, 7, 8]
xrange(3, 9)
[3, 4, 5, 6, 7, 8]

In [30]: print enumerate(['a', 'b', 'c'])    # a generator
         print list(enumerate(['a', 'b', 'c']))

<enumerate object at 0x7fc56c141f50>
[(0, 'a'), (1, 'b'), (2, 'c')]

In [31]: print zip([1, 2, 3], [4, 5, 6])     # a list
         from itertools import izip
         print izip([1, 2, 3], [4, 5, 6])    # a generator
         print list(izip([1, 2, 3], [4, 5, 6]))

[(1, 4), (2, 5), (3, 6)]
<itertools.izip object at 0x7fc56c110050>
[(1, 4), (2, 5), (3, 6)]
```

2.2.4 Passing functions around

There are 2 ways to define functions. One is `def`:

```
In [32]: def foo(x):
             def subtractor(u):
                 return u - x
             return subtractor

In [33]: bar = foo(2)
         print bar
         print bar(7)

<function subtractor at 0x7fc56c107aa0>
5
```

Another way is to use `lambda` (anonymous functions):

```
In [34]: def foo(x):
             return lambda u: u - x

In [35]: bar = foo(2)
         print bar
         print bar(7)
```

```
<function <lambda> at 0x7fc56c1078c0>
5
```

Functions can also be used as arguments of other functions:

```
In [36]: def baz(fn):
             return fn(10) ** 3     # x ** y is x to the y-th power. Work with fractional y too!

In [37]: print baz(bar)       # -> bar(10) ** 3 --> (10 - 2) ** 3
```

```
512
```

2.2.5 Reading / Writing files

I like to use the `with` statement which closes the file after everything is done.

```
In [38]: s = []
         with open('numbers.txt') as fin:
             for line in fin:
                 s.append(int(line))
         print s
```

```
[2, 5, 3, 8, 6, 1, 7]
```

```
In [39]: with open('sum.txt', 'w') as fout:
             # There are many ways to write the sum to the file
             # Consult the documentation for these:
             fout.write('{0}\n'.format(sum(s)))
             fout.write('%d\n' % sum(s))
             print >> fout, sum(s)
```

The file now contains 3 lines of the number 32.

2.3 Gotchas

2.3.1 Division

In Python 2.7, the `/` sign rounds down integer divisions.

```
In [40]: print 5/2
         print 5./2         # Use a decimal point
         a = 5
         print a * 1./2
```

```
2
2.5
2.5
```

2.3.2 Tied objects

We can create a list of repeated objects like this:

```
In [41]: print [3] * 4
```

```
[3, 3, 3, 3]
```

However, don't do this:

```
In [42]: a = [[]] * 4
         print a
```

```
[[], [], [], []]
```

The problem is that all 4 empty lists are the **same** list:

```
In [43]: print id(a[0])
         print id(a[1])
         a[0].append('hello')
         print a
```

```
140485900902480
140485900902480
[['hello'], ['hello'], ['hello'], ['hello']]
```

One solution is to use list comprehension:

```
In [44]: a = [[] for _ in xrange(4)]
         print id(a[0])
         print id(a[1])
         a[0].append('hello')
         print a
```

```
140485935320888
140485935383184
[['hello'], [], [], []]
```

2.3.3 Global variable

```
In [45]: a = 4
         def foo():
             a = 5
             print a
         foo()
         print a
```

```
5
4
```

When you call `foo` and try to assign 5 to `a`, `foo` will create a **local** variable named `a`, which is different from the global `a` outside. To change the outside `a`, you need to put in the keyword `global`:

```
In [46]: a = 4
         def foo():
             global a
             a = 5
             print a
         foo()
         print a
```

```
5
5
```