Project 1 Report: Cellphone Passcodes

Zhang Naifu
znf18@mails.tsinghua.edu.cn

7 October 2018

1 Results Summary

We assume iPhone unlock consists of 4 keys with repetition from {0,...,9}, and that Android unlock screen is a 3x3 grid.

There are trivially 10^4 = **10000** permutations for the iPhone unlock sequence. The python script outputs **389112** permutations for the Android unlock sequence. Assuming that the iPhone and Android users use all possible permutations of the unlock sequence with uniform probability, **Android is more secure**.

2 Implementation

The python script is made up of 3 functions, *isValid()*, *bruteForce()* and *semiBruteForce()*. *isValid()* is used as a helper function in *bruteForce()* and *semiBruteForce()* to check for validity. *semiBruteForce()* is a faster and more efficient version of *bruteForce()* to enumerate permutations.

2.1 Checking Validity

Given a sequence, *isValid()* returns *True* if sequence is valid. Such a sequence must necessarily contain:

1. No repetition of any digits and
2. No jumps (e.g. from 1 to 3) or
3. If there's any jump, the jumped number in the middle must have appeared in the sequence before the jump

2.2 Brute Force Enumeration

bruteForce() generates all possible permutations of length n, including the invalid ones, before passing each through the *isValid()* function. It gives a final count of the valid sequences.

To tally the total number of unlock sequences, *bruteForce()* has to be called 6 times for n = {4,5,6,7,8,9}, and takes ∼9 seconds. This was stretching my patience.

2.3 Exploiting Symmetries

semiBruteForce() implements some shortcuts.

1. The number of valid sequences starting from 1 is the same as that starting from any corner {1,3,7,9}; instead of generating all permutations of length n, *semiBruteForce()* only considers those starting with 1, and count every such valid unlock sequence 4 times. By the same token, this is true of the set {2,4,6,8} as well
2. *semiBruteForce(8) = semiBruteForce(9)* for obvious reason

3 Complexity

The **time complexity** of *isValid()* depends on *str.count()* and *str.find()*. *str.count()* implements fastsearch and is therefore *O(n)*. *str.find()* implements a mix between Boyer-More and Horspool, which is *O(n)* on average.

bruteForce() calls on *itertools.permutations()* with *O(n!)* once, and *isValid()* *n!* times. Abusing the notation a little, the total running time from length 4 to 9 is

$$\sum_{n=4}^{9} O(n!) + n!O(n)$$

semiBruteForce() calls on *itertools.permutations()* once and only calls on *isValid()* for a subset of the permutations. The total running time is

$$\sum_{n=4}^{8} O(n!) + 3(n-1)!O(n)$$

The last term in each equation asymptotically dominates. Although both have the same asymptotic efficiency, *semiBruteForce()* is faster by a factor of $\sim$3 in practice, as it only considers permutations starting with {1,2,5} and we omit *semiBruteForce(9)*. Refer to sample output below.

For the same reason, *semiBruteForce()* is better in terms of **space complexity** by a factor of $\sim$3.

Sample Output

```
bruteForce:
389112 different unlock sequences of length 4 to 9
9.07 seconds taken
semiBruteForce:
389112 different unlock sequences of length 4 to 9
2.57 seconds taken
```

3.1 Efficiency Improvements

Further improvements could perhaps be made by considering only valid sequences of length *n* when generating sequences of length *n+1*, since no invalid sequence of length n would give valid sequence of length *n+1*. **Dynamic programming** could also be used. But I'm happy with the 2.5 seconds *semiBruteForce()* implementation so we stop here.

References

python string documentation *https://github.com/python/cpython/tree/v3.6.5/Objects/stringlib*
python itertools documentation *https://docs.python.org/3.5/library/itertools.html#itertools.permutations*