# ISS Assignment 2

By Aniruddha Bala
Roll No: 06-02-01-10-51-18-1-15655

October 7, 2018

## 1 Introduction to Scalable Systems Assignment - 2

### 1.1 Objective

1. To implement a nxm matrix in array and csr format
2. Perform various operations: load, add, multiply, bfs traversal
3. Analyze the empirical and asymptotic space and time complexity

For 1 and 2 please refer files MatrixImpl.cpp and Runner.cpp. In this report we shall see the results of the various operations performed on both array and csr implementations of the matrix and analyze their space and time complexity. The results below have been obtained for nxn square matrices.

### 1.2 System Configuration

All the results presented here were obtained after running the experiments on node1 of the Turing cluster. It has the following specifications:

```
aniruddhab@node1:~/aniruddhab$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             AuthenticAMD
CPU family:            21
Model:                 2
Model name:            AMD Opteron(tm) Processor 3380
Stepping:              0
CPU MHz:               1400.000
CPU max MHz:           2600.0000
CPU min MHz:           1400.0000
BogoMIPS:              5199.60
Virtualization:        AMD-V
L1d cache:             16K
L1i cache:             64K
L2 cache:              2048K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca c
tsc extd_apicid aperfmperf pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2 pop
t lwp fma4 tce nodeid_msr tbm topoext perfctr_core perfctr_nb cpb hw_pstate vmmca
aniruddhab@node1:~/aniruddhab$ dmidecode
# dmidecode 3.0
/sys/firmware/dmi/tables/smbios_entry_point: Permission denied
Scanning /dev/mem for entry point.
/dev/mem: Permission denied
aniruddhab@node1:~/aniruddhab$ free -g
              total        used        free      shared  buff/cache   available
Mem:             31           1          19           0          10          29
Swap:            18           1          16
aniruddhab@node1:~/aniruddhab$ free -g -h
              total        used        free      shared  buff/cache   available
Mem:            31G        1.6G         19G         16M         10G         29G
Swap:           18G        1.9G         16G
```

System Specifications

## 1.3 Loading a matrix

### 1.3.1 Results for dense data

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
```

**Array Implementation** The analysis is done for n varying from 32 to 655536 in powers of 2, the outputs of time taken to run and the memory consumed were saved to two csv files.

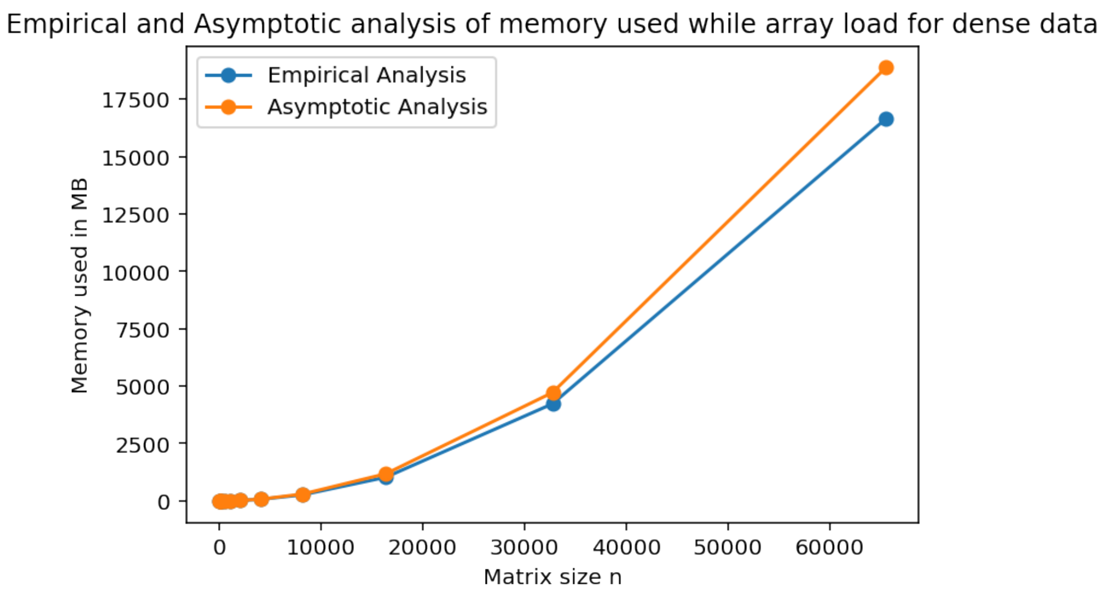
```
In [2]: %matplotlib inline
        %config InlineBackend.figure_format = 'retina'
        load_array_mem = pd.read_csv('load_array_mem.csv')
        print(load_array_mem)
        load_array_mem = load_array_mem.values
        # Empirical analysis
```

```
x = load_array_mem[:,0]
y = load_array_mem[:,1]/1024 #divided by 1024 to convert to MB
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c=0.0045
plt.plot(x, (c*x*x)/1024, label='Asymptotic Analysis', marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Memory used in MB')
plt.title('Empirical and Asymptotic analysis of memory used while array load for dense d
plt.legend()
```

| | Matrix size n | Memory used in KB |
|---|---|---|
| 0 | 32 | 1612 |
| 1 | 64 | 1612 |
| 2 | 128 | 1612 |
| 3 | 256 | 3260 |
| 4 | 512 | 4204 |
| 5 | 1024 | 7220 |
| 6 | 2048 | 19628 |
| 7 | 4096 | 68888 |
| 8 | 8192 | 265636 |
| 9 | 16384 | 1052324 |
| 10 | 32768 | 4329212 |
| 11 | 65536 | 17042448 |

Out[2]: <matplotlib.legend.Legend at 0x7f69f8532128>

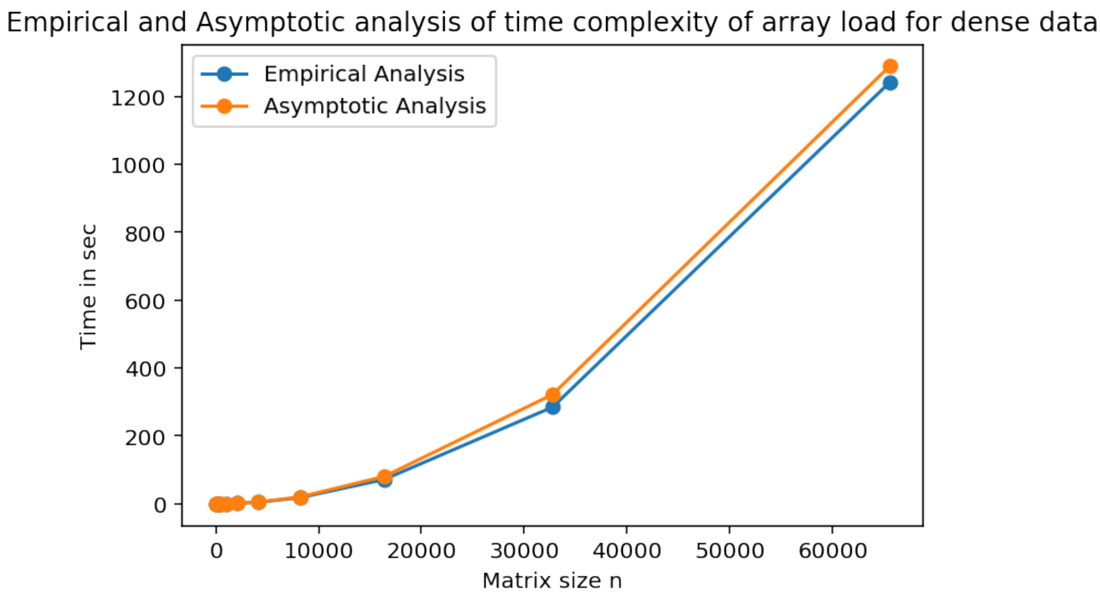


```
In [3]: load_array_time = pd.read_csv('load_array_time.csv')
        print(load_array_time)
        load_array_time = load_array_time.values
        # Empirical analysis
        x = load_array_time[:,0]
        y = load_array_time[:,1]/1000 #divided by 1000 to convert to secs
        plt.plot(x,y,label='Empirical Analysis',marker='o')

        # Asymptotic analysis
        c = 0.0003;
        plt.plot(x, (c*x*x)/1000, label='Asymptotic Analysis',marker='o')
        plt.xlabel('Matrix size n')
        plt.ylabel('Time in sec')
        plt.title('Empirical and Asymptotic analysis of time complexity of array load for dense
        plt.legend()
```

```
    Matrix size n    Time in ms
0              32  1.024568e+00
1              64  2.550357e+00
2             128  8.286016e+00
3             256  2.039012e+01
4             512  7.534698e+01
5            1024  2.908905e+02
6            2048  1.141751e+03
7            4096  4.516397e+03
8            8192  1.813130e+04
9           16384  7.165687e+04
10          32768  2.849183e+05
11          65536  1.240826e+06
```

