REPORT ON BRUTE FORCE CRACKER

Background:

The first task was to develop a brute force cracker to crack a MD5 hashed (unsalted) password database. It targets passwords that are no longer than four characters. The program reads hashed passwords from an input password database CSV file **(which must be in the same folder as the code)**, attempts to crack them, and writes the results to a new CSV file titled **"task1.csv". The input password database should be in the same folder as the codes.**

Tools and Libraries Used:

1). **hashlib:** Used for hashing passwords using the MD5 algorithm.

2). **itertools:** Used for generating combinations of characters for brute-force attempts.

3). **time:** Used for measuring the time taken to crack the passwords.

4). **csv:** Used for reading and writing CSV files.

5). **sys:** Used for handling command-line arguments.

6). **collections.defaultdict:** Used for storing cracked passwords in a dictionary with sets as default values.

Functions and Their Roles:

1) **md5_hash(password):** This function takes a password as input, encodes it, and returns its MD5 hash.

2) **brute_force_crack(hash_list):** This function performs the brute-force cracking of passwords. It generates all possible combinations of characters up to a length of four (using itertools) and compares their hashes with the given list of hashed passwords.

Variables:

-**chars:** A string containing all possible characters for the passwords.

-**max_length:** The maximum length of the passwords to be cracked.

-**cracked_passwords:** A dictionary to store cracked passwords.

-**successful_usernames:** A set to store usernames for which passwords were successfully cracked.

-**itertools.product:** Generates all possible combinations of characters up to the specified length.

-**md5_hash:** Hashes the generated password which is then used to compare with the given hashes

3). **read_csv(file_path):** This function reads the hashed passwords from a CSV file and returns a list of tuples containing usernames and their corresponding hashed passwords.

4). **write_csv(file_path, cracked_passwords, total_time, success_rate, original_usernames):** This function writes the cracked passwords, total time taken, and success rate to a CSV file. I also send in an extra parameter called original_usernames to make sure that the output is in the same order as the input file.

5). **main():** This is the main function that commands the entire process. It reads the input file, calls the brute_force_crack function, and writes the results to the output file.

Variables:

-**start_time:** Records the start time of the process.

-**input_file:** The input CSV file containing hashed passwords.

-**output_file:** The output CSV file to store the results.

-**hash_list:** The list of hashed passwords read from the input file.

-**original_usernames:** A list of usernames extracted from the hash list. This is what is passed to the write function to make sure the cracked passwords ordering is accurate.

-**cracked_passwords:** The dictionary of cracked passwords.

-**success_rate:** The success rate of the cracking process.

-**end_time:** Records the end time of the process.

-**total_time:** The total time taken for the process.

Performance:

The performance I got for this task was around 13 seconds for one password. If there are more than one password and they are less plain in nature, e.g. pAs1 instead of pass, it does take a proportionately longer amount of time. The success rate does depend on if the input hashed passwords were originally less than or equal to 4 characters in length. Failed passwords don't seem to significantly increase the processing time.

REPORT ON DICTIONARY ATTACK CRACKER

Background:

The second task was to develop a dictionary attack cracker to crack a MD5 hashed (unsalted) password database. It reads the hashed passwords and a list of common passwords from CSV files, attempts to crack the passwords by comparing their hashes, and writes the results to CSV file titled “task2.csv”. The tarball provided will have the **“common_passwords.csv”.** Both the common password file and the input password database should be placed in the **same folder** as the codes when **running** it. **Both files must be in the same folder as the codes.**

Tools and Libraries Used:

1). **hashlib:** Used for hashing passwords using the MD5 algorithm.

2). **time:** Used for measuring the time taken to crack the passwords.

3). **csv:** Used for reading and writing CSV files.

4). **sys:** Used for handling command-line arguments.

5) **collections.defaultdict:** Used for storing cracked passwords in a dictionary with sets as default values.

Functions and Their Roles:

1). **md5_hash(password):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

2). **read_input_csv(file_path):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

3). **read_common_csv(file_path):** This function reads a list of common passwords from a CSV file and returns a list of these passwords.

