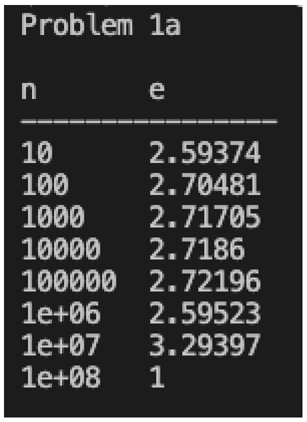
Charis Liao
CHEM 281
Luis Crivelli, Rebecca Vanessa Tomann

HOMEWORK II

**Problem 1.**

1a.

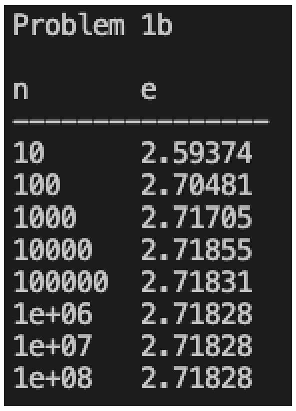


```
Problem 1a

n       e
----------------
10      2.59374
100     2.70481
1000    2.71705
10000   2.7186
100000  2.72196
1e+06   2.59523
1e+07   3.29397
1e+08   1
```

From the table above, we can clearly see that the number for 1e+08 is quite off compared to previous numbers. In the code, single precision was used in the function to calculate Euler's number (e). Single precision represents real numbers with a limited number of bits (In our case, it was 32 bits in C++). As n gets larger in calculating the Euler's number, the terms become extremely small which can lead to loss of precision in single precision calculations due to limited precision, loss of significant digits, and summation errors.

1b.



```
Problem 1b

n       e
----------------
10      2.59374
100     2.70481
1000    2.71705
10000   2.71855
100000  2.71831
1e+06   2.71828
1e+07   2.71828
1e+08   2.71828
```

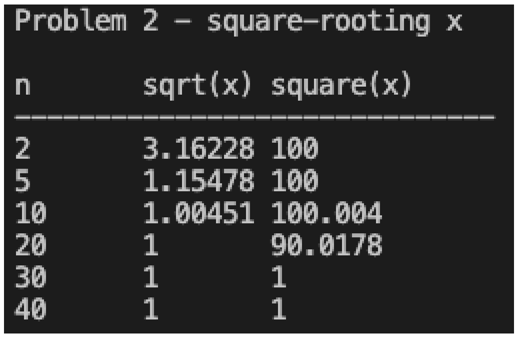
In 1b, we've improved the estimation by using log arithmetics and limits.

$log(1 + 1/n)^n = nlog(1 + 1/n)$

$$\lim_{n \to \infty} (1 + 1/n) = \lim_{x \to 0} log(1 + x)$$

In this case, x = 1/n. As n gets larger, x becomes extremely small to a point where it will be super close to zero. We can use Taylor expansion to solve this approximation, an as x gets closer to 0, the higher-order terms become smaller and smaller which become negligible and we can approximate $log(1 + x) \approx x$. In our code, we set up a threshold for when x gets smaller; otherwise, we can calculate as is. This solves the precision problem when x gets extremely small which is the reason why the numbers didn't break as shown.

**Problem 2.**

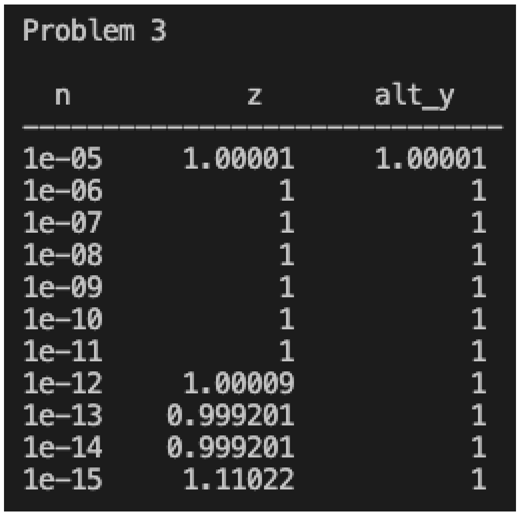


```
Problem 2 - square-rooting x

n       sqrt(x) square(x)
----------------------------
2       3.16228 100
5       1.15478 100
10      1.00451 100.004
20      1       90.0178
30      1       1
40      1       1
```

In problem 2, we are performing square-root and square on number $x$, n times. For instance, if n is 2, then we are square-rooting x twice then squaring it twice. From the result above, we can clearly see that as n gets larger, the performance of square-root and square becomes very off. This is due to limited precision as we are still performing the calculations under single precision. Another reason is due to the square root and squaring operations themselves. The square root operation and squaring operations are not exact inverses of each other when using floating point arithmetic. As we repeatedly square root and square the number, we introduce more rounding errors, which accumulate and lead to inaccuracies.

**Problem 3.**

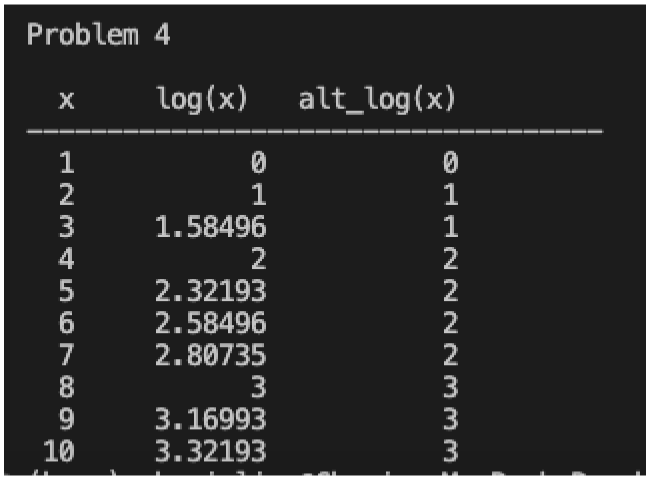


```
Problem 3

  n            z        alt_y
----------------------------
1e-05     1.00001     1.00001
1e-06           1           1
1e-07           1           1
1e-08           1           1
1e-09           1           1
1e-10           1           1
1e-11           1           1
1e-12     1.00009           1
1e-13    0.999201           1
1e-14    0.999201           1
1e-15     1.11022           1
```

In problem 3, we used double precision instead of single precision; however, we can still see some differences when n gets extremely small (starting from 1e-12). This might be due to small discrepancies of rounding errors and the inherent limitations of representing real numbers in a finite number of bits. Even though double precision provides more accurate results, there are still some limitations of floating point arithmetic.

**Problem 4.**



```
Problem 4

  x     log(x)   alt_log(x)
-----------------------------------
  1          0          0
  2          1          1
  3    1.58496          1
  4          2          2
  5    2.32193          2
  6    2.58496          2
  7    2.80735          2
  8          3          3
  9    3.16993          3
 10    3.32193          3
```

In problem 4, we are trying to calculate the exponent from the log function. Then, we will use an alternative way, bitwise manipulation, to calculate the exponent. Four steps were performed when calculating the exponent:

1. We have to convert the integer into bits for bitwise calculation.
2. Get rid of the mantissa because we only care about the integer bit
3. Mask the sign bit
4. Subtract the bias (in this case 1023)

From the table, we can see that for the alt_log(x) part, we get integers, but for the log(x) we get decimals when the number isn't multiples of 2. This is due to the fact that we have not taken mantissa into account. Note, the log(x) in this case is log base 2 of 2 instead of log base 10 of x. Since in the bitwise calculation, we're dealing with log base two, it is only reasonable for the log(x) calculation to be base 2 as well.