KIT103/KMA155 Practical 08: Euclidean Algorithm and applications

This week you'll implement the Euclidean algorithm, perhaps try a little encryption, and explore how the size of a hash table (and the hash function used) can impact on how evenly values are distributed.

Preparation

In the Labs section of MyLO download the files **mult_encrypt.py**, **hash.py** and the collection of three-letter words **tlw.txt**.

1 Euclidean Algorithm

Task: Implement the Euclidean Algorithm from the Apps 6 lecture, then try it out on some values to make sure it works.

Now let's use it to answer a question. James (not his real name) is coordinating a unit that has students in Hobart and Launceston. He wants to divide his class into groups for an assignment, but how big can the groups be? The groups cannot span the two locations, and there are 114 students in Hobart and 36 in Launceston.

Task: Use your new gcd() function to answer the question above.

Task: (optional) James now decides that the groups have to be within tutorial groups. There are seven groups in Hobart (sizes in a list are `[18, 20, 18, 16, 14, 12, 14]`) and two in Launceston (sizes `[20, 16]`). What's the maximum group size now? (Hint: the two campus locations are now irrelevant, only the sizes matter.)

2 (optional) A little encryption

Load the script mult_encrypt.py in Spyder. It contains an encryption function `encrypt(message, key, symbols)` that applies a multiplicative cipher with key=`key` to the `message`, given a set of allowed `symbols` (and a decryption function, too, but don't rush to use that just yet). Also defined in the script are `alphabet`, which is the string `'abcd...xyz'`, and `ALPHABET` which consists of the equivalent upper case characters.

Although you *could* use this now to encrypt (and decrypt) a message, try the following exercise instead:

Task: Generate a string representing the mapping from each alphabet symbol to its encrypted counterpart by encrypting one of the two strings of symbols and a key of your choice, as in `encrypt(alphabet, 7, alphabet)`. You can do this for various key values to see what the mapping would look like.

Task: Although *you* know you can use *gcd*(*key*, |*symbols*|) to determine if a given key will work with a particular set of symbols, how might you check that the key you've chosen isn't mapping different input symbols to the same output? (Hint: your solution can be short and uses ideas from the start of semester.)

3 A little computational science: hash functions

A hash function h is a function that takes an object (such as a string) and converts it to an integer. An important use of hash functions is in indexed data structures like hash tables. A hash table has a number of 'buckets' into which elements are placed. The result of the hash function h is converted to a position using h (mod *table size*). Ideally the hash function (in combination with the table size) should produce a fairly even spread, even when the data to be stored have similarities. **In this task you will plot this distribution for different functions and table sizes.**

The script hash.py contains two different hash functions, with a couple of variations:

- an implementation of Java's string hash code function that returns an integer: `java_hash_nowrap(word)` returns the hash of `word`
- the same function but with the result wrapped to a certain number of buckets: `java_hash(word, buckets)` returns `java_hash_nowrap(word) % buckets`
- a wrapper for the Python `hash()` function that wraps its result: `py_hash(word, buckets)` returns `hash(word) % buckets`

It also includes a function called `distribution` that accepts a collection of `words`, the `size` of an imaginary hash table, and the name of a `hash` function to use to map words to positions. It returns a list of counts for how many words are mapped to each position.

Task: Make sure you have downloaded the file of three-letter words and then try the following:

```
words = [ l.rstrip() for l in open('tlw.txt') ]
from matplotlib import pyplot as plt
size = 10
dist = distribution(words, size, java_hash)
plt.bar(range(0,size), dist)
plt.show()
```

Task: Repeating the last four lines in the code above (or defining a function to make the task easier) vary the following, in this order:

- change `java_hash` to `py_hash`. **What's the outcome?**
- change `size` to 31. **What's the outcome?**
- change `py_hash` back to `java_hash`. **What's the outcome? Why?**

Note: The x-axis may extend further to the right than the last bucket position *size* – 1.

4 Bonus After Class Activity: Encrypt a phone number with RSA

This task is **completely optional** and included for those who'd like to see the steps of one of the leading encryption algorithms in action. Ideally it should be done in pairs (so you can encrypt a 'message' for the other person).

Click to reveal the steps