4). **dictionary_attack(hash_list, common_pass_list):** This function performs the dictionary attack by comparing the hashes of common passwords with the given list of hashed passwords.

Variables:

-**cracked_passwords:** A dictionary to store cracked passwords.

-**successful_usernames:** A set to store usernames for which passwords were successfully cracked.

-**md5_hash:** Hashes the common passwords which is used to compare with the given hashes in the password database.

5). **write_csv(file_path, cracked_passwords, total_time, success_rate, original_usernames):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

6). main(): This is the main function that commands the entire process. It reads the input file, calls the dictionary attack function, and writes the results to the output file titled **"task2.csv".**

Variables:

-**start_time:** Records the start time of the process.

-**input_file:** The input CSV file containing hashed passwords.

-**output_file:** The output CSV file to store the results.

-**hash_list:** The list of hashed passwords read from the input file.

-**common_pass_list:** The list of commons passwords read from the common passwords file.

-**cracked_passwords:** The dictionary attack of cracked passwords.

-**success_rate:** The success rate of the cracking process.

-**end_time:** Records the end time of the process.

-**total_time:** The total time taken for the process.

Performance:

The performance I got for this task was miliseconds for one password. If there are around 6 passwords the processing time is still in the miliseconds. The success rate does depend on if the input password was found in the list of common passwords. Failed passwords don't seem to significantly increase the processing time.

REPORT ON RAINBOW TABLE CRACKER (UNSALTED)

Background:

The third task was to develop a rainbow table cracker to crack a MD5 hashed (unsalted) password database. A pre-computed rainbow table of hash values for common passwords is used to crack the list of MD5 hashed (unsalted) passwords. It reads the hashed passwords, stored in **the input password database** and a list of common passwords, stored in **"common_passwords.csv"**, and generates a rainbow table for the common passwords and their hashes. It uses this generated table to crack the passwords by comparing hashes. **Both the input password database and common passwords files must be in the same folder as the codes.**

Tools and Libraries Used:

1). **hashlib:** Used for hashing passwords using the MD5 algorithm.

2). **time:** Used for measuring the time taken to crack the passwords.

3). **csv:** Used for reading and writing CSV files.

4). **sys:** Used for handling command-line arguments.

5). **collections.defaultdict:** Used for storing cracked passwords in a dictionary with sets as default values.

Functions and Their Roles:

1). **md5_hash(password):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

2). **read_input_csv(file_path):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

3). **read_common_csv(file_path):** Serves the same role as explained in **REPORT ON DICTIONARY ATTACK CRACKER.**

4). **get_rainbowtable(list, rainbowtable_csv):** This function generates a rainbow table by hashing each common password and writes the rainbow table to a CSV file titled **"rainbowtable.csv"**.

5). **rainbowtable_attack(hash_list, rainbow_table):** This function performs the rainbow table attack by comparing the hashes of common passwords from the rainbow table with the given list of hashed passwords. It opens the file "rainbowtable.csv" for reading so I can access the hash values from the table.

Variables:

-**cracked_passwords:** A dictionary to store cracked passwords.

-**successful_usernames:** A set to store usernames for which passwords were successfully cracked.

6). **write_csv(file_path, cracked_passwords, total_time, success_rate, original_usernames):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

7). **main():** This is the main function that commands the entire process. It reads the input passwords database and the common passwords list. Generates a rainbow table based on the common passwords list and calls the rainbow table attack function, and then writes the results to the output file titled **"task3.csv".**

Variables:

-**start_time:** Records the start time of the process.

-**input_file:** The input CSV file containing hashed passwords.

-**output_file:** The output CSV file to store the results.

-**hash_list:** The list of hashed passwords read from the input file.

-**common_pass_list:** The list of commons passwords read from the common passwords file.

-**rainbow_table_csv:** The CSV file to store the rainbow table.

-**cracked_passwords:** The dictionary attack of cracked passwords.

