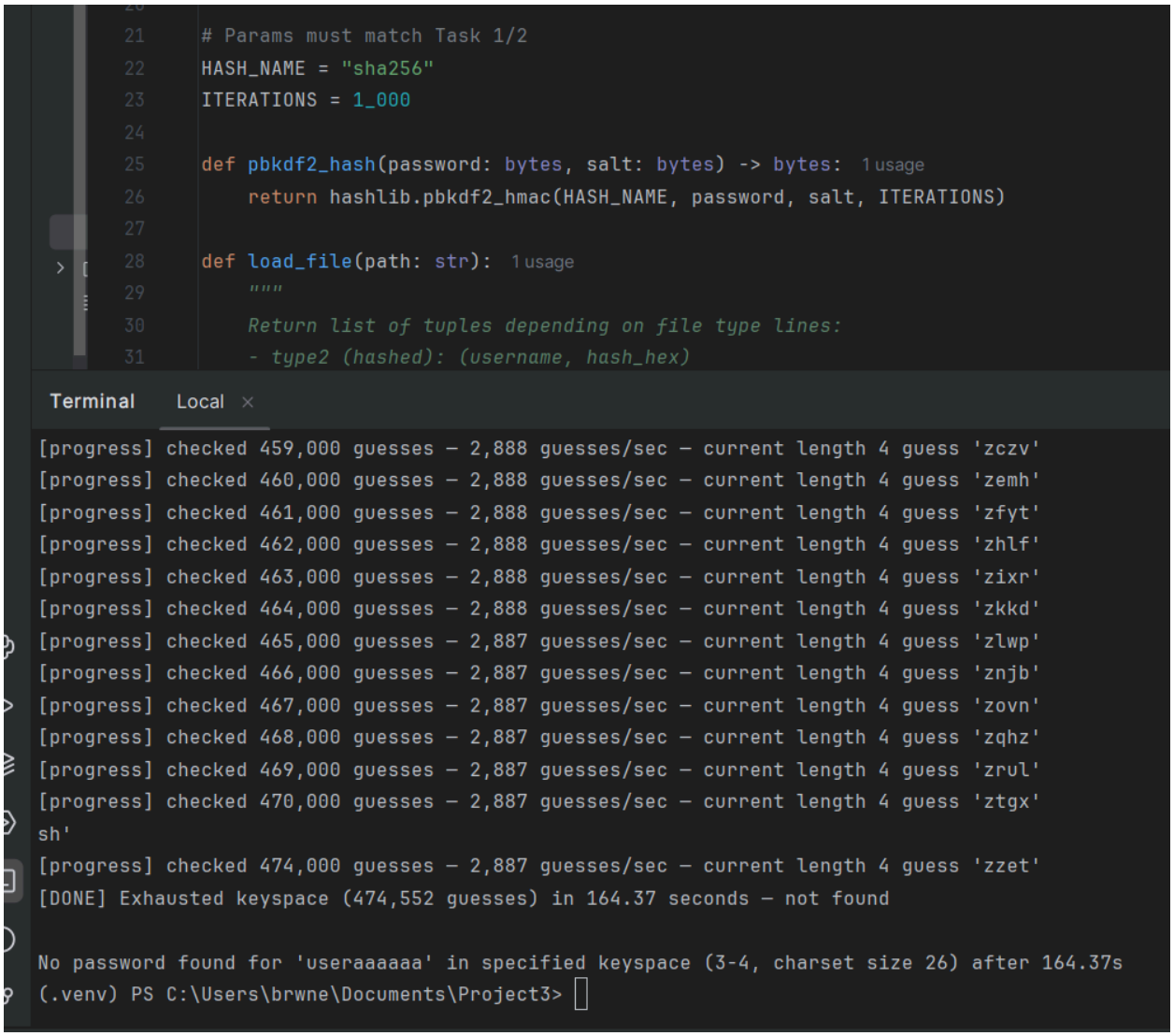
Project 3

Testing and Analysis

Task2_hashed.txt result



```
# Params must match Task 1/2
HASH_NAME = "sha256"
ITERATIONS = 1_000

def pbkdf2_hash(password: bytes, salt: bytes) -> bytes:  1 usage
    return hashlib.pbkdf2_hmac(HASH_NAME, password, salt, ITERATIONS)

def load_file(path: str):  1 usage
    """
    Return list of tuples depending on file type lines:
    - type2 (hashed): (username, hash_hex)
```

```
Terminal  Local ×
[progress] checked 459,000 guesses — 2,888 guesses/sec — current length 4 guess 'zczv'
[progress] checked 460,000 guesses — 2,888 guesses/sec — current length 4 guess 'zemh'
[progress] checked 461,000 guesses — 2,888 guesses/sec — current length 4 guess 'zfyt'
[progress] checked 462,000 guesses — 2,888 guesses/sec — current length 4 guess 'zhlf'
[progress] checked 463,000 guesses — 2,888 guesses/sec — current length 4 guess 'zixr'
[progress] checked 464,000 guesses — 2,888 guesses/sec — current length 4 guess 'zkkd'
[progress] checked 465,000 guesses — 2,887 guesses/sec — current length 4 guess 'zlwp'
[progress] checked 466,000 guesses — 2,887 guesses/sec — current length 4 guess 'znjb'
[progress] checked 467,000 guesses — 2,887 guesses/sec — current length 4 guess 'zovn'
[progress] checked 468,000 guesses — 2,887 guesses/sec — current length 4 guess 'zqhz'
[progress] checked 469,000 guesses — 2,887 guesses/sec — current length 4 guess 'zrul'
[progress] checked 470,000 guesses — 2,887 guesses/sec — current length 4 guess 'ztgx'
sh'
[progress] checked 474,000 guesses — 2,887 guesses/sec — current length 4 guess 'zzet'
[DONE] Exhausted keyspace (474,552 guesses) in 164.37 seconds — not found

No password found for 'useraaaaaa' in specified keyspace (3-4, charset size 26) after 164.37s
(.venv) PS C:\Users\brwne\Documents\Project3>
```

Task2_salted.txt result

```
[progress] checked 340,000 guesses — 2,895 guesses/sec — current length 4 guess 'siyx'
[progress] checked 350,000 guesses — 2,895 guesses/sec — current length 4 guess 'sxtn'
[progress] checked 360,000 guesses — 2,895 guesses/sec — current length 4 guess 'tmod'
