For tasks one until three I implemented simple algorithms that cracked the passwords in a very short amount of time. For task 1, used the “product” method of “itertools” to generate all possible strings out of the possible password characters, which in this case were a-z and 1-9. The time it takes to generate these strings increases exponentially with the length of the strings, so I programmed the function to stop when the number of cracked passwords matches the number of input passwords, so no strings are generated unnecessarily. This greatly decreased run time. For the second task, I imported the dictionary using urlopen, split the dictionary into lines (each line was a different password), hashed the passwords, and compared them with the input passwords. For task three, I implemented the same method, but I added the salts to the end of the passwords before hashing them, and then removed them after by removing the last 8 characters of the string.

For task 4, I chose to implement “password mangling” into my assignment. I had to research this topic, as I was not very familiar with it. I had heard of John the Ripper before, as I have an interest in cyber-attacks. John the Ripper is an open-source password cracker and is pretty much the ultimate implementation of password mangling and cracking. It was programmed in C, although scripts to run it in python exist. I, however, only sought to try and crack mangled passwords. Mangled passwords are passwords which swap letters with numbers and symbols. The most common rules (which I used) are:

* Replace ‘a’ with ‘@
* Replace ‘e’ with ‘3’
* Replace ‘i’ with ‘!’
* Replace ‘o’ with ‘0’
* Replace ‘s’ with ‘$’
* Replace ‘t’ with ‘7’

This technique is wrongly assumed to make passwords uncrackable when the technique follows some predictable rules. I am guilty myself of using these replacements, thinking that my password is less guessable. While it does improve upon basic passwords, mangled passwords can be generated from a word list relatively quickly. Thus, mangled passwords are susceptible to a dictionary attack if a mangling function is implemented. This was my goal for this task.

My algorithm was very basic, as I wanted something that would work, and I could not implement anything more ambitious in time. My attempts did not run properly. My basic algorithm would run through the wordlist (the same dictionary as used in task two and three), hash the passwords, and check them against the inputted passwords. This was the same as my implementation of the dictionary attack. The next step is the additional step which accounted for the possibility of password mangling techniques being used. If there was no match between the password from the wordlist and the passwords inputted (both in hashed form), then the function “passwordMangler” would use the “replace” method to swap letters with numbers and symbols according to the aforementioned mangling rules. In developing this method, I initially converted the password string to a tuple of characters, replacing the characters as instances of them were found. However, this caused issues, as the function was only mangling one instance of a character at a time. In finding a fix, I discovered that the “replace” method was simpler and fixed the issue.

After the password was mangled, it would be encoded, hashed, and checked against the inputted passwords. If no match was found, the algorithm would move onto the next password. It would repeat this for all the passwords in the dictionary. Were the dictionary to be exceptionally long, I would have implemented a feature to stop the algorithm when the list of cracked passwords matches in length to the list of inputted passwords, but password lists like the one I used are typically of manageable length and thus have a good runtime. My algorithm ran in under a second.