-**success_rate:** The success rate of the cracking process.

-**end_time:** Records the end time of the process.

-**total_time:** The total time taken for the process.

Performance:

The performance I got for this task was miliseconds for one password. If there are around 6 passwords the processing time is still in the miliseconds. The success rate does depend on if the input password was found in the list of common passwords. Failed passwords don't seem to significantly increase the processing time.

REPORT ON RAINBOW TABLE CRACKER (SALTED)

Background:

The fourth task was to develop a rainbow table cracker to crack a MD5 hashed (salted) password database. A pre-computed rainbow table of hash values for common passwords is used to crack the list of MD5 hashed (salted) passwords. It reads the hashed passwords, stored in **input password database** and a list of common passwords, stored in **"common_passwords.csv"**, and generates a rainbow table for the common passwords and their hashes generated from a **md5_hash(common password + salt of a specific salt for a password in the password database)**. It uses this generated table to crack the passwords by comparing hashes. **Both the input password database and common passwords files must be in the same folder as the codes.**

Tools and Libraries Used:

1). **hashlib:** Used for hashing passwords using the MD5 algorithm.

2). **time:** Used for measuring the time taken to crack the passwords.

3). **csv:** Used for reading and writing CSV files.

4). **sys:** Used for handling command-line arguments.

5). **collections.defaultdict:** Used for storing cracked passwords in a dictionary with sets as default values.

Functions and Their Roles:

1). **md5_hash(password):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

2). **read_input_csv(file_path):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

3). **read_common_csv(file_path):** Serves the same role as explained in **REPORT ON DICTIONARY ATTACK CRACKER.**

4). **get_rainbowtable(list, rainbowtable_csv):** This function generates a rainbow table by hashing each common password + the salt of a specific password from the database and writes the rainbow table to a CSV file titled **"rainbowtable.csv"**.

5). **rainbowtable_attack(hash_list, rainbow_table):** This function performs the rainbow table attack by comparing the hashes of common passwords from the rainbow table with the given list of hashed passwords. However, since generating a rainbow table for every salt in the password database and combining it into one extremely large file would be highly inefficient, I only create a rainbow table for the specific salt I need at each time. It opens the file “rainbowtable.csv” for reading so I can access the hash values from the table.

Variables:

-**cracked_passwords:** A dictionary to store cracked passwords.

-**successful_usernames:** A set to store usernames for which passwords were successfully cracked.

6). **write_csv(file_path, cracked_passwords, total_time, success_rate, original_usernames):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

7). **main():** This is the main function that commands the entire process. It reads the input passwords database and the common passwords list. Results are written to the output file titled **“task4.csv”.**

Variables:

-**start_time:** Records the start time of the process.

-**input_file:** The input CSV file containing hashed passwords.

-**output_file:** The output CSV file to store the results.

-**hash_list:** The list of hashed passwords read from the input file.

-**common_pass_list:** The list of commons passwords read from the common passwords file.

-**rainbow_table_csv:** The CSV file to store the rainbow table.

-**cracked_passwords:** The dictionary attack of cracked passwords.

-**success_rate:** The success rate of the cracking process.

-**end_time:** Records the end time of the process.

-**total_time:** The total time taken for the process.

Performance:

The performance I got for this task was miliseconds for one password. If there are around 6 passwords the processing time is still in the miliseconds. The success rate does depend on if the input password was found in the list of common passwords. Failed passwords don't seem to significantly increase the processing time.

REPORT ON HYBRID PASSWORD CRACKER

Background:

The fifth task was to develop a password cracker to crack a MD5 hashed (salted) password database that are variations of the base passwords in the list of common passwords. For each row in the password database file, I generate the variations for each of the 10,000 common passwords, in **"common_passwords.csv".** For each list of variations I add the salt, hash, and compare each of the hashed variation passwords to the hashed password of the current row in the password database. In this code when it is running, in the terminal it will print the progress of the passwords from the input file as they are processed. **Both the input password database and common passwords files must be in the same folder as the codes.**

Tools and Libraries Used

1). **hashlib**: Used for hashing passwords using the MD5 algorithm.