```
Out[3]: <matplotlib.legend.Legend at 0x7f69f83fc550>
```




**Observation for array implementation**

1. Memory vs Input size (n): To plot the asymptotic curve I have chosen c so that $c * g(n)$ is the upper bound for the curve $f(n)$. Therefore we can see that $f(n)$ (empirical curve) satisfies $f(n) <= 0.0045n^2$ for $n >= n_0$ for some $n_0$ between 0 to 5000. Therefore we can say that load requires $O(n^2)$ memory. Also theoretically we expect $n^2$ elements to take $O(n^2)$ space.

2. Time vs Input size (n) : Similarly to plot the asymptotic curve for time complexity I have chosen $c = 0.0003$ such that $c * g(n)$ is a tighter bound for $f(n)$. From the plot we can see that $f(n) <= 0.0003n^2$. Therefore we can say that load requires $O(n^2)$ time. Also by looking at the code we can analyze the time complexity, as we have 2 nested loops one running for the number of rows in a matrix and other is reading each line character wise, if we assume that we use fixed number of digits to represent each float say $k$ then the inner loop runs for $k * m$ where $m$ is the number of floats in that row if we have $n = m$ then we can say the time complexity is $k * n^2$

3. The largest matrix size that I could load within reasonable time was 65536x65536 which took around 1240.825 secs (approx 20 mins) to load. It is expected to load because theoretically the matrix should occupy

$$\frac{65536 * 65536 * 4}{2^{30}} = 16GB$$

which can fit in the main memory as main memory has 31 GB capacity. This can be also be verified from the experimental result obtained.

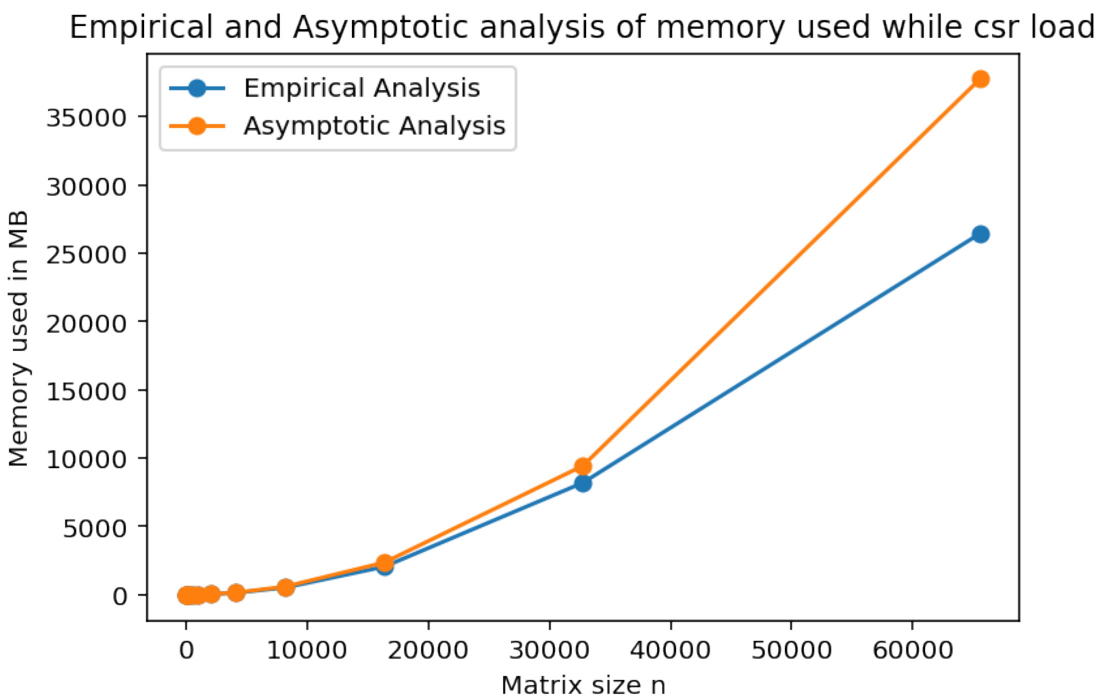
**CSR Implementation** The analysis is done for n varying from 32 to 655536 in powers of 2, the outputs of time taken to run and the memory consumed were saved to two csv files.

```
In [4]: load_csr_mem = pd.read_csv('load_csr_mem.csv')
        print(load_csr_mem)
        load_csr_mem = load_csr_mem.values
        # Empirical analysis
        x = load_csr_mem[:,0]
        y = load_csr_mem[:,1]/1024
        plt.plot(x,y,label='Empirical Analysis',marker='o')

        # Asymptotic analysis
        c=0.009
        plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
        plt.xlabel('Matrix size n')
        plt.ylabel('Memory used in MB')
        plt.title('Empirical and Asymptotic analysis of memory used while csr load')
        plt.legend()
```

```
    Matrix size n  Memory used in KB
0              32               1612
1              64               1612
2             128               3260
3             256               3548
4             512               5200
5            1024              11460
6            2048              36156
7            4096             134516
8            8192             527820
9           16384            2100576
10          32768            8392316
11          65536           27071568
```

```
Out[4]: <matplotlib.legend.Legend at 0x7f69f83ed8d0>
```



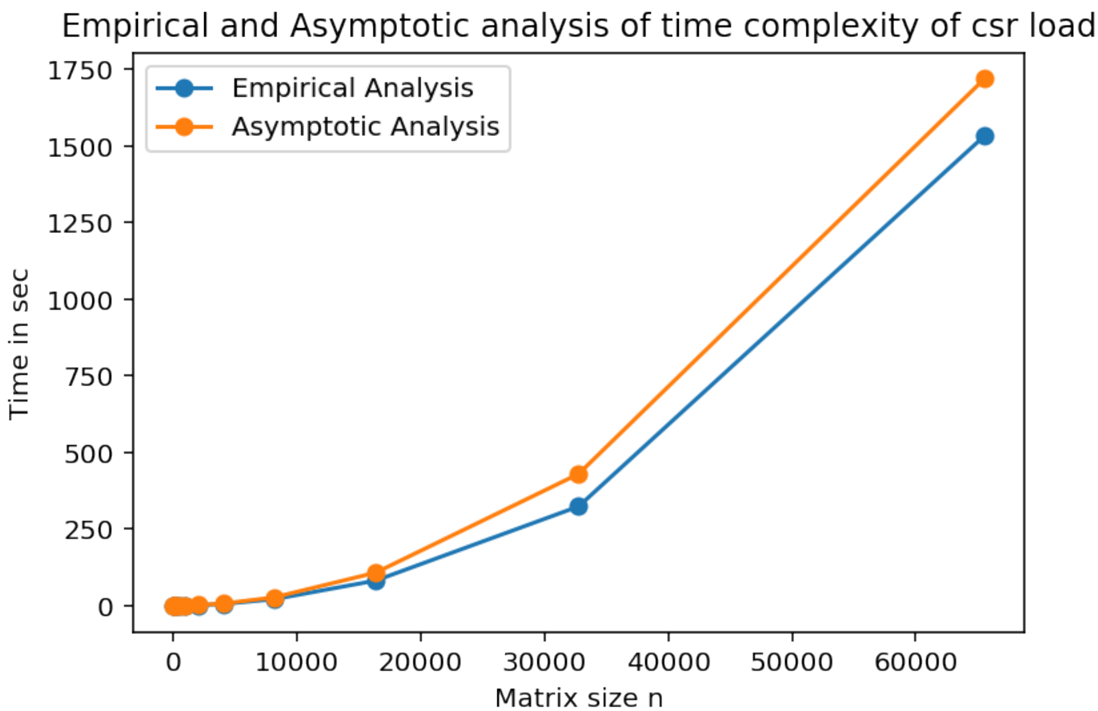

```
In [5]: load_csr_time = pd.read_csv('load_csr_time.csv')
        print(load_csr_time)
        load_csr_time = load_csr_time.values
        # Empirical analysis
        x = load_csr_time[:,0]
        y = load_csr_time[:,1]/1000
        plt.plot(x,y,label='Empirical Analysis',marker='o')

        # Asymptotic analysis
        c = 0.0004
        plt.plot(x, (c*x*x)/1000, label='Asymptotic Analysis',marker='o')
        plt.xlabel('Matrix size n')
        plt.ylabel('Time in sec')
        plt.title('Empirical and Asymptotic analysis of time complexity of csr load')
        plt.legend()
```

```
   Matrix size n    Time in ms
0             32  9.504460e-01
1             64  3.385009e+00
2            128  1.263003e+01
3            256  2.551260e+01
4            512  8.877443e+01
5           1024  4.012100e+02
6           2048  1.278410e+03
```

```
7            4096  5.120258e+03
8            8192  2.073626e+04
9           16384  8.145852e+04
10          32768  3.234241e+05
11          65536  1.531762e+06
```

```
Out[5]: <matplotlib.legend.Legend at 0x7f69f8371eb8>
```




**Observation for csr implementation**

1. Memory vs Input size (n): To plot the asymptotic curve I have chosen c so that $c * g(n)$ is the upper bound for the curve $f(n)$. Therefore we can see that $f(n)$ (empirical curve) satisfies $f(n) <= 0.009n^2$ for $n >= n_0$ for some $n_0$ between 0 to 5000. Therefore we can say that load requires $O(n^2)$ memory in case of dense matrix. As the matrix is dense and has rare zero elements therefore theoretically also we expect $n^2$ elements to take $O(n^2)$ space.

2. Time vs Input size (n) : Similarly to plot the asymptotic curve for time complexity I have chosen $c = 0.0004$ such that $c * g(n)$ is an upper bound for $f(n)$. From the plot we can see that $f(n) <= 0.0004n^2$. Therefore we can say that load requires $O(n^2)$ time. Also by looking at the code we can analyze the time complexity, as we have 2 nested loops one running for the number of rows in a matrix and other is reading each line character wise, if we assume that we use fixed number of digits to represent each float say $k$ then the inner loop runs for $k * m$ where $m$ is the number of floats in that row if we have $n = m$ then we can say the time

complexity is $k * n^2$ as other operations inside loop like adding an element to vector take constant time.

3. The largest matrix size that I could load within reasonable time was 65536x65536 which took around 1531.762 secs (approx 25 mins) to load. It is expected to load because theoretically the matrix should occupy
$$\frac{65536 * 65536 * 4}{2^{30}} = 16GB$$
which can fit in the main memory as main memory has 31 GB capacity. This can be also be verified from the experimental result obtained.

**Conclusion**

1. For dense data both array and csr implementations have worst case space complexity of $O(n^2)$ and worst case time complexity of $O(n^2)$ for load.
2. CSR implementation requires about 2 times more memory this is because we maintain two additional arrays to store the column indices and the cumulative count. However we should expect to see some improvement in sparse matrix case.
3. The time taken for load in both cases is almost similar.

### 1.3.2 Results for sparse data

The sparse data was generated with sparsity factor of 0.8 i.e roughly 80% of the elements the array are 0.

**Array Implementation** The analysis is done for n varying from 32 to 32768 in powers of 2, the outputs of time taken to run and the memory consumed were saved to two csv files.

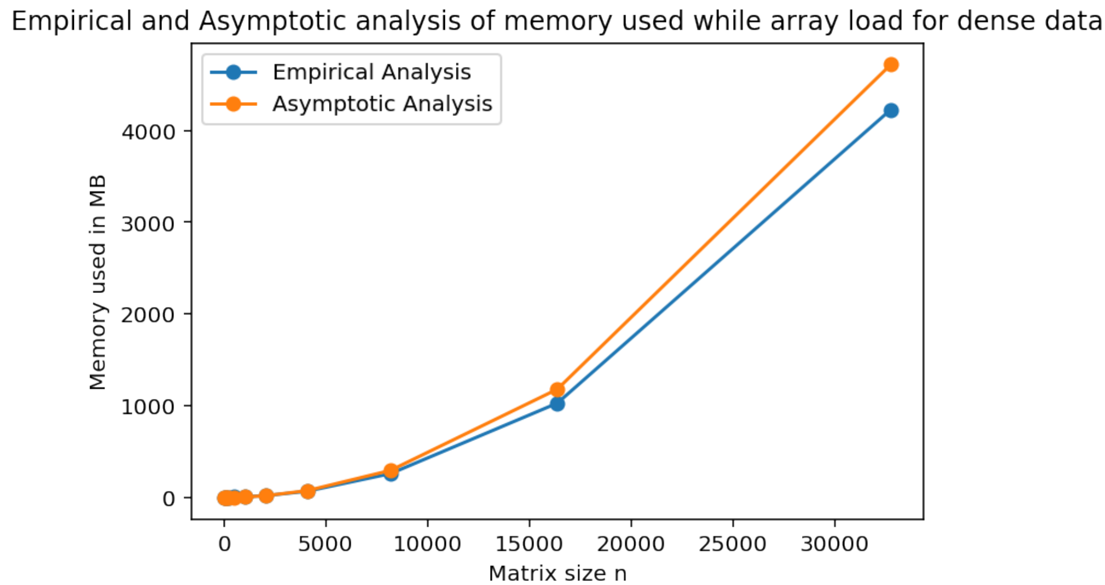
```
In [6]: load_array_mem = pd.read_csv('load_sparse_array_mem.csv')
        print(load_array_mem)
        load_array_mem = load_array_mem.values
        # Empirical analysis
        x = load_array_mem[:,0]
        y = load_array_mem[:,1]/1024
        plt.plot(x,y,label='Empirical Analysis',marker='o')