[progress] checked 370,000 guesses — 2,894 guesses/sec — current length 4 guess 'ubit'
[progress] checked 380,000 guesses — 2,894 guesses/sec — current length 4 guess 'uqdj'
[progress] checked 390,000 guesses — 2,894 guesses/sec — current length 4 guess 'vexz'
[progress] checked 400,000 guesses — 2,894 guesses/sec — current length 4 guess 'vtsp'
[progress] checked 410,000 guesses — 2,894 guesses/sec — current length 4 guess 'winf'
[progress] checked 420,000 guesses — 2,893 guesses/sec — current length 4 guess 'wxhv'
[progress] checked 430,000 guesses — 2,893 guesses/sec — current length 4 guess 'xmcl'
[progress] checked 440,000 guesses — 2,893 guesses/sec — current length 4 guess 'yaxb'
[progress] checked 450,000 guesses — 2,893 guesses/sec — current length 4 guess 'yprr'
[progress] checked 460,000 guesses — 2,893 guesses/sec — current length 4 guess 'zemh'
[progress] checked 470,000 guesses — 2,893 guesses/sec — current length 4 guess 'ztgx'
[DONE] Exhausted keyspace (474,552 guesses) in 164.03 seconds — not found

No password found for 'useraaaaaa' in specified keyspace (3-4, charset size 26) after 164.03s
(.venv) PS C:\Users\brwne\Documents\Project3>
```

Environment & test setup

I ran all experiments on my Windows laptop using Python 3.14 inside a virtual environment (PyCharm .venv). I used the provided task2_generate.py to create the datasets and the task3_cracker.py brute-force script (modified only to use a temporary low iteration count when needed for quick tests). The relevant hashing parameters used by my code are PBKDF2-HMAC-SHA256 with a per-test iteration count of ITER_TEST = 1,000 (I used this reduced iteration count for quick measurements - see note below about scaling to the assignment's ITER_REAL = 100,000). All tests used the lowercase charset (a–z) in the cracker (charset option 1). I generated the default dataset of 100 accounts with password lengths between 3 and 8 using the Task 2 generator; for timing I ran focused small keyspace tests of lengths 3–4 because longer ranges take exponentially longer to brute force.

