What the Program Does:

For my final project, I created a Password Strength Analyzer & Generator. The program is a terminal-based application written in C++. It allows users to input a password, and the program will analyze the strength of that password based on several factors such as length, use of special characters, and entropy (a measure of randomness). The program also suggests improvements if the password is weak. Additionally, the program can generate a strong password according to user preferences, such as length and whether special characters are included.

This program is useful because it helps users create stronger, more secure passwords and encourages good password practices. In today's world, cybersecurity is really important, and passwords are one of the first lines of defense. With this tool, people can ensure they are using passwords that are harder for hackers to guess or crack.

Algorithms Used:

1. Password Entropy Calculation Algorithm:

- Purpose: The entropy calculation is used to determine how random and secure a password is.

- Steps:

  1. Count the total number of possible characters in the password (e.g., if it includes uppercase, lowercase, digits, and special characters).

  2. Calculate the entropy using the formula:
     Entropy=log2(Number of possible combinations)\text{Entropy} = \log_2(\text{Number of possible combinations})Entropy=log2(Number of possible combinations)
  3. The higher the entropy, the more secure the password is.

Code Snapshot:

cpp
CopyEdit

```cpp
double calculateEntropy(std::string password) {
    int numCharacters = password.length();
    int numCombinations = 0;

    // Example: Uppercase, lowercase, digits, special characters
    numCombinations = 26 + 26 + 10 + 32;

    double entropy = numCharacters * log2(numCombinations);
    return entropy;
}
```

-
- Big O: The time complexity is O(n), where n is the length of the password, because we loop through each character to calculate entropy.

2. Password Strength Analysis Algorithm:

- Purpose: This algorithm analyzes whether the password follows common security guidelines (length, mixed character types, etc.).
- Steps:
    1. Check if the password is at least 8 characters long.
    2. Check if the password includes lowercase, uppercase, digits, and special characters.
    3. Return the password strength: "Strong", "Medium", or "Weak" based on these checks.

Code Snapshot:

cpp
CopyEdit

```
std::string analyzePasswordStrength(std::string password) {
    if (password.length() < 8) {
        return "Weak";
    }

    bool hasLower = false, hasUpper = false, hasDigit = false,
hasSpecial = false;
    for (char ch : password) {
        if (islower(ch)) hasLower = true;
        else if (isupper(ch)) hasUpper = true;
        else if (isdigit(ch)) hasDigit = true;
        else if (ispunct(ch)) hasSpecial = true;
    }

    if (hasLower && hasUpper && hasDigit && hasSpecial) {
        return "Strong";
    }
    return "Medium";
}
```

-
- Big O: The time complexity is O(n), where n is the length of the password because we loop through each character to check its type.

3. Random Password Generation Algorithm:

- Purpose: This algorithm generates a strong password with user-defined preferences such as length and whether special characters are included.
- Steps:
    1. Take user input for password length and special character preference.
    2. Generate random characters and ensure the password meets the user's criteria (length and special characters).
    3. Return the generated password.

Code Snapshot:

cpp
CopyEdit

```
std::string generateRandomPassword(int length, bool includeSpecial) {
    std::string password = "";
    std::string allChars =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    std::string specialChars = "!@#$%^&*()_+";

    for (int i = 0; i < length; ++i) {
        char ch = allChars[rand() % allChars.length()];
        password += ch;
    }

    if (includeSpecial) {
        password += specialChars[rand() % specialChars.length()];
    }

    return password;
}
```

-
- Big O: The time complexity is O(n), where n is the length of the password, because the loop runs once for each character in the password.

Data Structures Used:

1. String:
    - Purpose: A `string` is used to store and manipulate passwords. It's a simple and effective way to store sequences of characters.
    - Reason for Choosing: Passwords are text-based, and `std::string` provides efficient ways to access and modify characters.
    - How it Was Used: I used strings to input the user's password, analyze it, and generate random passwords.
2. Queue:
    - Purpose: A queue is used to store password history (when a user chooses to view past passwords).
    - Reason for Choosing: A queue is ideal because passwords are stored and retrieved in the order they were added (FIFO – First In, First Out).
    - How it Was Used: I created a queue to store and display password history when the user chooses to view previous passwords.
3. Map:
    - Purpose: A map is used to store password records (password, entropy, timestamp) for each password entered.

- Reason for Choosing: A map allows for efficient searching by password and is ideal for storing key-value pairs (password, strength, etc.).
- How it Was Used: The map stores and displays the analysis of past passwords with their corresponding strength and timestamp.

Design/Development Process:

1. Opportunity Encountered:
   One opportunity I encountered was when designing the random password generation. Initially, I was only generating passwords with alphanumeric characters, but I added an option to include special characters to make the password stronger. This allowed me to offer users more customization.
2. Error Encountered and Resolution:
   A challenge I encountered was handling the randomness of the password generation. Initially, my code would sometimes generate weak passwords because it didn't enforce the inclusion of all character types (uppercase, lowercase, numbers, and special characters). I resolved this by ensuring that the generated password would always meet the user’s criteria.

What I Would Change or Add:

In the next version, I would add a feature to check if the password is part of a dictionary (a list of common passwords or words). This would prevent users from using easily guessable passwords. I would also like to add a GUI for better user experience, but for this project, I kept it terminal-based to focus on core programming concepts.