        # Asymptotic analysis
        c=0.0045
        plt.plot(x, (c*x*x)/1024, label='Asymptotic Analysis', marker='o')
        plt.xlabel('Matrix size n')
        plt.ylabel('Memory used in MB')
        plt.title('Empirical and Asymptotic analysis of memory used while array load for dense d
        plt.legend()
```

```
   Matrix size n  Memory used in KB
0             32               1616
1             64               1616
```

| | | |
|---|---|---|
| 2 | 128 | 1616 |
| 3 | 256 | 3264 |
| 4 | 512 | 4200 |
| 5 | 1024 | 7224 |
| 6 | 2048 | 19368 |
| 7 | 4096 | 68880 |
| 8 | 8192 | 265632 |
| 9 | 16384 | 1052300 |
| 10 | 32768 | 4329132 |

```
Out[6]: <matplotlib.legend.Legend at 0x7f69f82ce9b0>
```

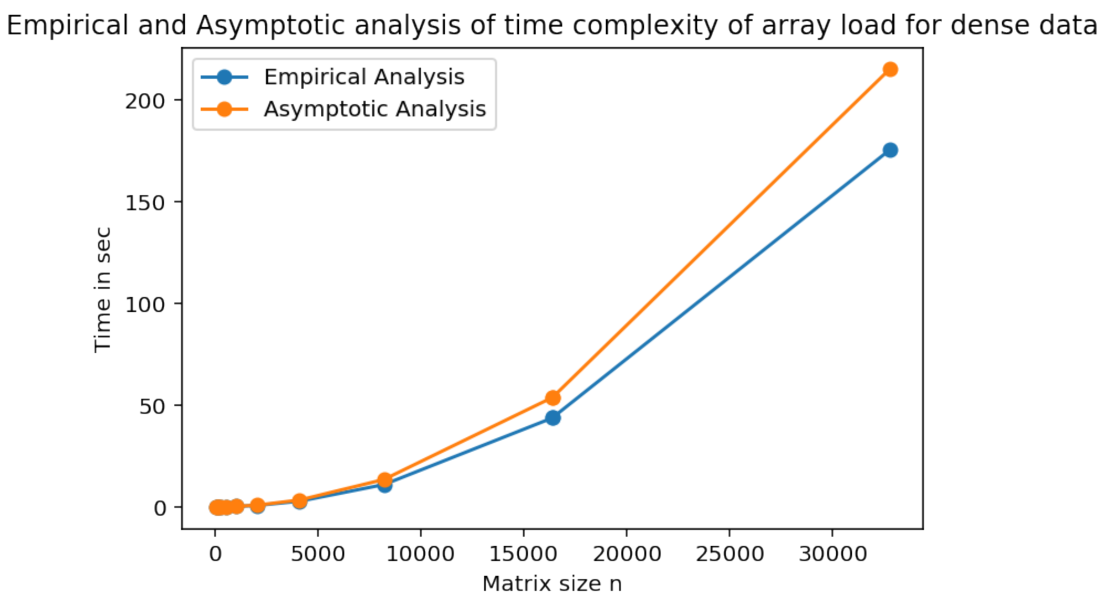



```
In [7]: load_array_time = pd.read_csv('load_sparse_array_time.csv')
        print(load_array_time)
        load_array_time = load_array_time.values
        # Empirical analysis
        x = load_array_time[:,0]
        y = load_array_time[:,1]/1000
        plt.plot(x,y,label='Empirical Analysis',marker='o')

        # Asymptotic analysis
        c = 0.0002;
        plt.plot(x, (c*x*x)/1000, label='Asymptotic Analysis',marker='o')
        plt.xlabel('Matrix size n')
        plt.ylabel('Time in sec')
        plt.title('Empirical and Asymptotic analysis of time complexity of array load for dense
        plt.legend()
```

| | Matrix size n | Time in ms |
|---|---|---|
| 0 | 32 | 0.9506 |
| 1 | 64 | 2.0612 |
| 2 | 128 | 5.0561 |
| 3 | 256 | 13.7372 |
| 4 | 512 | 48.1714 |
| 5 | 1024 | 184.2111 |
| 6 | 2048 | 696.8582 |
| 7 | 4096 | 2739.7646 |
| 8 | 8192 | 10899.7845 |
| 9 | 16384 | 43651.3917 |
| 10 | 16384 | 43724.0154 |
| 11 | 32768 | 175123.0631 |

```
Out[7]: <matplotlib.legend.Legend at 0x7f69f63a0f60>
```




**Observations for array implemetation**

1. Memory vs Input size (n): To plot the asymptotic curve I have chosen c so that $c * g(n)$ is the upper bound for the curve $f(n)$. Therefore we can see that $f(n)$ (empirical curve) satisfies $f(n) <= 0.0045n^2$ for $n >= n_0$ for some $n_0$ between 0 to 5000. Therefore we can say that load requires $O(n^2)$ memory. Also theoretically we expect $n^2$ elements to take $O(n^2)$ space.

2. Time vs Input size (n) : Similarly to plot the asymptotic curve for time complexity I have chosen $c = 0.0002$ such that $c * g(n)$ is a tighter bound for $f(n)$. From the plot we can see that $f(n) <= 0.0002n^2$. Therefore we can say that load requires $O(n^2)$ time. Also by looking at the code we can analyze the time complexity, as we have 2 nested loops one running for

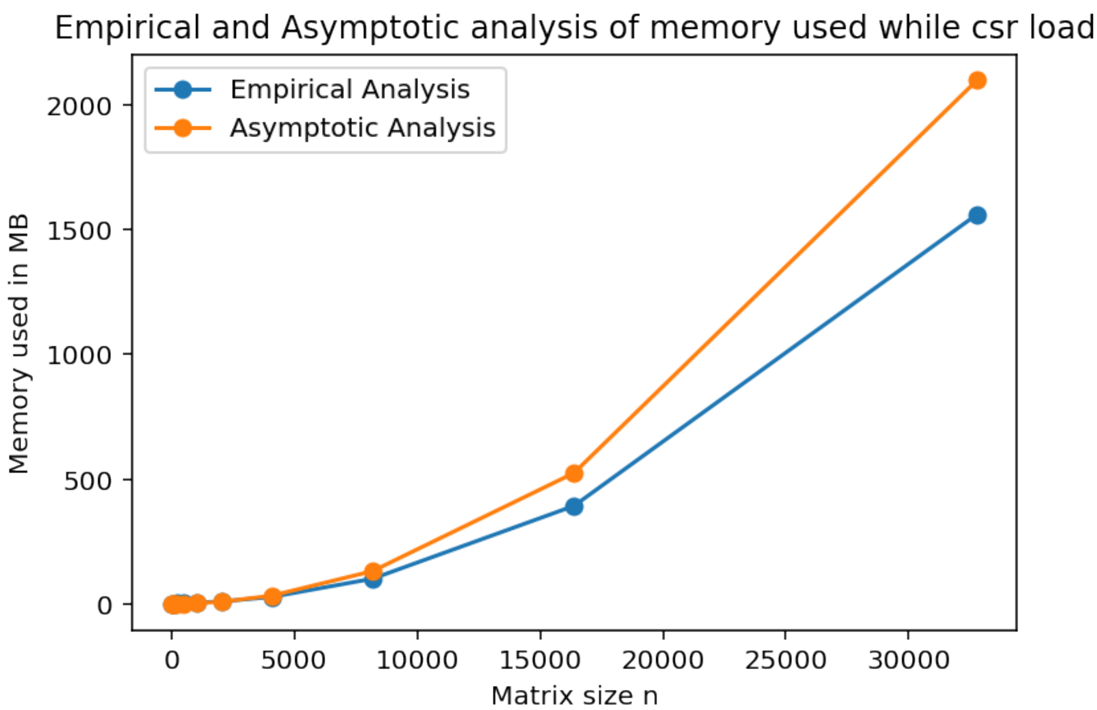
the number of rows in a matrix and other is reading each line character wise, if we assume that we use fixed number of digits to represent each float say $k$ then the inner loop runs for $k * m$ where $m$ is the number of floats in that row if we have $n = m$ then we can say the time complexity is $k * n^2$

In [8]:
```
load_csr_mem = pd.read_csv('load_sparse_csr_mem.csv')
print(load_csr_mem)
load_csr_mem = load_csr_mem.values
# Empirical analysis
x = load_csr_mem[:,0]
y = load_csr_mem[:,1]/1024
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c=0.002
plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Memory used in MB')
plt.title('Empirical and Asymptotic analysis of memory used while csr load')
plt.legend()
```