What I ran (actual measured tests)

1. task2_hashed.txt (type-2, hashed). I selected the first entry (line 1: useraaaaaa) and asked the cracker to exhaust the lowercase keyspace for passwords length 3–4 ($26^3 + 26^4 = 474{,}552$ guesses). Using ITER_TEST = 1,000, the cracker reported:

[DONE] Exhausted keyspace (474,552 guesses) in 164.37 seconds — not found

2. task2_salted.txt (type-3, salted). I repeated the same test (same keyspace 3–4) on the salted file (line 1). The cracker reported:

[DONE] Exhausted keyspace (474,552 guesses) in 164.03 seconds — not found

Scaling method (how I predict longer lengths / assignment iteration count)

PBKDF2 runtime is linear with the iteration count. I used ITER_TEST = 1,000 to measure quickly and then scaled results to ITER_REAL = 100,000 by multiplying time by (ITER_REAL / ITER_TEST) = 100. The measured guesses/second (at ITER_TEST) and the scaled guesses/second (at ITER_REAL) are computed as:

Type-2 (hashed) measured:

Guesses tried: G = 474,552

Time measured: T_test = 164.37 s

g_test = G / T_test ≈ 474,552 / 164.37 ≈ 2,887.1 guesses/sec

Scaled (real) speed: g_real ≈ g_test × (ITER_TEST / ITER_REAL) ≈ 2,887.1 × (1,000 / 100,000) ≈ 28.87 guesses/sec

Scaled time for the same 3–4 keyspace: T_real = T_test × 100 ≈ 16,437 s ≈ 4.57 hours

Type-3 (salted) measured:

Guesses tried: G = 474,552

Time measured: T_test = 164.03 s

g_test ≈ 474,552 / 164.03 ≈ 2,893.6 guesses/sec

g_real ≈ 28.94 guesses/sec

T_real ≈ 16,403 s ≈ 4.56 hours

Predicted times for longer password lengths (lowercase only)

I did not run the cracker for lengths 5–8 because full searches grow exponentially and would take impractically long to complete. Instead I predicted times using the measured g_real (≈ 28.9

guesses/sec). Below I show the keyspace for each fixed length and the estimated time to exhaust that keyspace:

length 5: 26^5 = 11,881,376 guesses

Time ≈ 11,881,376 / 28.9 ≈ 411,000 s ≈ 114.3 hours (≈ 4.76 days)

length 6: 26^6 = 308,915,776 guesses

Time ≈ 308,915,776 / 28.9 ≈ 10,703,000 s ≈ 124.0 days

length 7: 26^7 = 8,031,810,176 guesses

Time ≈ 8.03e9 / 28.9 ≈ 278,394,000 s ≈ ≈ 8.83 years

length 8: 26^8 = 208,827,064,576 guesses

Time ≈ 2.088e11 / 28.9 ≈ 7.234e9 s ≈ ≈ 229.3 years

How long it takes the cracker to crack one password in type 2 and type 3 for various bounds:

For the 3–4 lowercase keyspace I measured the test times above; scaled to the assignment iteration count (100,000) the same keyspace would take roughly 4.56–4.57 hours on my machine for both type-2 and type-3 files.

I did not run exhaustive tests for lengths 5–8 because they require days/years of CPU time; I predicted their times with the method shown above. (Predictions: length-6 ≈ 124 days, length-7 ≈ 8.8 years, length-8 ≈ 229 years for lowercase only.)

Which format is more secure?

Type-3 (salted + PBKDF2) is more secure in practice. My measurements show nearly identical per-password brute-force times for a single target (because each guess must be hashed with the known salt), but salt prevents precomputed/rainbow attacks and prevents one cracked password from immediately revealing the same password in other accounts. Therefore salted storage is the better practice.

Concrete numbers to compare formats:

On the 3–4 lowercase test, scaled runtimes were ~4.57 hours (type-2) vs ~4.56 hours (type-3) — effectively the same for a single account brute force. The security advantage of salt is not in

increasing single-account brute force time but in preventing mass precomputation and reuse attacks.

