Task A:

The original `find_prime_factors` function had an error in its signature, where the default value for the parameter `prime_factors` was an empty list; while the function will work fine the first time it is called, any further call will see an incorrect list of prime factors, since the list object itself remains the same. For example, if we make two calls to get the prime factors of `10` and `15`, we would expect the first call to return `[2, 5]` and the second call to return `[3, 5]`, however the second call will instead return `[2, 5, 3, 5]`. This bug is relatively simple to fix: instead of having `prime_factors` having an empty list as a default value, we set it to `None`, and when running the function create a new list if no list was passed.

A second possible error is the fact that the same factor can be repeated multiple times: if we wish to find the prime factors of, for example, `8`, the function would return `[2, 2, 2]`, as `8` can be divided by two three times. There are several solutions to this, such as converting the list to a set, and then back to a list, although none were implemented as it is technically not an incorrect result.

A series of tests were written to test the function: first, the function is tested in a range from `0` to `999_999`, incrementing the value by one; then from `1_000_000` to `1_000_000_000_000`, incrementing the tested value by a factor of ten (i.e., multiplied by 10). The `primefac` library is used to test the correctness of the results, as it is a known good prime factorization library. These same tests are used in order to test the correctness of the optimized version of the function.

The main way this function can be optimized is by separating the process by which all even factors are “extracted”, from the rest: by factorizing all 2’s out, we are able to significantly reduce the number of iterations (in the main loop) the function has to do, as well as the number of operations. For example, instead of checking every number from `2` to `$\sqrt{n}$`, we are able to instead check every odd number, starting from `3`; additionally, we can cut in half the number of multiplications we have to do (from the `while i * i <= n:` step), which will lead to large time savings, especially with larger numbers.

While the function is relatively readable, there are some ways to improve it further: one such way is adding a docstring and type hints, making it both simpler to use and its purpose more clear. Additionally, adding some extra spacing between code blocks, and some comments will further help when reading the code, both making the code flow simpler to keep in mind and clearer to a programmer unfamiliar with the code.

As mentioned before, one bug found was the default value being an empty list for one of the function parameters; the solution was relatively simple, it being changing it to default to `None`, and instead creating a new list if none is passed by the user. Additionally, the

function was changed to handle factorization of all even numbers separately, simplifying the factorization process. This lead to real time savings, with the optimized function cutting its runtime by half when factorizing the first million integers.

Finally, an additional, optional parameter was added to the new, optimized function, allowing the user to remove duplicates if so desired. This defaults to `False` to conform with most other implementations of this algorithm (as they all include duplicates).

The code complexity is $O(\sqrt{n})$, as if the number to be factorized is a perfect square we will need to check every number from `2` to `$\sqrt{n}$`.

Task B:

Seeing as most bank accounts have similar functions, allowing the user to withdraw and deposit cash, it was worth using inheritance to avoid rewriting the same code multiple times. A base `BankAccount` class was written with a constructor, a method to deposit money, one to withdraw it, and finally a way to get account information, in the form of a string containing the account holder and balance.

As a checking account has a fee for withdrawals, the base `withdraw` method is overridden, to add the fee to the requested amount; this value is then passed to the base `withdraw` method by calling `super()`.

The savings account remains the same as the base class, except for a new method to apply interest. This will simply multiply the current balance by 1.025, effectively adding an interest of 2.5% on the balance.

One change that was done compared to the original `BankAccount` class was to return the available balance from the `deposit` and `withdraw` methods, allowing to test the code programmatically and using assertions, rather than having to look at a console output; a new method was also added to return the available balance, for the same reason.

Additionally, both the transaction fee for the checking account and the interest rate for the savings account were created as class members rather than instance members; the reason for this choice is that generally these tend to be the same for all accounts, which means that it can make sense that it should be the same object for all instances.

A series of test cases were written to demonstrate and test the functionality of the three classes, such as testing that depositing and withdrawing money works as expected, and for checking and savings accounts respectively, that the transaction fee is also subtracted from the balance and that the interest rate is applied correctly. Assertions were used during tests, easily allowing a tester to see whether a test has failed or all have succeeded.