```
    Matrix size n  Memory used in KB
0              32               1616
1              64               1616
2             128               1616
3             256               3264
4             512               3752
5            1024               4936
6            2048               9628
7            4096              28200
8            8192             103048
9           16384             401992
10          32768            1597420
```

Out[8]:
```
<matplotlib.legend.Legend at 0x7f69f6374cc0>
```



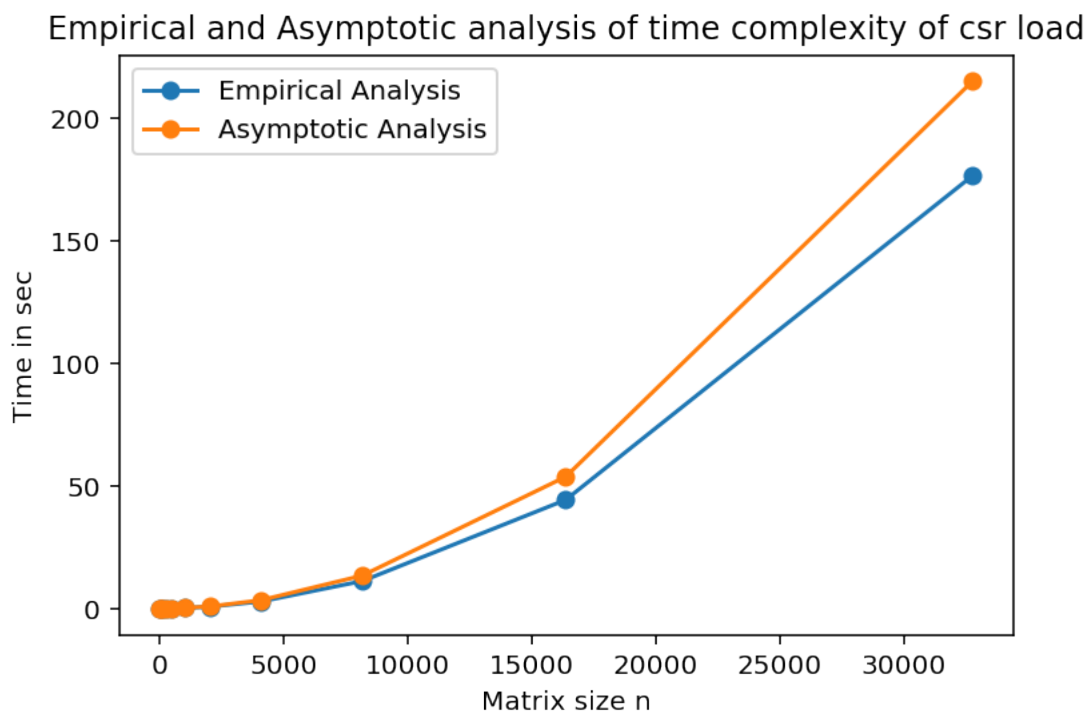

```
In [9]: load_csr_time = pd.read_csv('load_sparse_csr_time.csv')
        print(load_csr_time)
        load_csr_time = load_csr_time.values
        # Empirical analysis
        x = load_csr_time[:,0]
        y = load_csr_time[:,1]/1000
        plt.plot(x,y,label='Empirical Analysis',marker='o')

        # Asymptotic analysis
        c = 0.0002
        plt.plot(x, (c*x*x)/1000, label='Asymptotic Analysis',marker='o')
        plt.xlabel('Matrix size n')
        plt.ylabel('Time in sec')
        plt.title('Empirical and Asymptotic analysis of time complexity of csr load')
        plt.legend()
```

```
   Matrix size n  Time taken in ms
0             32            0.8408
1             64            2.1772
2            128            7.2989
3            256           15.6903
4            512           50.3193
5           1024          190.3335
6           2048          712.5982
```

```
7            4096        2802.5850
8            8192       11171.4577
9           16384       44266.2726
10          32768      176275.1295
```

```
Out[9]: <matplotlib.legend.Legend at 0x7f69f60ba5c0>
```




**Observations for csr implementation**

1. Memory vs Input size (n): To plot the asymptotic curve I have chosen c so that $c * g(n)$ is the upper bound for the curve $f(n)$. Therefore we can see that $f(n)$ (empirical curve) satisfies $f(n) <= 0.002n^2$ for $n >= n_0$ for some $n_0$ between 0 to 5000. Therefore we can say that load requires $O(n^2)$ memory in case of dense matrix. As the matrix is dense and has rare zero elements therefore theoretically also we expect $n^2$ elements to take $O(n^2)$ space.

2. Time vs Input size (n) : Similarly to plot the asymptotic curve for time complexity I have chosen $c = 0.0002$ such that $c * g(n)$ is an upper bound for $f(n)$. From the plot we can see that $f(n) <= 0.0002n^2$. Therefore we can say that load requires $O(n^2)$ time. Also by looking at the code we can analyze the time complexity, as we have 2 nested loops one running for the number of rows in a matrix and other is reading each line character wise, if we assume that we use fixed number of digits to represent each float say $k$ then the inner loop runs for $k * m$ where $m$ is the number of floats in that row if we have $n = m$ then we can say the time complexity is $k * n^2$ as other operations inside loop like adding an element to vector take constant time.

**Conclusion**

1. From above observations we see that with sparse data the space complexity in case of csr implementation reduces, the constant factor c remains the same in case of array implementation i.e.$c = 0.0045$ for both dense and sparse data, but in case of csr implementation the constant drops from 0.09 to 0.02 i.e

$$\frac{0.09 - 0.02}{0.09} * 100 = 77\%$$

lesser compared to before which is expected because the sparse array has 80% 0s. Therefore the best case space complexity will be $o((1 - sparsity) * n^2)$ , but the worst case space complexity is still $O(n^2)$

2. The time complexity remains almost same as before i.e. $O(n^2)$

## 1.4 Addition of two matrices

### 1.4.1 Results for dense data

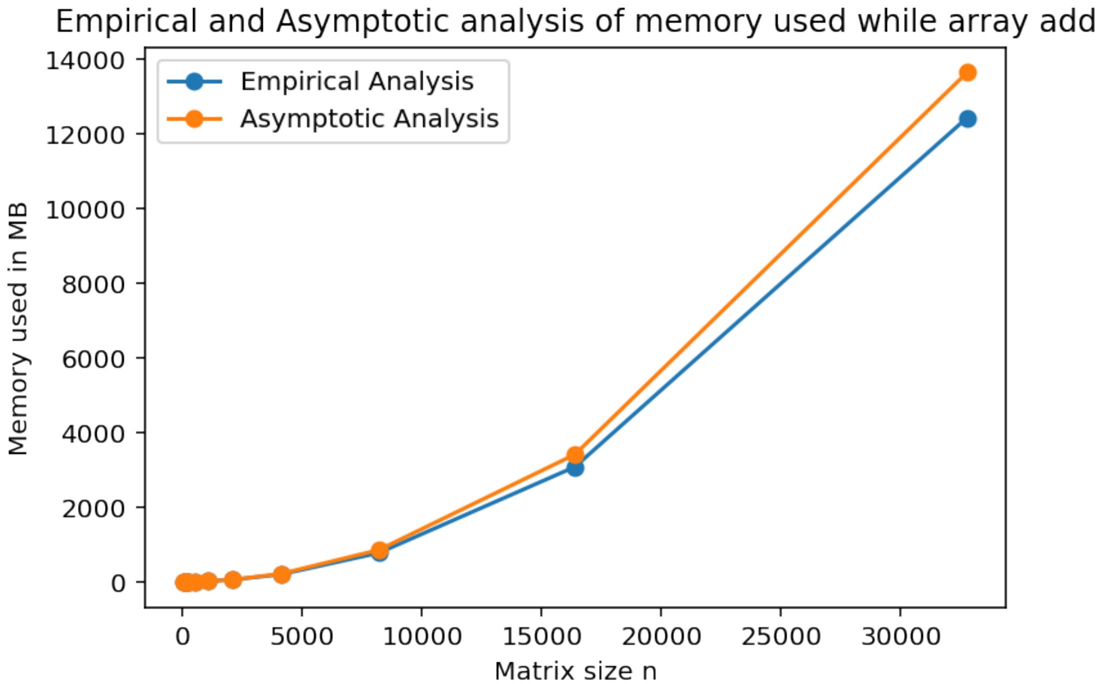
The analysis has been done for n varying from 32 to 32768 in powers of 2.

```
In [10]: add_array_mem = pd.read_csv('add_array_mem.csv')
         print(add_array_mem)
         add_array_mem = add_array_mem.values
         # Empirical analysis
         x = add_array_mem[:,0]
         y = add_array_mem[:,1]/1024
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.013
         plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
         plt.xlabel('Matrix size n')
         plt.ylabel('Memory used in MB')
         plt.title('Empirical and Asymptotic analysis of memory used while array add')
         plt.legend()
```

```
    Matrix size n  Memory used in KB
0              32               1612
1              64               1612
2             128               3260
3             256               3788
4             512               6028
5            1024              15404
6            2048              52364
7            4096             200004
8            8192             790152
9           16384            3150192
10          32768           12719196
```

Out[10]: <matplotlib.legend.Legend at 0x7f69f6034da0>



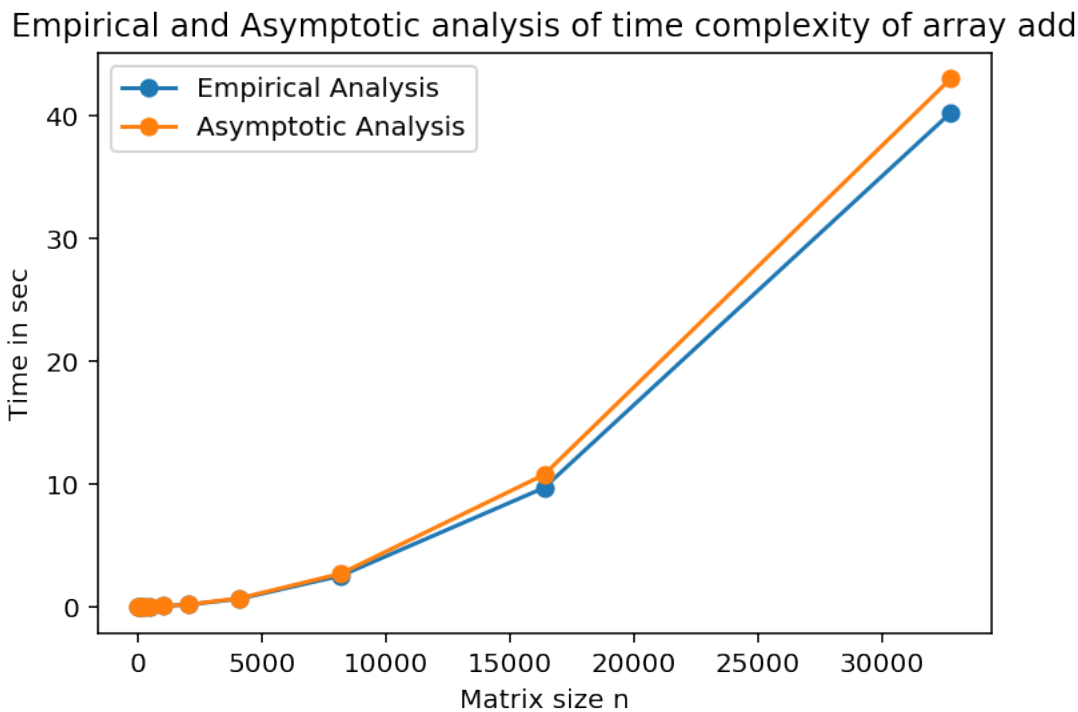

In [11]:
```
add_array_time = pd.read_csv('add_array_time.csv')
print(add_array_time)
add_array_time = add_array_time.values
# Empirical analysis
x = add_array_time[:,0]
y = add_array_time[:,1]/1000
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c = 0.00004;
plt.plot(x, (c*x*x)/1000, label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Time in sec')
plt.title('Empirical and Asymptotic analysis of time complexity of array add')
plt.legend()
```

