1 Problem 1

1.1 Part A

I'll use the `time` library, which measures time since the epoch. Calling the time before the function and after the function, and then subtracting the two values will give the time taken to run the function.

1.1.1 Example Time Calculation

In [1]:
```
from time import *
import numpy as np

a = time()

x = np.linspace(0,100,101);

b = time()

difference = b - a

print(difference)
```

```
0.0003631114959716797
```

1.2 Parts B & C

I wrote a script to solve this both ways. This uses matrices that are 100 x 100 and solves the matrix equation 1000 times, then averages.

In [2]:
```
run Problem_1_b_c.py
```

```
Inverse Method:  0.0008504364490509033
Numpy Linear Algebra Solver Method:  0.0002674200534820557
```

The ratio between the two is given below:

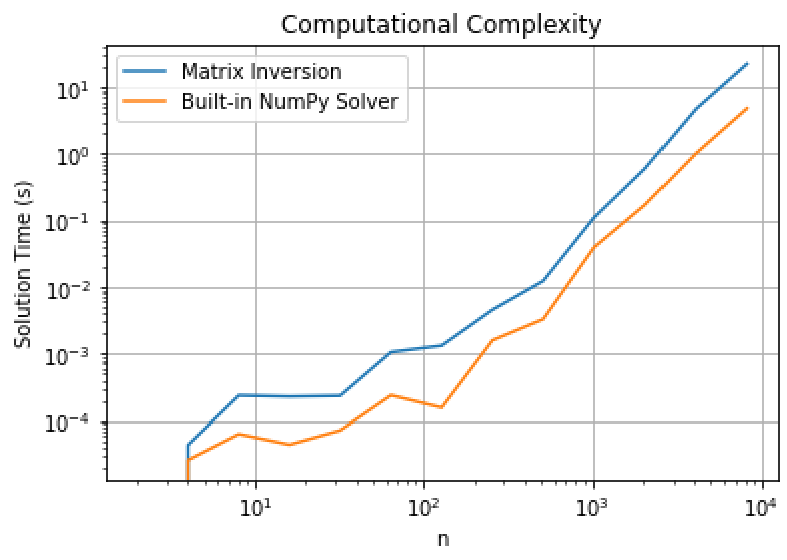
In [3]:
```
ratio = tavg_inv / tavg_imp
print(ratio)
```

```
3.1801521164080127
```

1.3 Part D

In [4]:
```
run Problem_1_d.py
```

```
n =  4
Inverse Method:  4.351139068603515 6e-05
Numpy Linear Algebra Solver Method:  2.5987625122070312e-05
n =  8
Inverse Method:  0.00023865699768066406
Numpy Linear Algebra Solver Method:  6.341934204101562e-05
n =  16
Inverse Method:  0.00022995471954345703
Numpy Linear Algebra Solver Method:  4.398822784423828e-05
n =  32
Inverse Method:  0.00023603439331054688
Numpy Linear Algebra Solver Method:  7.200241088867188e-05
n =  64
Inverse Method:  0.0010600090026855469
Numpy Linear Algebra Solver Method:  0.00023996829986572266
n =  128
Inverse Method:  0.001326918601989746
Numpy Linear Algebra Solver Method:  0.00015664100646972656
n =  256
Inverse Method:  0.004567027091979980 5
Numpy Linear Algebra Solver Method:  0.001592874526977539
n =  512
Inverse Method:  0.012356996536254883
Numpy Linear Algebra Solver Method:  0.0033004283905029297
n =  1024
Inverse Method:  0.10994851589202881
Numpy Linear Algebra Solver Method:  0.03925442695617676
n =  2048
Inverse Method:  0.5965710878372192
Numpy Linear Algebra Solver Method:  0.17074990272521973
n =  4096
Inverse Method:  4.746130347251892
Numpy Linear Algebra Solver Method:  1.022537112236023
n =  8192
Inverse Method:  22.426724076271057
Numpy Linear Algebra Solver Method:  4.802940011024475
```



```
Finite-differencing the last four points in each (in logarithmic space):
Inverse Method Exponent:  2.5574156524146088
Numpy Method Exponent:  2.3116395382031034
```

It looks like they have approximately the same asymptotic computational complexity.

2 Problem 2

2.1 Part A

Question: *Given two proposed factors of an N digit integer, argue that the number of computer operations needed to verify whether their product is correct is less than a constant times N^2:*

Answer: This problem is basically just asking whether scalar multiplication can be done in less than N^2 operations, where N is the number of digits in the final answer. We can do long multiplication just like we learned in grade school, but instead of waiting until the end we can constantly add the solutions from each step together. This reduces the number of step required to solve the problem. I don't really understand how to calculate the actual value of the exponent or whether it's logarithmic.