2). **time**: Used for measuring the time taken to crack the passwords.

3). **csv**: Used for reading and writing CSV files.

4). **sys**: Used for handling command-line arguments.

5). **threading**: Used for multithreading to speed up the password cracking process.

6). **collections.defaultdict**: Used for storing cracked passwords in a dictionary with sets as default values.

Functions and Their Roles:

1). **md5_hash(password):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

2). **generate_case_combinations(word):** This function generates all possible case combinations of a given word.

3). **append_digits(combinations):** This function appends digits (up to four digits) to each combination of the word.

4). **replace_characters(combinations):** This function replaces certain characters in each combination with numbers or symbols.

5). **process_single_password(dictionary, lock, results, thread_id, task_queue):** This multithreading function processes the shared dictionary list and attempts to crack the passwords.

How passwords are processed in each thread:

First, I get the original password from the common password list, and I create all the case combinations and get a list with all the case combinations:

case_combos(original_password) = CASE_LIST

Next, I take the case list and put it through the digit appender.

append_digits(CASE_LIST) = DIGIT_APPEND_LIST

Before calling the next function, I append this new list to the previous CASE_LIST:

CASE_LIST.extend(DIGIT_APPEND_LIST) = CASE+DIGIT_LIST

Now, I take this case + digit list and put it through the replace characters function:

Replace_characters(CASE+DIGIT_LIST) = REPLACE_LIST

I now append this REPLACE_LIST to the previous CASE_LIST:

CASE_LIST.extend(REPLACE_LIST)

Now I have my final list for all the variations of the given password. That I can compare and check. There are 2 threads doing this, working on 2 different passwords at a time that come from the input password database.

6). **crack_passwords(password_db, dictionary):** This function commands the multithreading process to crack the passwords. I launch two threads to handle this. Each thread processes the tasks from the queue, trying to crack the passwords using the transformations. It tracks the number of successful cracks and returns the results with the success count and total count. This is where I check if there were any usernames not successfully cracked and assign them a “FAILED” string.

Variables:

-**task_queue:** This is a queue used to store the user's username, hashed password, and salt.

-**success_count:** This is the number of successful passwords cracked.

-**total_count:** This is just to return the length of the passwords database.

7). **read_input_csv(file_path):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

8). **read_common_csv(file_path):** This function reads a list of commons passwords from the **"common_passwords.csv"** file and returns a list of these passwords.

9). **write_csv(file_path, cracked_passwords, total_time, success_rate, original_usernames):** Serves the same role as explained in **REPORT ON BRUTE FORCE CRACKER.**

10). **main():** This function commands the entire process. It reads the password database file and common passwords file, calls the hybrid password cracking function, and writes the results to the output file titled **"task5.csv".**

Variables:

-**start_time:** Records the start time of the process.

-**output_file:** The output CSV file to store the results.

-**hash_list:** The list of hashed passwords read from the input file.

-**common_pass_list:** The list of commons passwords read from the common passwords file.

-**rainbow_table_csv:** The CSV file to store the rainbow table.

-**cracked_passwords:** The dictionary attack of cracked passwords.

-**success_rate:** The success rate of the cracking process.

-**end_time:** Records the end time of the process.

-**total_time:** The total time taken for the process.

Performance:

The performance I got for this task depended on where in the common passwords list the input hashed password is. If the input hashed password was the word “password” it takes 30 seconds. If the input hashed password is some variation of the word “password” it depends on which variation it is. If it is one of the last variations in the list that stores all the variations of “password” then it can take closer to 57 seconds to process. Now if I do something like “smokey”, which is the 150th password in the common passwords list, then it will take closer to 8-9 minutes to process. For the hybrid password cracker, a failed password seems to have a significant impact on the processing speed. If there is a failed password the program will have to go through every password in the list of 10,000 passwords resulting in a large computation time which will be in the hours.