```
   Matrix size n  Time taken in ms
0             32          0.076313
1             64          0.272781
2            128          1.062193
3            256          2.178393
4            512          8.767897
5           1024         39.173371
```

```
6             2048        154.673751
7             4096        624.444223
8             8192       2506.031043
9            16384       9668.181594
10           32768      40177.521355
```

```
Out[11]: <matplotlib.legend.Legend at 0x7f69f6014908>
```




**Observations for array implementation**

1. From the Memory vs matrix size plot we see that in case of addition the memory usage is liitle more than twice the memory usage in case of normal load operation. This is expected as now we are dealing with two matrices of size nxn. We can approximately calculate the factor increase in this case by taking the ratio of c obtained in this case with the c obtained in normal load case which is $0.013/0.0045 = 2.88$

2. From the Memory vs matrix size plot we can see that worst case space complexity is of the order of $O(0.013n^2)$ which is $O(n^2)$

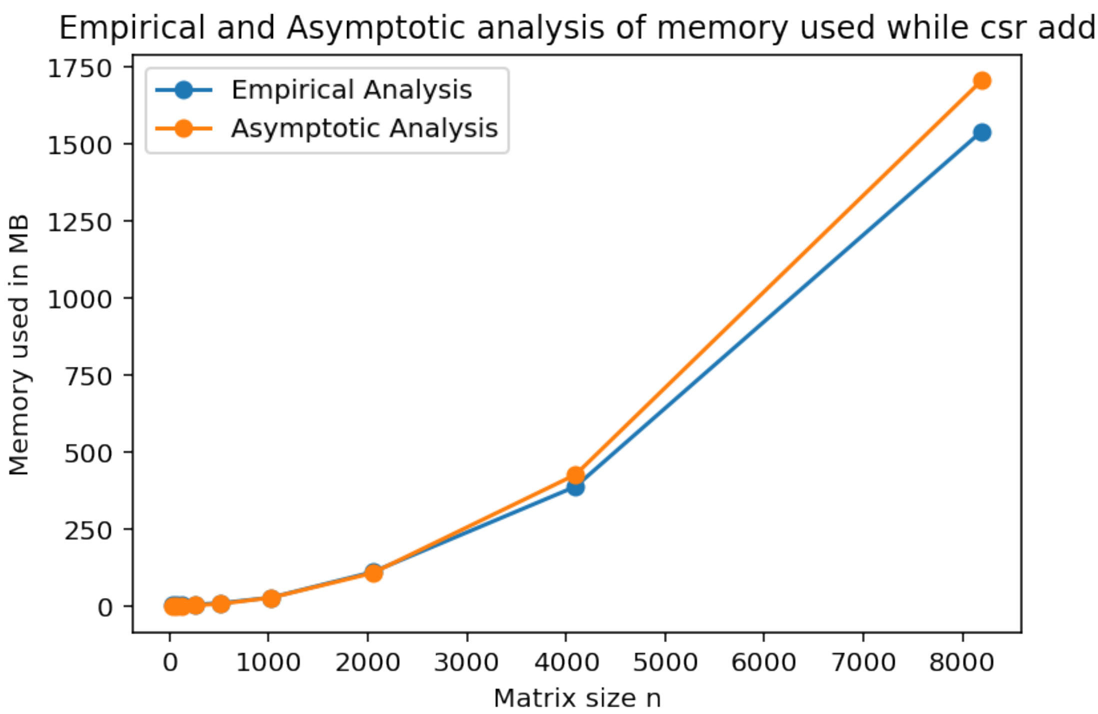
3. The time complexity of add is theoretically expected to be $O(n^2)$ as we are element wise adding all the $n^2$ elements. The worst case time complexity can also be seen from the curve to ve $O(0.00004n^2)$ which is also $O(n^2)$.

```
In [12]: add_csr_mem = pd.read_csv('add_csr_mem.csv')
         print(add_csr_mem)
         add_csr_mem = add_csr_mem.values
         # Empirical analysis
         x = add_csr_mem[:,0]
         y = add_csr_mem[:,1]/1024
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.026
         plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
         plt.xlabel('Matrix size n')
         plt.ylabel('Memory used in MB')
         plt.title('Empirical and Asymptotic analysis of memory used while csr add')
         plt.legend()
```

```
   Matrix size n  Memory used in KB
0             32               1612
1             64               1612
2            128               3524
3            256               4708
4            512               9252
5           1024              27748
6           2048             112452
7           4096             396524
8           8192            1576076
```

```
Out[12]: <matplotlib.legend.Legend at 0x7f69f5f8df98>
```



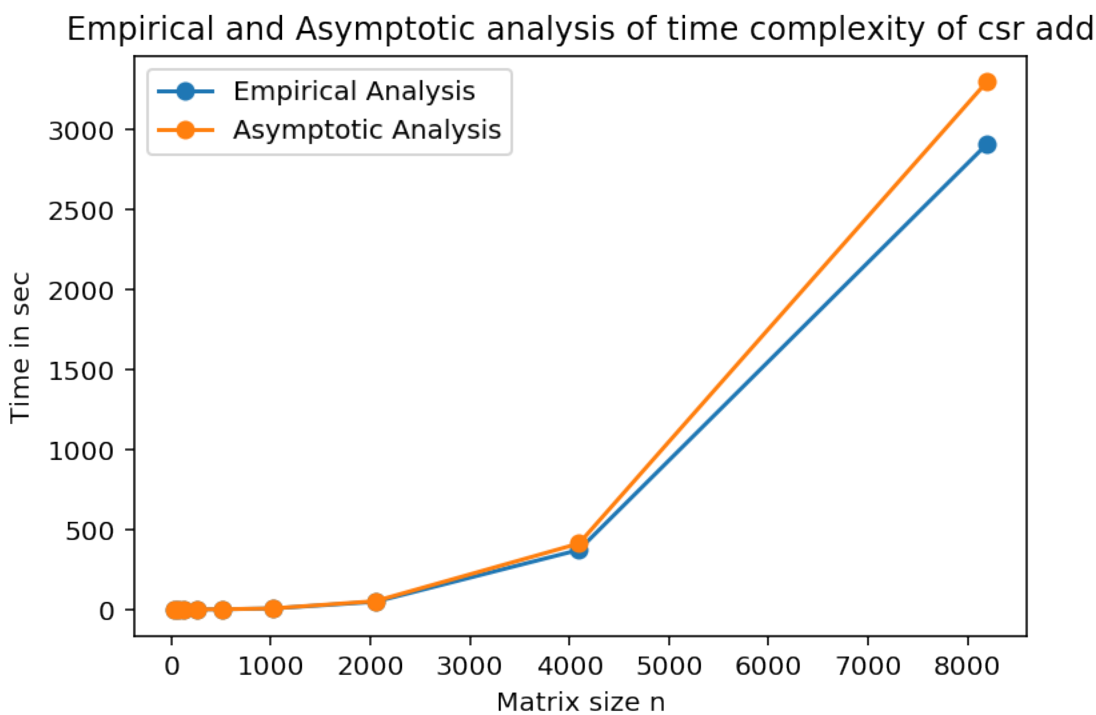

```
In [13]: add_csr_time = pd.read_csv('add_csr_time.csv')
         print(add_csr_time)
         add_csr_time = add_csr_time.values
         # Empirical analysis
         x = add_csr_time[:,0]
         y = add_csr_time[:,1]/1000
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c = 0.000006;
         y = np.power(x,3)
         plt.plot(x, (c*y)/1000, label='Asymptotic Analysis',marker='o')
         plt.xlabel('Matrix size n')
         plt.ylabel('Time in sec')
         plt.title('Empirical and Asymptotic analysis of time complexity of csr add')
         plt.legend()
```

```
   Matrix size n  Time taken in ms
0             32      5.834510e-01
1             64      4.238848e+00
2            128      1.564635e+01
3            256      9.776024e+01
4            512      7.574256e+02
5           1024      5.902395e+03
```

```
6          2048     4.716333e+04
7          4096     3.724040e+05
8          8192     2.908749e+06
```