2.2 Part B

Show that $\neg(X \land Y) = (\neg X) \lor (\neg Y)$:

Let's prove this by brute force:

Suppose $X = true, Y = true$:

- $\neg(T \land T) = F$
- $(\neg T) \lor (\neg T) = F \lor F = F$

Suppose $X = true, Y = false$:

- $\neg(T \land F) = T$
- $(\neg T) \lor (\neg F) = F \lor T = T$

Suppose $X = false, Y = true$

- $\neg(F \land T) = T$
- $(\neg F) \lor (\neg T) = T \lor F = T$

Suppose $X = false, Y = false$

- $\neg(F \land F) = T$
- $(\neg F) \lor (\neg F) = T \lor T = T$

The four potential combinations evaluate the same way on both, showing that the conjuncture $\neg(X \wedge Y) = (\neg X) \vee (\neg Y)$ is likely true.

Now we will try to write the 3-color plots in conjunctive normal form: **Attempted, but not finished**

Plot 1: Only three different colors are needed

- $(A \vee C) \wedge (B \vee A \vee C) \wedge (D \vee A \vee C)$
- $(X_1 \vee X_2 \vee X_3) \wedge ()$

Plot 2: All must have different colors to satisfy this clause

- $(A \vee B \vee C \vee D)$

Well, I made the mistake of thinking that I would be able to finish this problem if I started on Thursday. It turns out I was wrong, and I won't be finishing this problem on time. Therefore, I will now simply explain the approach that I would take in words to each of the parts to this problem, and if that's worth some credit, great. If not, that's fine too since this is still a useful exercise.

2.3 Part C

The DP algorithm is a simple algorithm for determining whether a series of boolean statements has an existing combination of states that result in an ultimately satisfied state. Here's the logic of the algorithm:

1. Set one of the variables to true (all others being false)
2. Reduce the clauses that contain that variable
3. If they all return true, then choose another variable to set to true
4. Reduce the clauses containing that newly-set variable
5. If they also all return true, choose yet another variable to set to true
6. If at any point any of the clauses containing your newly-set variable return false, return to your previous variable, and try switching it from true to false, then continue forward (the next variable should still be set to true from before)
7. As needed, back up through the variables as many times as you need to until you can go through all of the variables and return a true for the entire tree.

2.4 Part D

I would set up the **kSAT** problem by creating a random matrix of integers that is $M \times N$, and changing values if they put two variables in the same row. The integers would be chosen between $\pm N$, with positive values corresponding to true statements and negative values corresponding to NOT statements.

For small values of M/N, we would expect most of the problems to be solvable because the problem is unconstrained, meaning that there are likely multiple combinations that will produce an overall true result. These are also likely to run fast because the likelihood of finding just one of the paths quickly is fairly high.

For large values of M/N, we would expect most of the problems to be unsolvable because the problem is overconstrained, meaning that there are so many "rules" that they likely contradict each other. The system may still be solvable if the rules happen to be noncontradictory, but the likelihood of this is smaller and smaller the larger that M/N gets. For large M/N values the algorithm is likely to run quickly because each branch quickly gets ruled out as none of them are satisfiable.

For values of M/N near some critical point, the solvability of the system starts to transition from likely solvable to likely not solvable. It is in these situations where the DP algorithm must explore almost every branch of the tree, and hence the run times are likely to be largest when M/N is in this region.

2.5 Part E

I would expect that checking for singletons would make the DP algorithm run faster because singletons help you immediately determine whether a variable must be set to true or false, which negates your need to "flip that switch". In fact, I would expect it to be sped up a lot by checking for singletons because as you progress through the tree you will find more and more singletons as more variabelse are set.

2.6 Part F

The exponent should be negative, because the longer the time runs, the less likely the system is to ultimately be solved. I'm not sure what it should be, but I am pretty confident that is should be negative.

I believe that choosing the most common variable first would indeed speed up your solution time because you don't have to spend a long time deciding what the state of the most commonly-used variable is.

Conversely, I also expect starting with the least commonly-used variable would dramatically slow down the algorithm. You can likely think of this as trying to build the roof before the foundation, or adding detail to something whose basic shape is not yet complete.

2.7 Part G

The best algorithm would probably have been the algorithm that started with the most commonly-used variable. Obviously, I don't have results for this, but I would hope for my algorithm to converge to the values specified for **2SAT** and **3SAT**. I would also anticipate that **2SAT** is in **P** and that **3SAT** grows exponentially, since that's what the assignment asks about.

So that's the homework! I definitely should have started sooner. I also think that in the future it would help to have this problem assigned after talking about the DP algorithm and noting that we will need to spend several days doing this problem. It's my fault that I started too late, but hopefully my conceptual understanding helps salvage a part of the points. If not, that's totally fine too. Thanks!