Minimum password length to be “secure” on my system (justified):

Based on the measured speeds and predictions above, a lowercase-only password of length 8 would require on the order of hundreds of years to brute force by exhaustive search on my machine, so I recommend minimum 8 characters if only lowercase is used. If you allow uppercase, digits and symbols the required length for comparable security may be shorter; conversely, if passwords are dictionary-based, a wordlist attack would be far faster and the length requirement changes.

What I did NOT do / why

I did not exhaustively brute force longer ranges (5–8) in real time because the full keyspace run times are impractically long (days to centuries). Instead I used the measured speed from a small, fully exhausted keyspace and scaled the times mathematically to estimate the longer runs. I documented the scaling method above so these predictions are reproducible.

Project 3 Summary:

I ran controlled brute-force experiments on both the type-2 (hashed) and type-3 (salted) password files generated for the assignment using lowercase character sets. For a small, fully exhausted keyspace (lengths 3–4; 474,552 guesses) I measured 164.37 s for the hashed file and 164.03 s for the salted file with a reduced test iteration count; scaled to the assignment’s PBKDF2 iteration count (100,000) those tests estimate to ≈4.56–4.57 hours. I did not run full exhaustive tests for lengths 5–8 (they are prohibitively long), but I mathematically predicted their run times (length 6 ≈ 124 days, length 7 ≈ 8.8 years, length 8 ≈ 229 years at lowercase-only). Salted PBKDF2 is the preferred format because it prevents precomputation and mass-reuse attacks; I recommend a minimum password length of 8 (lowercase only) or stronger (with mixed case/digits/symbols) for practical protection on my machine.