```
Out[13]: <matplotlib.legend.Legend at 0x7f69f5ef25f8>
```




**Observations for csr implementation**

1. The memory consumption for csr representation has almost incresed by the same factor as in array case when compared to load with csr. This can be seen by taking the ratio of the two constants obtained in the two cases which is $0.026/0.009 = 2.88$

2. From the memory vs input size plot in case of csr we can see that worst case space complexity is $(0.026n^2)$ which is $O(n^2)$

3. From the time vs input size plot we see that the worst case time complexity is $O(0.000006n^3)$ which is also $O(n^3)$. The time complexity in this case is $O(n^3)$ because the get(i,j) method of csr is costly. This is because given an i and j we first find the number of non zero elements in the given row from the cumulative count array and then for each of those non zero elements in the row we check their column indices if equal to j. This have worst case time complexity of $O(n)$ in case where there are all non zero elements in a given row. This makes the overall time complexity to be $O(n^2)$.

### 1.4.2 Results for sparse data

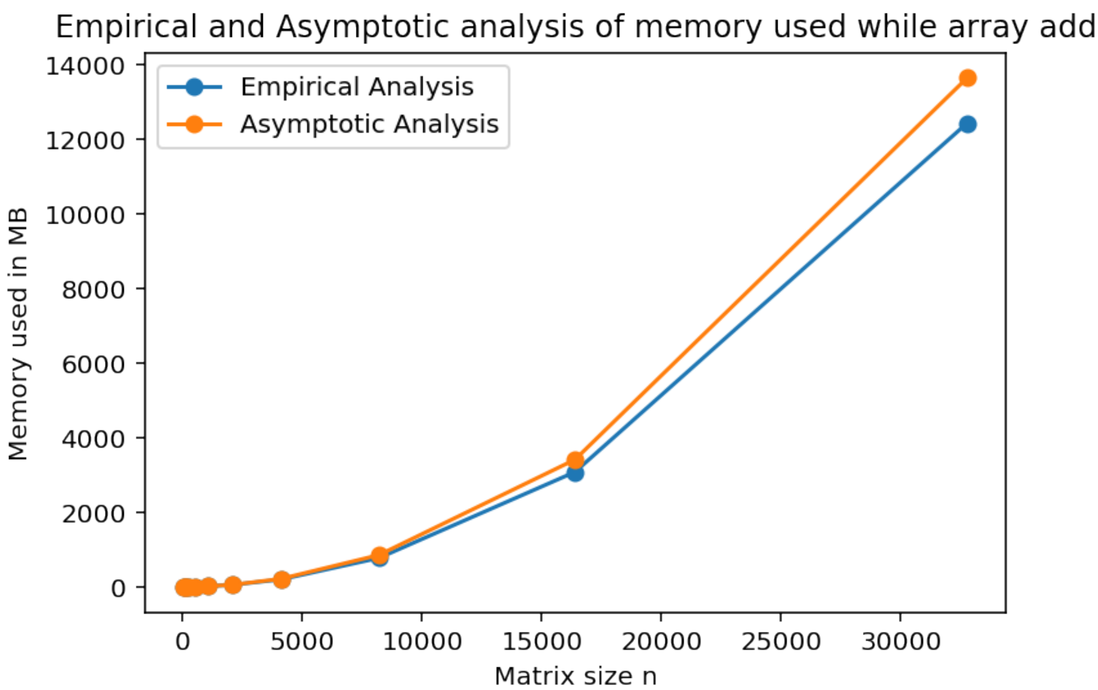
The sparse data was generated with sparsity factor of 80%.

```
In [14]: add_sparse_array_mem = pd.read_csv('add_sparse_array_mem.csv')
         print(add_sparse_array_mem)
         add_sparse_array_mem = add_sparse_array_mem.values
         # Empirical analysis
         x = add_sparse_array_mem[:,0]
         y = add_sparse_array_mem[:,1]/1024
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.013
         plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
         plt.xlabel('Matrix size n')
         plt.ylabel('Memory used in MB')
         plt.title('Empirical and Asymptotic analysis of memory used while array add')
         plt.legend()
```

```
    Number of vertices n  Memory used in KB
0                     32               1616
1                     64               1616
2                    128               3264
3                    256               3792
4                    512               6044
5                   1024              15408
6                   2048              52368
7                   4096             199996
8                   8192             790148
9                  16384            3150192
10                 32768           12719200
```

```
Out[14]: <matplotlib.legend.Legend at 0x7f69f5e73f60>
```



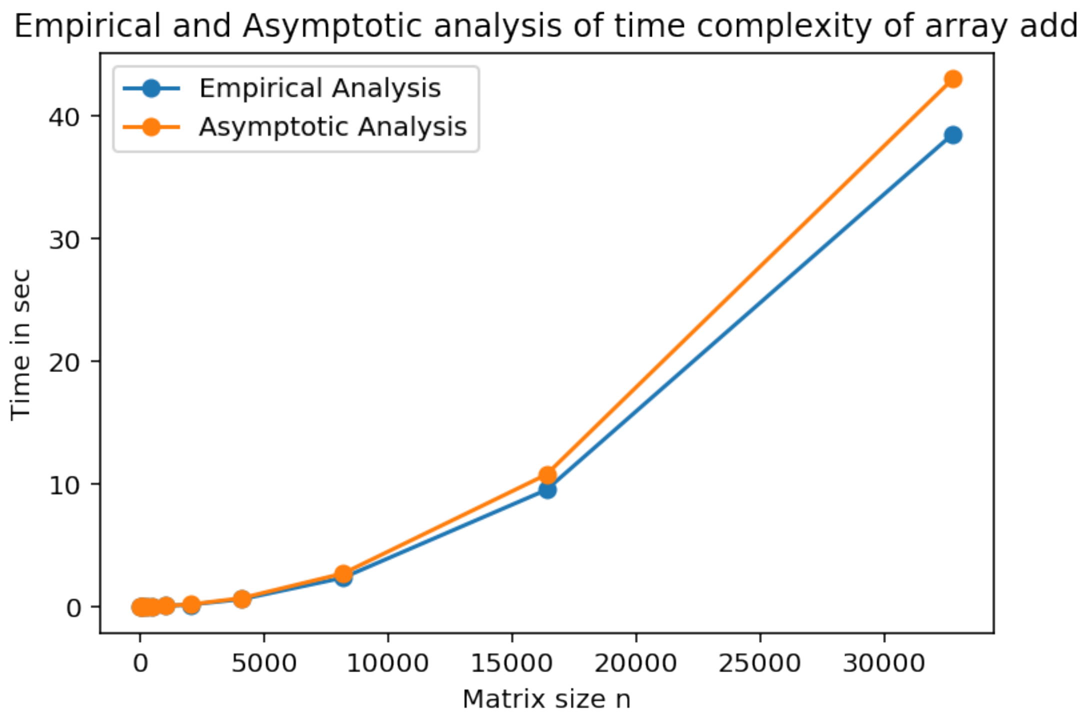

In [15]:
```
add_sparse_array_time = pd.read_csv('add_sparse_array_time.csv')
print(add_sparse_array_time)
add_sparse_array_time = add_sparse_array_time.values
# Empirical analysis
x = add_sparse_array_time[:,0]
y = add_sparse_array_time[:,1]/1000
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c = 0.00004;
plt.plot(x, (c*x*x)/1000, label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Time in sec')
plt.title('Empirical and Asymptotic analysis of time complexity of array add')
plt.legend()
```

```
   Matrix size n  Time in ms
0             32      0.0757
1             64      0.2834
2            128      0.5435
3            256      1.8360
4            512      8.4766
5           1024     36.7567
6           2048    144.1742
7           4096    580.7530
```

```
8            8192   2334.4311
9           16384   9489.5494
10          32768  38464.0341
```

```
Out[15]: <matplotlib.legend.Legend at 0x7f69f5e52a20>
```




**Observations for array implementation**

1. The space complexity remains the same as in the dense case i.e. $O(n^2)$
2. The time complexity is also same as before i.e $O(n^2)$

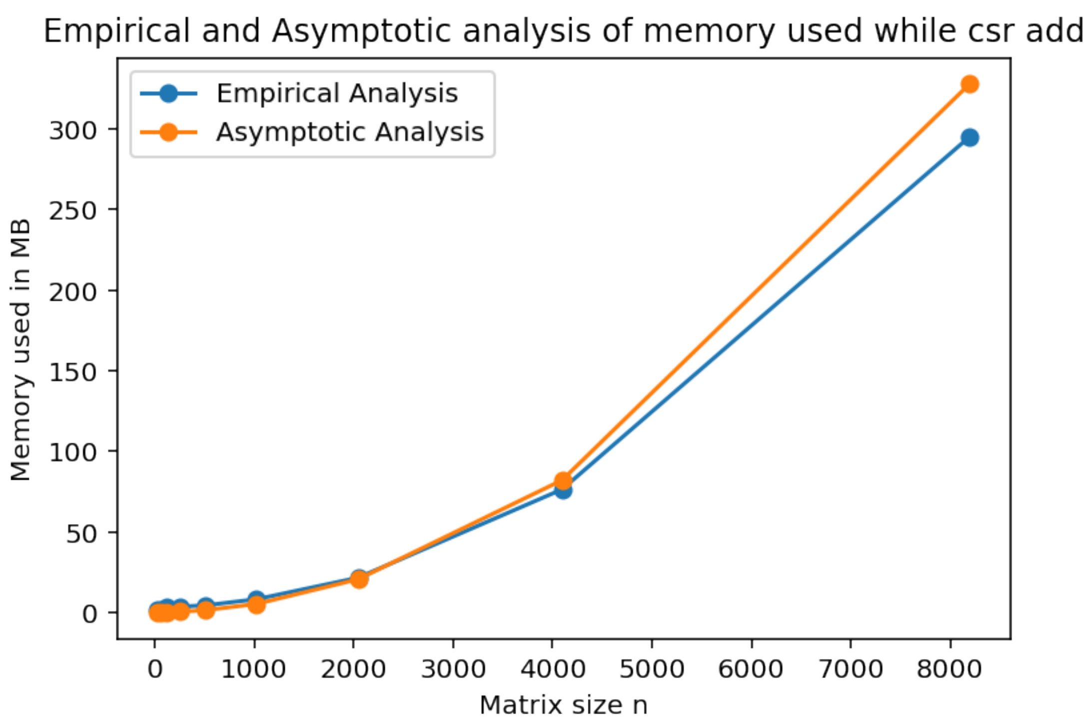
```
In [16]: add_sparse_csr_mem = pd.read_csv('add_sparse_csr_mem.csv')
         print(add_sparse_csr_mem)
         add_sparse_csr_mem = add_sparse_csr_mem.values
         # Empirical analysis
         x = add_sparse_csr_mem[:,0]
         y = add_sparse_csr_mem[:,1]/1024
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.005
         plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
```

```
        plt.xlabel('Matrix size n')
        plt.ylabel('Memory used in MB')
        plt.title('Empirical and Asymptotic analysis of memory used while csr add')
        plt.legend()
```

```
   Matrix size n  Memory used in KB
0             32               1616
1             64               1616
2            128               3264
3            256               3528
4            512               4408
5           1024               8260
6           2048              21992
7           4096              78032
8           8192             301984
```

