Thue-morse sequence:

The program starts off with function main where the user is asked to input a number which is held in the variable *n*, which is then passed to the function *thue_morse().*

In *thue_morse()*, the variable *t* is created and given the value 0, the first character in the sequence. This variable is then used inside a loop that adds the two halves of the equation $\mathbf{tn-1}$ and inverse $\overline{\mathbf{tn-1}}$ together to create a thue-morse sequence, this loop occurs for the number of times given by *n*.

To get the inverse of the first half of the string, another auxiliary function is called named *find_inverse()* which returns the opposite character that is found at the current index in the string. This works by selecting each character found in the given iteration of t from *thue_morse()* and comparing it against the value 0. Using == to compare the statement is true if the value at the current index of *t* is 0 meaning that 1 will be added to a new string called inverse, else if false then it is assumed that the value within *t* is 1 and so a 0 is added to the new string instead.

This inversed string is returned to *thue_morse()* where it is added to its complement to create a longer thue-morse sequence, which then can be used in the next iteration to create a larger sequence until n iterations have been completed, the final sequence is then returned to function main to be printed into the terminal.

Using the representative samples, the code gives the following output:

```
Input a positive integer: 0
0
PS C:\Users\David\OneDrive -
free.py
Input a positive integer: 1
01
PS C:\Users\David\OneDrive -
free.py
Input a positive integer: 2
0110
PS C:\Users\David\OneDrive -
free.py
Input a positive integer: 3
01101001
```

Giving the program a value of 0 gives 0

Giving the program a value of 1 gives 01

Giving the program a value of 2 gives 0110

Giving the program a value of 3 gives 01101001

All these outputs matches the expected output

Square free:

Program starts with asking the user for a positive integer. If the value, *n*, passed to *square_free()* is >=0 then *n* is used in a for loop for the number of iterations that the auxiliary function, *replace()*, will occur. Function *replace()* is passed the arguments *a* and *count* which are both set to a value of 1 initial, of datatype string and integer respectively. This auxiliary function replaces each character found in *a* depending on the value found at the current index and whether the index position of the value is even or odd, this being determined by another function called *is_even()* that returns true if the modulus of the current index, as tracked by variable *count*, equals 0 as it means the index number is even or false if it is odd.

The current character at a specified index of *a* is replaced by the corresponding value as determined by the logic mentioned above. Replacement is done by adding the new string to a variable *new_str*, which originally starts as an empty string each time the function *replace()* is called by the for loop in *square_free()*. This variable is returned to *square_free()* when the end of string *a* is reached by the *replace()* for loop.

This new value of *a* is then given to the function *replace()* again until the end of range(n) is reached.

Using the representative samples, the code gives the following output:

```
Input a positive integer: 0
1
PS C:\Users\David\OneDrive - Federat
e_free.py
Input a positive integer: 1
123
PS C:\Users\David\OneDrive - Federat
e_free.py
Input a positive integer: 2
123 132 312
PS C:\Users\David\OneDrive - Federat
e_free.py
Input a positive integer: 3
123 132 312 321 312 132 312 321 231
```

Giving the program a value of 0 gives 1

Giving the program a value of 1 gives 123

Giving the program a value of 2 give 123 132 312

Giving the program a value of 3 gives 123 132 312 321 312 132 312 321 231

All these outputs matches the expected output

Count squares

The user is first asked to input a sequence of characters, this is then passed to *count_squares()* as *s* where the search for squares in the string occurs.

This is done by using nested for loops, the outer most loop defines the size of the search as assigned to variable *search_group*. This is done by creating a range from one to half the length of *s.* This is done as it'll be impossible to find a square whose size is larger than half the length of the string. To ensure that a float is not created, int division (//) is used. 1 is then added to this number as *range()* is half-inclusive.

The next nested for loop, the first inner loop, defines the number of times a *search_group* is applied to the string to find squares. This is done by dividing the length of *s* by the value of the current *search_group* as it'll be impossible to find anymore squares beyond this number as the *search_group* will be looking beyond the length of the string, again int division is used to avoid a float. This value is then minus by 1 as again it'll take the *search_group* beyond the length of the string otherwise.

The inner-most for loop accounts for the offset that is needed per each *search_group*. For example, given the string "123123123" when *search_group* = 3, by only starting the search at 0, only two squares will be found when really four should be found within this search iteration. These other two squares, "231231" and "312312" can only be found if the search starts from index 1 and 2 respectively. The value offset accounts for this, starting from 0, it increases by 1 at the start of every iteration of the first inner loop to shift the start of the search by 1 to the right find these other squares.

All these for loops values are created to allow for slice ranges that are dynamic and able to be applied to strings of whatever length. The elif statement compares two substrings as returned by the slice range based on the current iteration, if the values match, then 1 is added to the int variable *num_squares*.

A if statement is used prior to this to handle exceptions caused by the *search_group* searching beyond the length of the string despite best efforts to avoid this using the above for loops.

Once the for loops are complete, *num_squares* is returned to be printed in *main()* by order substitution.

Negative integers weren't used in string slices due to the half-inclusive nature of them. Using negative integers in the slices would mean not all cases could be handled the same, for example to select the last two characters with negatives it'll be [-2:] whilst other characters would be [-3:-1].

Using the representative sample, the code gives the following output:

```
Input numbers 1-3 in a sequence to see number of squares present: 1231233
Number of squares in 1231233: 2 squares
```

Giving the program a value of 1231233 gives 2 squares which matches the expected output.