**Task 1:**

The idea of the solution to task 1 was to generate every possible string over the given alphabet, generating them in shortlex order. This was achieved by looping through the list of hashes that needed to be cracked, trying to break one at a time. Then for each hash the programme enters a while loop that runs until the hash password pair I found. Within this loop is a for loop, generating every string of the alphabet in shortlex order, and generating the related hash in sha512 for this string. If there is a match, it breaks the for loop, changes the condition of the while loop as to finish the while loop, then moves onto the next hash needing to be brute forced, then repeats the process. When the algorithm completes it returns a list containing the cracked passwords in plaintext.

**Task 2:**

Task 2 was to design an algorithm implementing a dictionary attack. For this, the function took in not only the hashes needing cracked, but also the dictionary of common passwords. The function then ensures the file pointer is at the beginning, before initiating a python dictionary for password hash pairs that have already been generated to be stored in. The function then loops through the hashes given to crack, first checking if a matching password has already been generated by checking the python dictionary. If so, the function saves the plaintext password to a list. If not, the function runs through each password in the passed in dictionary generating the hash for it, starting from wherever the pointer is left from previous runs, ensuring no repetition in hash generation. It then saves this password hash combo to the python dictionary. If the hash generated is the same as the one trying to be cracked, the plaintext is saved to the output and the function moves to the next hash to crack, if not it continues to search through the dictionary.

**Task 3:**

Task 3 was designed to do the same as task 2, however now with salted passwords. To do this, the code for task 2 was initially copied across. This function takes in a list of tuples, with the hash, and the plaintext salt used for this password. As well as this, it also has a password dictionary input as in task 2. In the function, I removed the python dictionary, and any code checking its’ contents. This was done because each password has a unique salt, so storing hash-password pairs would be meaningless. The function now takes a hash salt pair from the input, separates the two strings, and appends the hash to each password in the dictionary before finding its hash. If the hash matches then it is appended to the list of found passwords and resets the file pointer to the start for the next hash, if not, it continues to parse the dictionary.

**Task 4:**

**Appendix:**

**Task 1:**

def function\_1(hash\_list):  
 start = time.time()  
 alphabet = string.ascii\_lowercase + string.digits  
 answers = []  
 for hash in hash\_list:  
 counter = 1 #Used for progressively increasing length of string generated  
 found = 0 # Create a variable for if item has been found to allow for skipping of current run without breaking from entire loop  
 while found != 1: # While loop runs until password is found  
 for string1 in map(''.join, itertools.product(alphabet, repeat=counter)): # Use of itertools idea from: https://stackoverflow.com/questions/16347583/how-to-generate-all-possible-strings-in-python  
 if hashlib.sha512(bytes(string1, 'ascii')).hexdigest() == hash: # Computes hash of generated string, compares to given hash  
 answers.append(string1)  
 found = 1  
 break  
 counter += 1  
  
 print(answers)  
 end = time.time()  
 print(end - start)  
 return answers

**Task 2:**

# Task 2: Use given dictionary to generate hashes of common passwords and compare.  
def function\_2(passdict, hashes):  
 passdict.seek(0) # In case of prior access to file, reset pointer to start  
 start = time.time()  
 hashdict = {} # Use Python Dictionary, keys: Hashes, contents: password  
 answers = []  
 for hash in hashes:  
 if hash in hashdict: # If hash is already computed from previous running, then no need to compute  
 answers.append(hashdict[hash])  
 else: # Strips newlines, calcs hash, adds to dict, and compares to see if it matches given hash  
 for password in passdict:  
 password = password.strip()  
 newhash = hashlib.sha512(bytes(password, 'ascii')).hexdigest()  
 hashdict[newhash] = password  
  
 if newhash == hash:  
 answers.append(password)  
 break  
  
 print(answers)  
 end = time.time()  
 print(end-start)  
 return answers

**Task 3:**

# Task 3: To crack salted passwords given salt. Reused code from task 2, removing hashdict  
def function\_3(passdict, hashes):  
 start = time.time()  
 answers = []  
 for hash in hashes:  
 passdict.seek(0) # Resets pointer to start for each hash as no precalculating because of unique salts  
 for password in passdict:  
 i\_salt = password.strip()+hash[1] # Strips newlines, adds salt  
 newhash = hashlib.sha512(bytes(i\_salt, 'ascii')).hexdigest() # calcs hash  
  
 if newhash == hash[0]: # If hashes match, stores passwords, stripping newlines  
 answers.append(password.strip())  
 break  
  
 print(answers)  
 end = time.time()  
 print(end - start)  
 return answers