```
Out[16]: <matplotlib.legend.Legend at 0x7f69f5dcefd0>
```



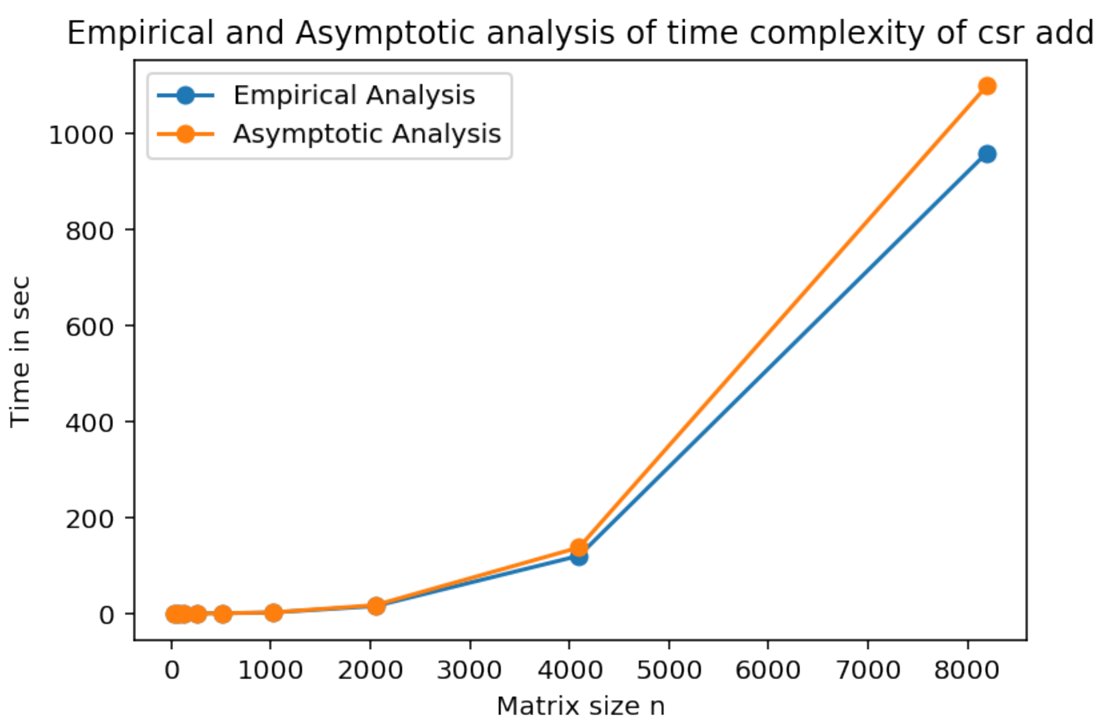

```
In [17]: add_sparse_csr_time = pd.read_csv('add_sparse_csr_time.csv')
         print(add_sparse_csr_time)
         add_sparse_csr_time = add_sparse_csr_time.values
         # Empirical analysis
         x = add_sparse_csr_time[:,0]
```

```
y = add_sparse_csr_time[:,1]/1000
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c = 0.000002;
y = np.power(x,3)
plt.plot(x, (c*y)/1000, label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Time in sec')
plt.title('Empirical and Asymptotic analysis of time complexity of csr add')
plt.legend()
```

|   | Matrix size n | Time in ms |
|---|---|---|
| 0 | 32 | 0.3251 |
| 1 | 64 | 1.8009 |
| 2 | 128 | 5.2876 |
| 3 | 256 | 33.1668 |
| 4 | 512 | 248.0601 |
| 5 | 1024 | 1928.3961 |
| 6 | 2048 | 15072.1288 |
| 7 | 4096 | 120168.8706 |
| 8 | 8192 | 959803.9203 |

```
Out[17]: <matplotlib.legend.Legend at 0x7f69f5d44f98>
```



**Observations for csr implementation**

1. The space complexity is now $O((1 - sparsity) * n^2)$ which is also $O(n^2)$ in worst case. This can be also be verified for ex: consider n=8192 the memory usage in case of sparse data is $301984 KB$ which is almost 20% of that used in dense case.

2. The time complexity in this case is much lesser than in the dense case. And roughly we can say that best case time complexity is $O((1 - sparsity) * n^3)$ if the matrix is sparse in such a way that the non zero elements are almost uniformly distributed so that each row is expected to contain $(1 - sparsity) * n$ non zero elements. Still the worst case time complexity remains $O(n^2)$

**Conclusion** In case of addition there are some pros and cons of both the implementations

1. The array implementation is best if its a dense matrix , it occupies $O(n^2)$ space and the addition of two matrices can be completed in $O(n^2)$ time.

2. If the matrices to be added are sparse then csr implementation proves to be useful as it requires $O((1 - sparsity) * n^2)$ of memory but the additions operation can prove costly if the matrix is not sufficiently sparse as it takes approx $O((1 - sparsity) * n^3)$ time due to costly get operation.

## 1.5 Multiplication of two matrices

### 1.5.1 Results for dense data

The results are obtained for n ranging from 32 to 2048 in case of array implementation and 32 to 1024 in case of csr implementation.

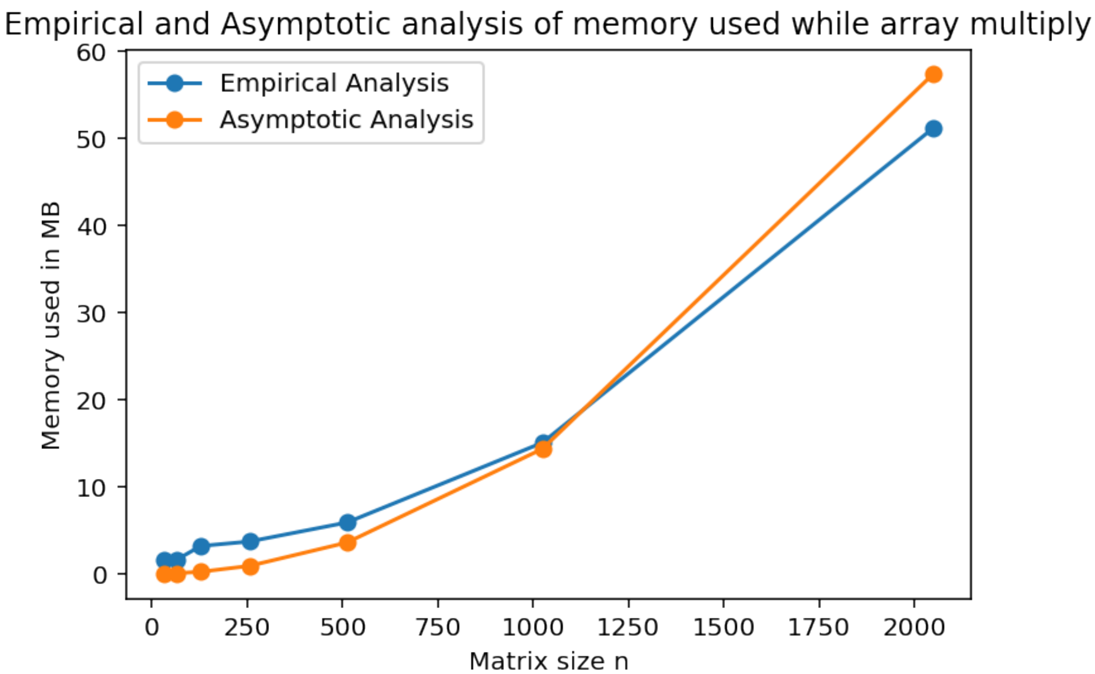
```
In [18]: multiply_array_mem = pd.read_csv('multiply_array_mem.csv')
         print(multiply_array_mem)
         multiply_array_mem = multiply_array_mem.values
         # Empirical analysis
         x = multiply_array_mem[:,0]
         y = multiply_array_mem[:,1]/1024
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.014
         plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
         plt.xlabel('Matrix size n')
         plt.ylabel('Memory used in MB')
         plt.title('Empirical and Asymptotic analysis of memory used while array multiply')
         plt.legend()
```

```
   Matrix size n  Memory used in KB
0             32               1612
1             64               1612
2            128               3260
3            256               3788
```

```
4            512              6028
5           1024             15404
6           2048             52364
```

```
Out[18]: <matplotlib.legend.Legend at 0x7f69f5d22940>
```



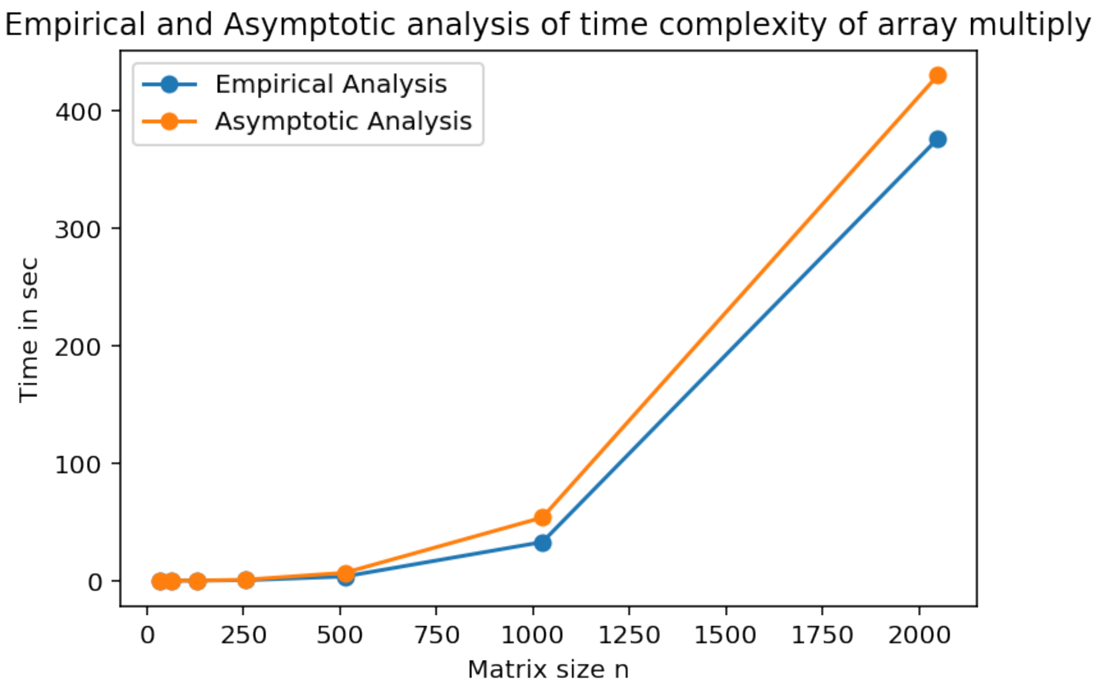

```
In [19]: multiply_array_time = pd.read_csv('multiply_array_time.csv')
         print(multiply_array_time)
         multiply_array_time = multiply_array_time.values
         # Empirical analysis
         x = multiply_array_time[:,0]
         y = multiply_array_time[:,1]/1000
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.00005
         y = np.power(x,3)
         plt.plot(x, (c*y)/(1000), label='Asymptotic Analysis',marker='o')
         plt.xlabel('Matrix size n')
         plt.ylabel('Time in sec')
         plt.title('Empirical and Asymptotic analysis of time complexity of array multiply')
         plt.legend()
```

