Catherine Turner
ITP 125

Final Project: Option 1

Additional Information

Script Instructions

The only required library is the Python standard library.

List of Hashed Passwords

The hashed passwords are contained in the hashes.txt file. Passwords and their corresponding MD5 hashes are also listed, for clarity, below:

Password	MD5 Hash
Z	21c2e59531c8710156d34a3c30ac81d5
AD	e182ebbc166d73366e7986813a7fc5f1
God	aeb9573c09919d210512b643907e56b8
1234	81dc9bdb52d04dc20036dbd8313ed055
AbCdE	37e464916dcb6dfc3994ca4549e97272
Trojan	a5312fd5717d3e2fd419132e3c3ac51b
F1ghtOn	57b46b0a1ac529ef1a3d6fa016b6995e
P@ssword	382e0360e4eb7b70034fbaa69bec5786

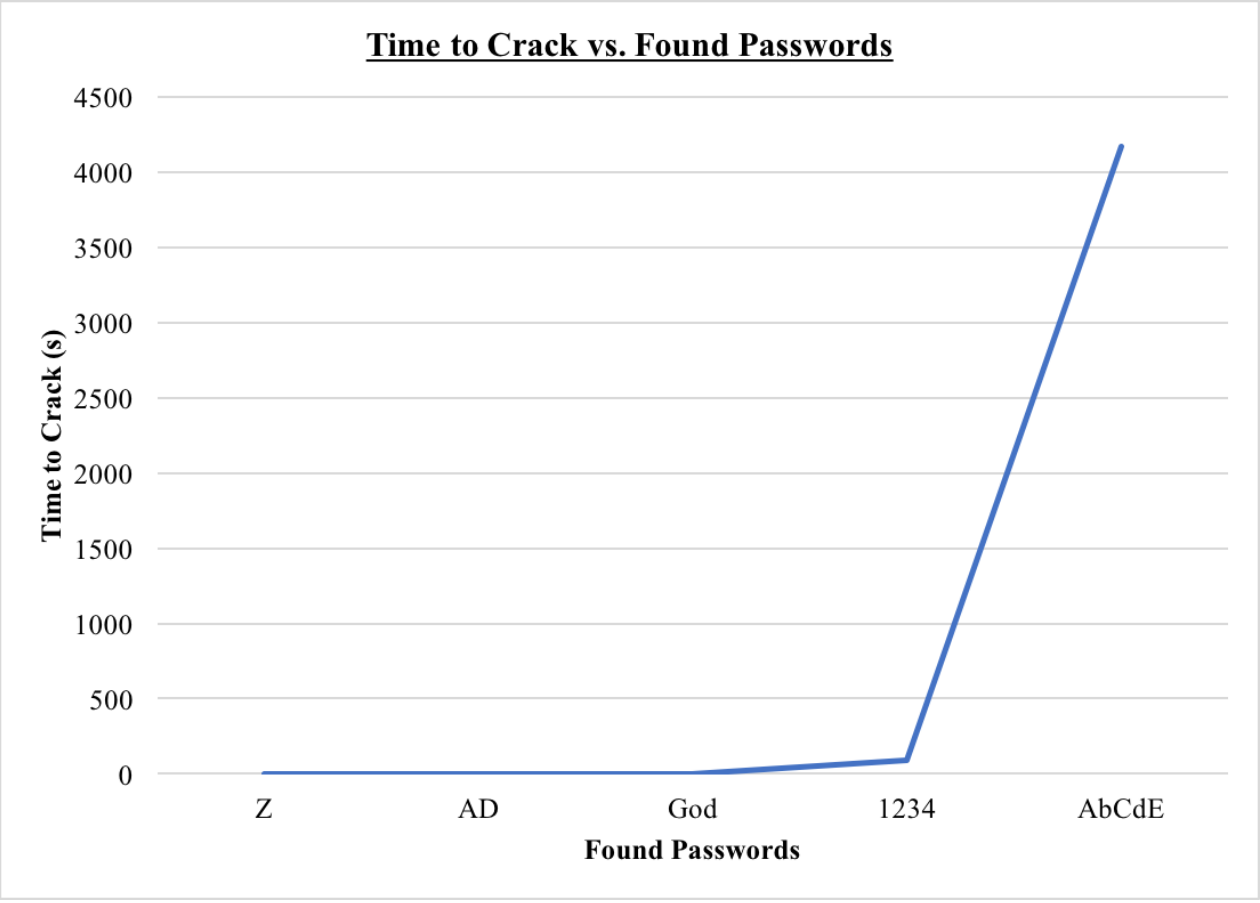
Results

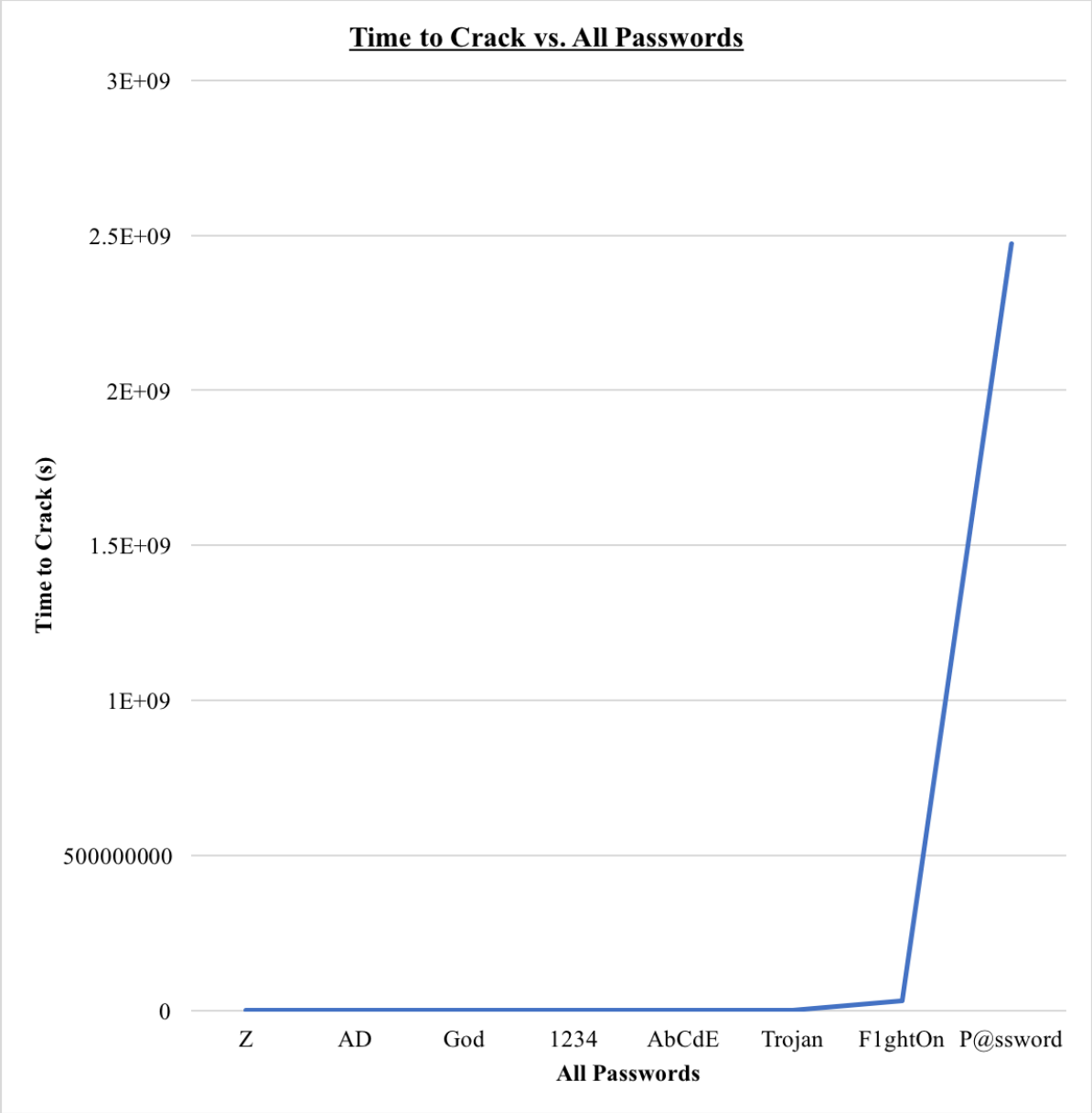
Password	Was it cracked?	Time to Crack (seconds)
Z	Yes	0.00043892860412597656
AD	Yes	0.008678913116455078
God	Yes	0.5748090744018555
1234	Yes	90.87185525894165
AbCdE	Yes	4,173.749496936798

Trojan	No	350,594.957743 (predicted)
F1ghtOn	No	29,449,976.4504 (predicted)
P@ssword	No	2,473,798,021.83 (predicted)

Analysis

As the length of the password increases, the time to crack the password obviously increases along with it. The first chart below displays all cracked passwords and the time it took to crack them. The following chart displays all passwords including actual crack times and predicted crack times for those that could not be resolved.





In order to predict how long it would take to crack the passwords I wasn't able to crack, I first determined the amount of possibilities that exist for every password length. Every character could be a lowercase character (26 options), an uppercase character (26 options), or a punctuation mark (32 options included in my code). By using the formula for permutations with replacement allowed (as in, characters can be repeated), I determined the expected amounts of combinations below.