auth_task1.py:

```
#!/usr/bin/env python3

import os
import sys
import hashlib
import secrets
from getpass import getpass
from typing import Optional, Tuple
```

```python
# Config
PLAIN_FILE = "passwords_plain.txt"
HASH_FILE = "passwords_hashed.txt"
SALTED_FILE = "passwords_salted.txt"

# Hashing params
HASH_NAME = "sha256"
ITERATIONS = 100_000
SALT_SIZE = 16  # bytes


# ---------- Utilities ----------
def valid_username(name: str) -> bool:
    """Username must be alphabetic only and up to 10 chars."""
    return name.isalpha() and 1 <= len(name) <= 10


def pbkdf2_hash(password: bytes, salt: bytes, iterations=ITERATIONS) -> bytes:
    """Return raw bytes of PBKDF2-HMAC-SHA256."""
    return hashlib.pbkdf2_hmac(HASH_NAME, password, salt, iterations)


def read_map_from_file(filename: str) -> dict:
    """Read file lines into a dict. Handles various formats."""
    if not os.path.exists(filename):
        return {}
    d = {}
    with open(filename, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            parts = line.split(":")
            # Unknown formats handled by how caller expects contents
            d[parts[0]] = parts[1:]
    return d


def username_exists(username: str) -> bool:

    for fname in (PLAIN_FILE, HASH_FILE, SALTED_FILE):
        if username in read_map_from_file(fname):
            return True
    return False


# ---------- Account Creation ----------
def create_account():
    print("\n--- Create account ---")
```

```
    username = input("Enter username (alphabetic only, max 10 chars): ").strip()
    if not valid_username(username):
        print("Invalid username. Must be alphabetic only and 1-10 chars.")
        return

    if username_exists(username):
        print("Username already exists in one of the password files. Choose
another.")
        return

    password = getpass("Enter password: ")
    password2 = getpass("Re-enter password: ")
    if password != password2:
        print("Passwords do not match. Aborting.")
        return

    # Write plaintext
    with open(PLAIN_FILE, "a", encoding="utf-8") as f:
        f.write(f"{username}:{password}\n")

    empty_salt = b""
    hashed = pbkdf2_hash(password.encode("utf-8"), empty_salt)
    with open(HASH_FILE, "a", encoding="utf-8") as f:
        f.write(f"{username}:{hashed.hex()}\n")

    # Write salted: generate salt and store salt:hash
    salt = secrets.token_bytes(SALT_SIZE)
    salted_hash = pbkdf2_hash(password.encode("utf-8"), salt)
    with open(SALTED_FILE, "a", encoding="utf-8") as f:
        f.write(f"{username}:{salt.hex()}:{salted_hash.hex()}\n")

    print("Account created and data written to all 3 password files.")


# ---------- Authentication ----------
def auth_against_plain(username: str, guess_password: str) -> bool:
    data = read_map_from_file(PLAIN_FILE)
    entry = data.get(username)
    if not entry:
        return False
    stored = entry[0]
    return guess_password == stored


def auth_against_hash(username: str, guess_password: str) -> bool:
    data = read_map_from_file(HASH_FILE)
    entry = data.get(username)
    if not entry:
        return False
```

```
    stored_hex = entry[0]

    empty_salt = b""
    guess_h = pbkdf2_hash(guess_password.encode("utf-8"), empty_salt)
    return guess_h.hex() == stored_hex


def auth_against_salted(username: str, guess_password: str) -> bool:
    data = read_map_from_file(SALTED_FILE)
    entry = data.get(username)
    if not entry or len(entry) < 2:
        return False
    salt_hex, stored_hex = entry[0], entry[1]
    try:
        salt = bytes.fromhex(salt_hex)
    except ValueError:
        return False
    guess_h = pbkdf2_hash(guess_password.encode("utf-8"), salt)
    return guess_h.hex() == stored_hex


def authenticate():
    print("\n--- Authenticate ---")
    username = input("Username: ").strip()
    if not valid_username(username):
        print("Invalid username format. Must be alphabetic only and 1-10
chars.")
        return

    password = getpass("Password: ")

    # Check each file independently and print separate messages
    ok_plain = auth_against_plain(username, password)
    ok_hash = auth_against_hash(username, password)
    ok_salted = auth_against_salted(username, password)

    # Output separate messages for each verification
    print("\nVerification results:")
    print(f"- Plaintext file ({PLAIN_FILE}): {'SUCCESS' if ok_plain else
'FAIL'}")
    print(f"- Hashed file ({HASH_FILE}): {'SUCCESS' if ok_hash else 'FAIL'}")
    print(f"- Salted file ({SALTED_FILE}): {'SUCCESS' if ok_salted else
'FAIL'}")

    # Overall info
    if ok_plain or ok_hash or ok_salted:
        print("\nAt least one password file verified the credentials.")
    else:
        print("\nNo password file verified the credentials.")
```

```
# ---------- CLI ----------
def main():
    print("Project 3 - Task 1 (Python)")
    while True:
        print("\nOptions:")
        print("1) Create account")
        print("2) Authenticate")
        print("3) Exit")
        choice = input("Choose (1/2/3): ").strip()
        if choice == "1":
            create_account()
        elif choice == "2":
            authenticate()
        elif choice == "3":
            print("Goodbye.")
            break
        else:
            print("Invalid option. Try again.")


if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("\nInterrupted. Exiting.")
        sys.exit(0)
```

