13/5/2025

Report on Milestone 2 solution

Team IDs: Hadi Shaheen **315490193**, Hen Frydman **208009845**

Difficulty level: we steadily got the correct password in under 10 minutes for difficulty = **8**

1. Sample Execution

Below are sample execution examples of two console sessions for each team member (hadi `315490193` and hen `208009845`, difficulty `8`).

```python
if __name__ == "__main__":
    # The username to be used for the attack (the needed ID)
    USERNAME = input("Enter username: ")
    # the needed difficulty
    DIFFICULTY = float(input("Enter difficulty: "))
    DIFFICULTY_FIX = DIFFICULTY / 1.5
    DIFFICULTY_multiplier = math.ceil((DIFFICULTY)  / 3)
    DIFFICULTY_half = math.floor(DIFFICULTY+1  / 2)
    REPEATS = DIFFICULTY_multiplier * DIFFICULTY_half
    INITIAL_REPEATS = (DIFFICULTY_multiplier + 1)  * DIFFICULTY_half
    password, running_time = guess_password()
    print("password: ", password)
    mins = math.floor(running_time/60)
    secs = math.floor(running_time % 60)
    print("running time: ", mins, " minutes, ", secs, " seconds"  )
```

```
Enter username: 315490193
Enter difficulty: 8
password:  xcolefwvfdtlkrpl
running time:  9  minutes,  41  seconds
```

```python
if __name__ == "__main__":
    # The username to be used for the attack (the needed ID)
    USERNAME = input("Enter username: ")
    # the needed difficulty
    DIFFICULTY = float(input("Enter difficulty: "))
    DIFFICULTY_FIX = DIFFICULTY / 1.5
    DIFFICULTY_multiplier = math.ceil((DIFFICULTY)  / 3)
    DIFFICULTY_half = math.floor(DIFFICULTY+1  / 2.5)
    REPEATS = DIFFICULTY_multiplier * DIFFICULTY_half
    INITIAL_REPEATS = (DIFFICULTY_multiplier + 1)  * DIFFICULTY_half
    password, running_time = guess_password()
    print("password: ", password)
    mins = math.floor(running_time/60)
    secs = math.floor(running_time % 60)
    print("running time: ", mins, " minutes, ", secs, " seconds"  )
```

```
Enter username: 208009845
Enter difficulty: 8
password:  lhpnnzmulqnhwpef
running time:  8  minutes,  22  seconds
```

- A table of all the passwords found in under 10 minutes:

Difficulty	Hadi - 315490193	Hen - 208009845
1	Tflqcluenhffcczt	qizyofhsqjmaucne
2	Qunhtnqzmpqumoce	pzjshkritliodxpn
3	Xfrihtzfephdhvku	zbtcxvnmntrdmxtp
4	Zpchnebqlbkninow	jqotziwsacbqznta
5	Bllesdirxgfnifpp	eaxgemrwgorytudt
6	Okrhgiryqwyarhay	bljkadqwvjefbpsf
7	Ezccsmkbvkglurcf	gztmoehuyqxeptis
8	xcolefwvfdtlkrpl	lhpnnzmulqnhwpef
9	bcwuabswymyjuxfs	

2. Analysis

1. theoretically: using the Brute-force search space approach.

- Password length L = 16, alphabet size |Σ| = 26.

- Total possible combinations = 26^16 ≈ 4.3×10^22.

2. Our program's actual number of attempts.

- For the first position we don't apply the early stopping mechanism in order to be accurate in the margin calculation, so we make 26*REPEATS = 26 * (Diificulty*1.5), let's say difficulty is 8 then we make 26*12 = 312 requests for the first position.

- for the rest of the position we wait until at least 5 threads had finished then we start applying the early stop mechanism. So let's say we find the correct letter half-way on average then we make 13 * 12 = 156 to find the right one. And for the 15 letters we make like 15 * 156 = 2340attempts to find the right one.

- if we found a wrong letter in the previous iteration then we backtrack. This would cost us let's say like to additional such iterations meaning 2*156 = 312 additional requests. And as we have 15 letters in the password (other than the first) and let's say we missed in 0.3 of them (like 5 letters). Then we make additional 5 * 312 = 1560 requests.

- overall, on average for difficulty = 8, we make about 312 + 2340 + 1560 = 4212. To be statically correct and as we are using threads, we can say that in the wors case we will do as double attempts of this number ≈ 8000.

3. Speed-up factor.

If let's say in the brute-force approach we make as half of the attempts space to find the right password. Then it holds that the speed-up = (2.1×10^22) / 8000.

3. Optimizations

- Parallel probing: up to 26 simultaneous requests of HTTP GETs using ThreadPoolExecutor, each thread is responsible for a char from the alphabet.
- To reduce the noises and be accurate on the timing's calculations, for each char in the alphabet its thread performs repeated trails (the amount of the repeats is increased as the difficulty is increased)
- Early stopping: dynamically computed threshold (margins) according to the timing's differences. And use it to stop early when at least k=5 trials have got results already.
-

- If the early stopping does not hold for any of the chars, we don't hurry to backtrack, instead we soften the margins values and try to again (up to 5 times no more) to find the correct letter for the current position, otherwise we back track
- Robust statistics & outlier filtering: discard times outside [0.6median,1.4median] and use median instead of average.
- Difficulty-based tuning: adjust REPEATS and thresholds per difficulty.
- Retry/back-off strategy: sleep 1–3s on network errors.
- Lightweight HTTP calls: minimal request overhead to isolate server stalls.
- Working at night times or look for times when it seems to be an empty traffic in the way to the server helped doing the homework easier to manage with.