For reference, the formula for permutations with replacement allowed is n^r, with n being the number of potential characters and r being the length of the password.

Number of characters in password	Number of potential character combinations
1	84
2	7046
3	592704
4	49787136
5	4182119424
6	351298031616
7	29509034655744
8	2478758911082496

Next, I divided the amount of time it took to solve the 5-character password by the number of potential combinations (4173.749496936798 / 4182119424 = 9.9799864e-7), thus assuming that it took 9.9799864e-7 seconds to find each combination, determine its hash, and compare that hash to the desired hash from the file. I extrapolated this idea by multiplying 9.9799864e-7 seconds by the number of combinations for the 6-, 7-, and 8-character passwords.

In the future, the password cracking process could be made faster in a few ways. First of all, I could create a list within my program of all possible character combinations for passwords of 1-8 characters before checking the hashes, so that they wouldn't have to be regenerated every time. Next, programs in Python don't automatically take advantage of multiple CPUs. Thus, I could have applied multi-core programming or even cloud computing to distribute my work across multiple networked nodes. I could've even sped up that process by incorporating GPU computing, using the GPU as a co-processor to accelerate CPUs. However, I think that this

project provided me with newfound knowledge about how simple brute force password cracking can be, yet simultaneously how time-consuming.

Sources:

https://p16.praetorian.com/blog/multi-core-and-distributed-programming-in-python

https://www.boston.co.uk/info/nvidia-kepler/what-is-gpu-computing.aspx