CS1001 Pset 1 - Tom P. Bleher

Problem 2

Solution 2.a

For all three cases, the input of the user to the function is not valid. To counter these undesired inputs, the function has the `assert` lines, which require that the input will be of type string of length 8.

For the cases when the code encounters an invalid input that is expected, such as the cases for cases, (a.) and (b.), the `assert` will return an `AssertionError`. The error is deliberately placed to make sure the user input to the function will be proper, and the code will run as expected.

The last error for a type `str` of length eight which has non-integer characters is not covered with the `assert` and the error will come when the function will try to convert the character into `int` type: `val = int(id_num[i])`. This is because converting non-digit characters to `int` type is invalid.

Solution 2.b

Now, removing the 'safety nets', we will run into other errors since our function is not supposed to handle the requested inputs (for example: it doesn't make sense to check the last digit of an Israeli ID when the input is the name of your favorite book). For example:

When passing the function, a list containing a number:

```
>>> control_digit([21388668])
IndexError: list index out of range
```

This is because unlike the valid passed string which has `len(ID)=8` and 8 indexes, my passed list has one element (the integer 21388668) and so when the for loop arrives at `i=2`, the list does not have another item and the requested index in the list is out of range.

Similarly, for other input of an `str` type of length shorter than eight, the code will fail when trying to index the string for an index which is out of range for the string.

Since the third case of an `str` type input of length eight with no digits did not have an `assert` in the first place, the error will not change.

Solution 2.c

iteration	i	id_num[i]	val	total
1	0	8	8	8
2	1	7	7	13
3	2	6	6	19
4	3	5	5	20
5	4	4	4	24
6	5	3	3	30
7	6	2	2	32
8	7	1	1	34

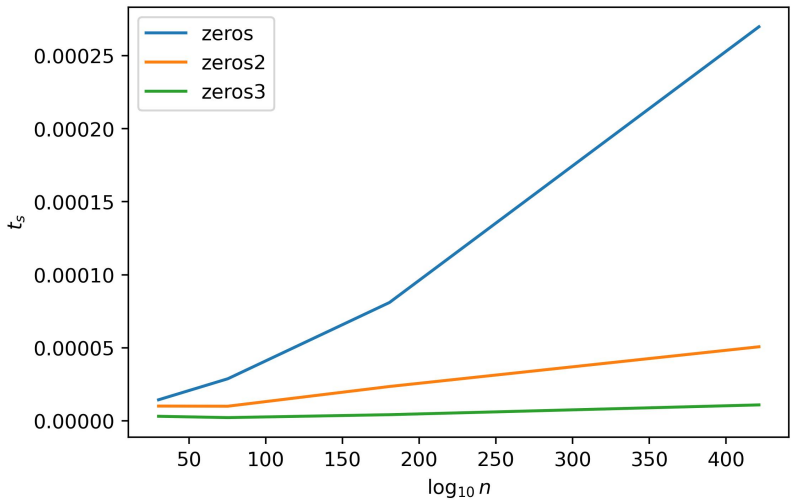
iteration	i	id_num[i]	val	total
1	0	2	2	2
2	1	1	1	4
3	2	3	3	7
4	3	8	8	14
5	4	8	8	22
6	5	6	6	25
7	6	6	6	31
8	7	8	8	38

Problem 3

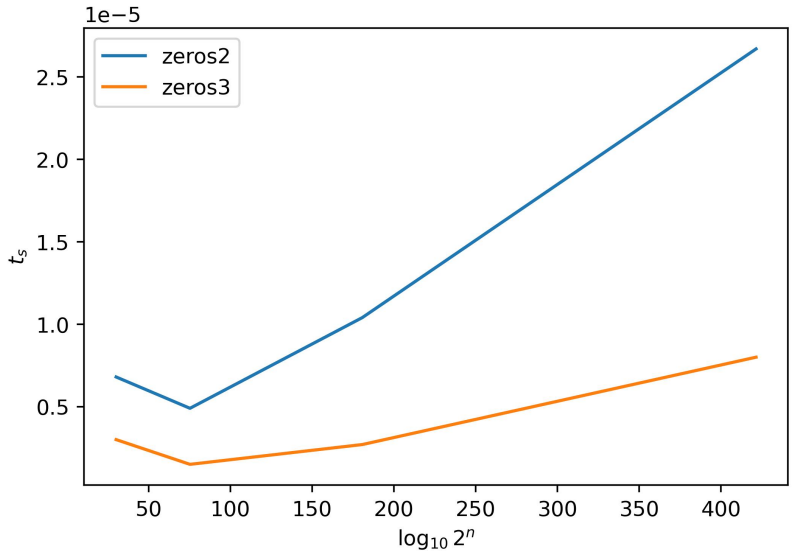
Solution 3.a.b

Using the following code which utilizes the `matplotlib` library, I get the following plot:

Since using the numbers directly resulted in an `OverflowError`, I resorted to using the base 10 log of the number. From the graph, we can clearly see that the `str.count` based function is the most efficient. The plot for the original `zeros` function has a steep relationship with $\log_{10} n$ which means that it raises exponentially for n.



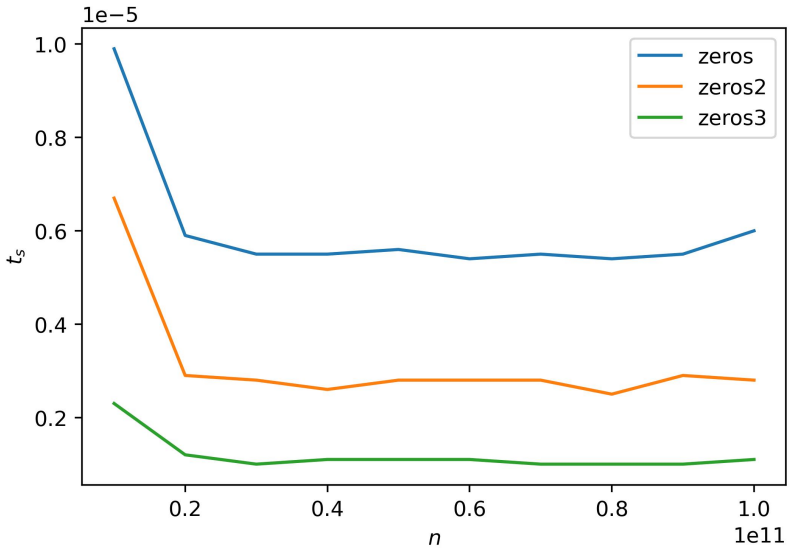
If, for a moment, I remove `zeros` function so we can focus on the other plots:



Beyond $\sim \log_{10} 2^{75}$, both functions start to raise exponentially, but still the rate of change for the Python-based function is smaller. When comparing the two functions to the `zeros` function there is pretty much no comparison, they are already on different orders of magnitude.

Solution 3.c

Running my small plotting code on the numbers $\{1e10, 2e10, \ldots, 10e10\}$, the plot suggests that the differences for numbers of the same order of magnitude is less drastic, but nevertheless noticeable:



Solution 3.d

The number is so large that it is likely that it will pretty much take forever (literally) to run. While the loop in the second example runs on the numbers of the large number and has 301 iterations accordingly, the example requires 301^{10} iterations which is ridiculously larger to the point of impossible.

Problem 5

Solution 5.b

The following false function is suggested:

```
def is_anagram_v2(st1, st2):
    for ch in st1:
        if st1.count(ch) != st2.count(ch):
            return False
    return True
```

This function would fail because it will not account for characters of `st2` which are not present in `st1`. For example:

```
>>>is_anagram_v2("silent", "silenta")
True
```

The fix is to check for both strings:

```
def is_anagram_v2(st1, st2):
    for ch in st1+st2:
        if st1.count(ch) != st2.count(ch):
```

```
            return False
    return True
```

Solution 5.c

The lists resulting from `sorted()` will order all the characters according to alphabetical order. The lists are equal if they contain the same elements in the same order.

```
def is_anagram_v3(st1, st2):
        return sorted(st1)==sorted(st2)
```

There is no direct connection between the length of the code and the compiled time, but code which is shorter might be easier on the eye, although sometimes harder to digest. It is up to the developer to decide which phrasing is better.

I believe in this case the using the built it `sorted` function would be faster since built-in functions are usually more efficient, and it is very likely that the experienced developers had accounted for things I have not thought of.