task2_generate.py:

```
#!/usr/bin/env python3

import os
import string
import secrets
import hashlib
from typing import Tuple, Set

PLAIN_OUT = "task2_plain.txt"
HASH_OUT = "task2_hashed.txt"
SALTED_OUT = "task2_salted.txt"


HASH_NAME = "sha256"
ITERATIONS = 100_000
SALT_SIZE = 16  # bytes
```

```
def pbkdf2_hash(password: bytes, salt: bytes) -> bytes:
   return hashlib.pbkdf2_hmac(HASH_NAME, password, salt, ITERATIONS)


def ask_int(prompt: str, default: int = None, min_v: int = None, max_v: int =
None) -> int:
   while True:
       raw = input(f"{prompt}{' ['+str(default)+']' if default is not None else
''}: ").strip()
       if not raw and default is not None:
           val = default
       else:
           if not raw.isdigit():
               print("Please enter a whole number.")
               continue
           val = int(raw)
       if min_v is not None and val < min_v:
           print(f"Value must be >= {min_v}.")
           continue
       if max_v is not None and val > max_v:
           print(f"Value must be <= {max_v}.")
           continue
       return val


def generate_username(i: int, used: Set[str]) -> str:

   base = "user"
   letters = string.ascii_lowercase

   suffix = []
   x = i
   for _ in range(6):
       suffix.append(letters[x % 26])
       x //= 26
   name = (base + "".join(suffix))[:10]  # ensure <=10

   while name in used or not name.isalpha() or len(name) == 0 or len(name) >
10:
       name = (base + "".join(secrets.choice(letters) for _ in range(6)))[:10]
   used.add(name)
   return name


def generate_password(min_len: int, max_len: int) -> str:

   letters = string.ascii_lowercase
   length = secrets.randbelow(max_len - min_len + 1) + min_len
```

```
    return "".join(secrets.choice(letters) for _ in range(length))


def main():
    print("=== Task 2: Random Password File Generator ===")

    # Prompt for parameters
    n_accounts = ask_int("Number of accounts to generate", default=100, min_v=1,
max_v=100000)
    min_len = ask_int("Minimum password length", default=3, min_v=1, max_v=100)
    max_len = ask_int("Maximum password length", default=8, min_v=min_len,
max_v=100)

    # Confirm overwrite if files exist
    out_files = [PLAIN_OUT, HASH_OUT, SALTED_OUT]
    existing = [f for f in out_files if os.path.exists(f)]
    if existing:
        print("\nThe following files already exist and will be OVERWRITTEN:")
        for f in existing:
            print(" -", f)
        resp = input("Continue? (y/n): ").strip().lower()
        if resp != "y":
            print("Aborted. No files written.")
            return

    used_usernames: Set[str] = set()

    with open(PLAIN_OUT, "w", encoding="utf-8") as f_plain, \
         open(HASH_OUT, "w", encoding="utf-8") as f_hash, \
         open(SALTED_OUT, "w", encoding="utf-8") as f_salt:

        for i in range(n_accounts):
            username = generate_username(i, used_usernames)
            password = generate_password(min_len, max_len)

            # type 1: plaintext
            f_plain.write(f"{username}:{password}\n")

            # type 2: hashed (no salt) - PBKDF2 with empty salt to match Task 1
            empty_salt = b""
            h2 = pbkdf2_hash(password.encode("utf-8"), empty_salt).hex()
            f_hash.write(f"{username}:{h2}\n")

            # type 3: salted
            salt = secrets.token_bytes(SALT_SIZE)
            h3 = pbkdf2_hash(password.encode("utf-8"), salt).hex()
            f_salt.write(f"{username}:{salt.hex()}:{h3}\n")

    print("\nDone!")
```

```
    print(f"- Plaintext file: {PLAIN_OUT}")
    print(f"- Hashed file   : {HASH_OUT}")
    print(f"- Salted file   : {SALTED_OUT}")


if __name__ == "__main__":
    main()
```

task3_cracker.py:

```
#!/usr/bin/env python3

import time
import itertools
import os
import sys
import hashlib
from typing import Tuple, Optional

# Params must match Task 1/2
HASH_NAME = "sha256"
ITERATIONS = 100_000

def pbkdf2_hash(password: bytes, salt: bytes) -> bytes:
    return hashlib.pbkdf2_hmac(HASH_NAME, password, salt, ITERATIONS)

def load_file(path: str):

    if not os.path.exists(path):
        raise FileNotFoundError(path)
    entries = []
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            parts = line.split(":")
            if len(parts) == 2:
                entries.append(tuple(parts))  # (username, hash_hex)
            elif len(parts) >= 3:
                # treat first as username, second as salt, rest joined as hash
(handles stray colons)
                username = parts[0]
                salt_hex = parts[1]
                hash_hex = ":".join(parts[2:])
                entries.append((username, salt_hex, hash_hex))
            else:
                # unexpected format
                print("Skipping unknown line format:", line)
```

```
    return entries

def try_bruteforce(target_hash_hex: str, salt_bytes: bytes, charset: str,
min_len: int, max_len: int, show_every: int=10000) -> Optional[str]:

    target_hash = target_hash_hex.lower()
    total_checked = 0
    start = time.time()
    # iterate lengths from min_len to max_len
    for L in range(min_len, max_len + 1):
        # product yields tuples of characters
        for tup in itertools.product(charset, repeat=L):
            guess = "".join(tup)
            guess_bytes = guess.encode("utf-8")
            h = pbkdf2_hash(guess_bytes, salt_bytes).hex()
            total_checked += 1
            if total_checked % show_every == 0:
                elapsed = time.time() - start
                rate = total_checked / elapsed if elapsed > 0 else 0
                print(f"[progress] checked {total_checked:,} guesses —
{rate:,.0f} guesses/sec — current length {L} guess '{guess}'")
            if h == target_hash:
                elapsed = time.time() - start
                print(f"[FOUND] password='{guess}' after {total_checked:,}
guesses in {elapsed:.2f} seconds")
                return guess
    elapsed = time.time() - start
    print(f"[DONE] Exhausted keyspace ({total_checked:,} guesses) in
{elapsed:.2f} seconds — not found")
    return None

def interactive():
    print("\n=== Task 3: Brute-force password cracker ===")
    path = input("Path to password file (e.g. task2_hashed.txt or
task2_salted.txt): ").strip()
    try:
        items = load_file(path)
    except FileNotFoundError:
        print("File not found:", path)
        return

    if not items:
        print("No entries found in the file.")
        return

    print(f"Loaded {len(items)} entries.")
    # show a few entries
    for i, it in enumerate(items[:10], start=1):
        print(f"{i:2d}) {':'.join(it)}")
```

```
    print("...")

    # select entry to crack
    sel = input("Enter the username to crack (or enter a line number):
").strip()
    selected = None
    # if numeric, select that index
    if sel.isdigit():
        idx = int(sel) - 1
        if 0 <= idx < len(items):
            selected = items[idx]
        else:
            print("Invalid index.")
            return
    else:
        # find username
        for it in items:
            if it[0] == sel:
                selected = it
                break
        if selected is None:
            print("Username not found in file.")
            return

    # Derive target info depending on tuple length
    if len(selected) == 2:
        # type 2 (hashed): (username, hash_hex)
        username, hash_hex = selected
        salt_bytes = b""  # empty salt for type 2
        print(f"Target: {username}  (type 2 hashed, no salt)")
    else:
        username, salt_hex, hash_hex = selected
        try:
            salt_bytes = bytes.fromhex(salt_hex)
        except Exception as e:
            print("Invalid salt hex for entry:", e)
            return
        print(f"Target: {username}  (type 3 salted)")

    # charset prompt
    print("\nChoose charset option:")
    print("1) lowercase letters (a-z)  [fastest]")
    print("2) lowercase + digits (a-z0-9)")
    print("3) lowercase+upper+digits (a-zA-Z0-9)  [much larger keyspace]")
    print("4) Custom (type characters)")
    ch = input("Choice [1]: ").strip() or "1"
    if ch == "1":
        charset = "abcdefghijklmnopqrstuvwxyz"
    elif ch == "2":
```

```
        charset = "abcdefghijklmnopqrstuvwxyz0123456789"
    elif ch == "3":
        charset =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    else:
        charset = input("Enter characters to use (no spaces): ").strip()
        if not charset:
            print("Invalid charset.")
            return

    min_len = input("Minimum password length [3]: ").strip()
    min_len = int(min_len) if min_len.isdigit() else 3
    max_len = input("Maximum password length [8]: ").strip()
    max_len = int(max_len) if max_len.isdigit() else 8
    if min_len > max_len:
        print("Min length must be <= max length.")
        return

    # estimate keyspace
    keyspace = sum(len(charset) ** L for L in range(min_len, max_len + 1))
    print(f"\nEstimated keyspace to try: {keyspace:,} guesses")

    confirm = input("Start brute-force? (y/n) [n]: ").strip().lower() or "n"
    if confirm != "y":
        print("Aborted.")
        return

    print("Starting brute-force. This can take time depending on keyspace and
your CPU...")
    starttime = time.time()

    found = try_bruteforce(hash_hex, salt_bytes, charset, min_len, max_len)

    total_time = time.time() - starttime
    if found:
        print(f"\nSUCCESS: '{username}' password = '{found}' (found in
{total_time:.2f} seconds)")
    else:
        print(f"\nNo password found for '{username}' in specified keyspace
({min_len}-{max_len}, charset size {len(charset)}) after {total_time:.2f}s")

if __name__ == "__main__":
    interactive()
```