```
   Matrix size n  Time taken in ms
0             32          1.732421
```

```
1          64       9.601549
2         128      44.780586
3         256     401.937713
4         512    3549.884462
5        1024   32825.011947
6        2048  375698.135671
```

```
Out[19]: <matplotlib.legend.Legend at 0x7f69f5c8f7f0>
```




**Observation for array implementation**

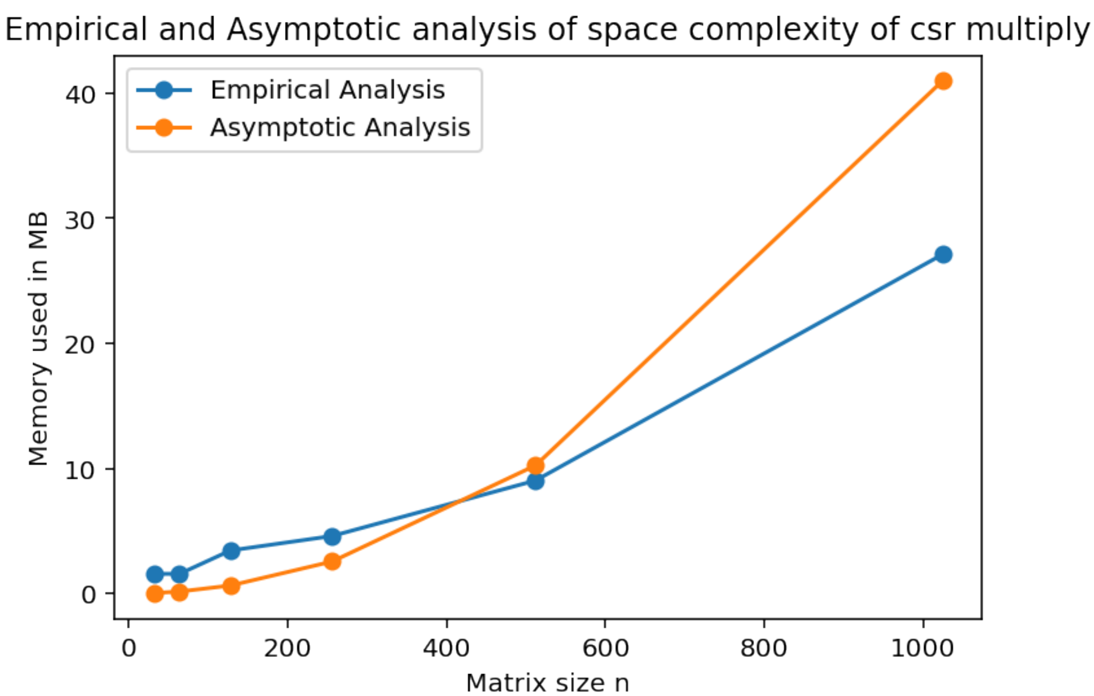
1. The space complexity is $O(n^2)$ as we still need to store $n * n$ elements of the two matrices to be multiplied
2. The time complexity is $O(n^3)$ this is because we have 3 nested loops running from 1 to n and the get function in case of array implementation takes $O(1)$ time.

```
In [20]: multiply_csr_mem = pd.read_csv('multiply_csr_mem.csv')
         print(multiply_csr_mem)
         multiply_csr_mem = multiply_csr_mem.values
         # Empirical analysis
         x = multiply_csr_mem[:,0]
         y = multiply_csr_mem[:,1]/1024
         plt.plot(x,y,label='Empirical Analysis',marker='o')
```

```
# Asymptotic analysis
c=0.04
plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Memory used in MB')
plt.title('Empirical and Asymptotic analysis of space complexity of csr multiply')
plt.legend()
```

```
   Matrix size n  Memory used in KB
0             32               1612
1             64               1612
2            128               3524
3            256               4708
4            512               9252
5           1024              27748
```

```
Out[20]: <matplotlib.legend.Legend at 0x7f69f5c00978>
```



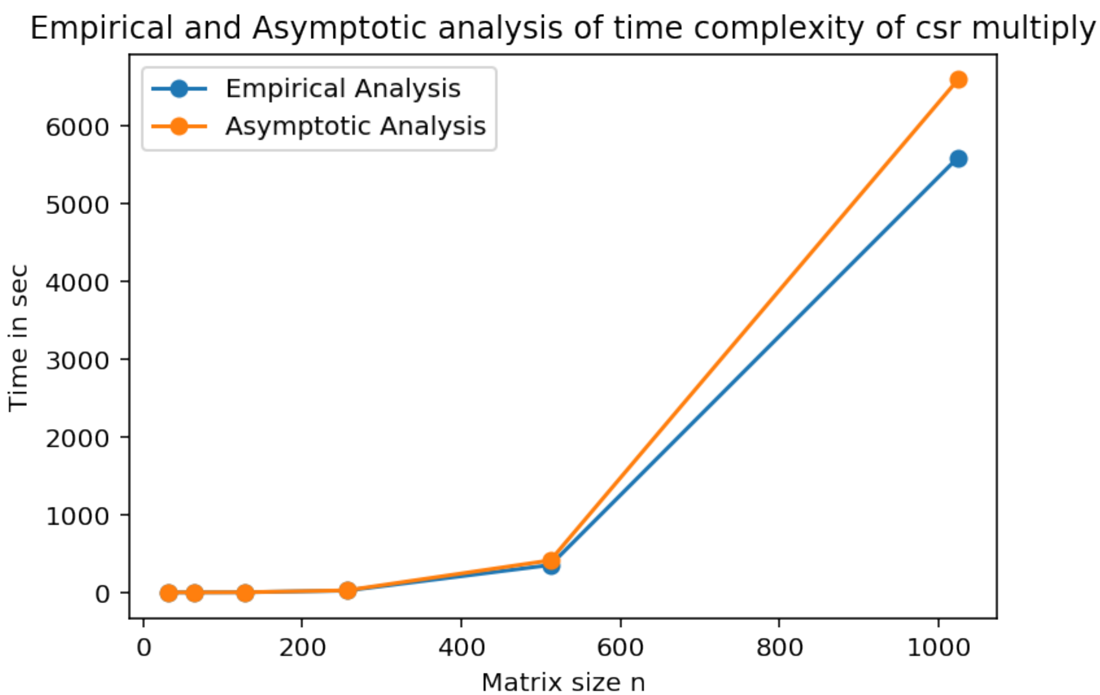

```
In [21]: multiply_csr_time = pd.read_csv('multiply_csr_time.csv')
         print(multiply_csr_time)
         multiply_csr_time = multiply_csr_time.values
         # Empirical analysis
         x = multiply_csr_time[:,0]
```

```
y = multiply_csr_time[:,1]/1000
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c=0.000006
y = np.power(x,4)
plt.plot(x, (c*y)/(1000), label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Time in sec')
plt.title('Empirical and Asymptotic analysis of time complexity of csr multiply')
plt.legend()
```

```
   Matrix size n  Time taken in ms
0             32      1.462717e+01
1             64      1.025147e+02
2            128      1.515804e+03
3            256      2.273023e+04
4            512      3.517397e+05
5           1024      5.594735e+06
```

```
Out[21]: <matplotlib.legend.Legend at 0x7f69f5b7bcf8>
```




**Observations for csr implementation**

1. In dense case the space complexity of csr representation is also $O(n^2)$ as in the worst case we require to store all the $n * n$ non zero elements.
2. But the worst case time complexity is $O(n^4)$ this is because of the expensive get function which is getting called everytime inside the 3 nested loops. The get function in worst case does $O(n)$ operations when all the numbers in a given row are non zero.

### 1.5.2 Results for sparse data

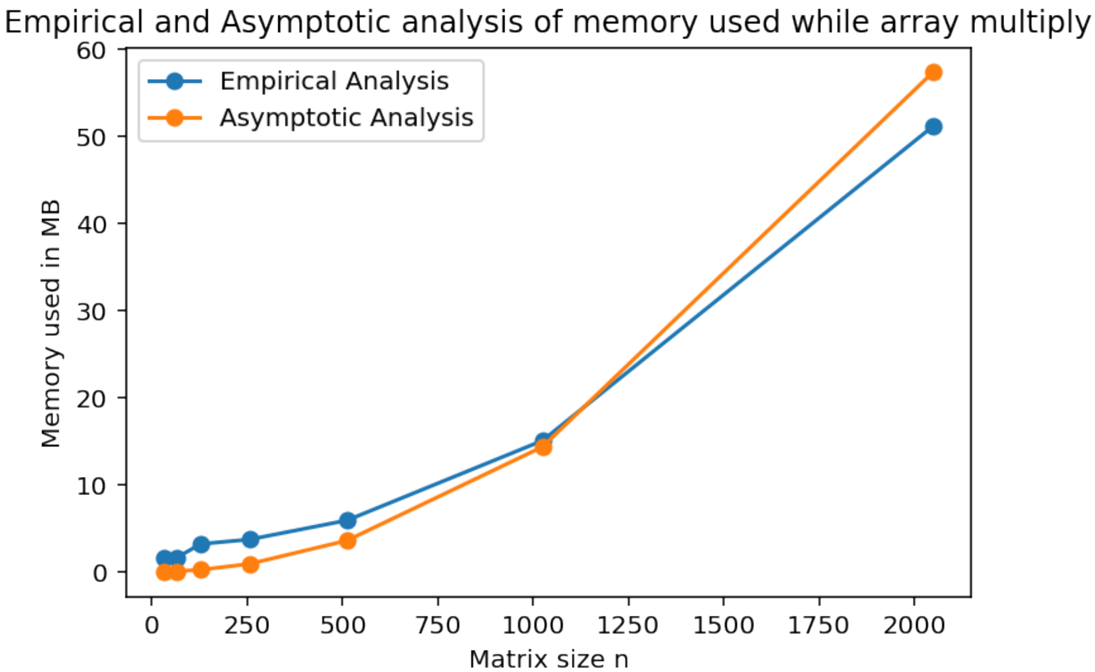
The data was generated with a sparsity factor of 80% i.e 80% of the total elements in the matrix are zeroes.

In [22]:
```
multiply_sparse_array_mem = pd.read_csv('multiply_sparse_array_mem.csv')
print(multiply_sparse_array_mem)
multiply_sparse_array_mem = multiply_sparse_array_mem.values
# Empirical analysis
x = multiply_sparse_array_mem[:,0]
y = multiply_sparse_array_mem[:,1]/1024
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c=0.014
plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Memory used in MB')
plt.title('Empirical and Asymptotic analysis of memory used while array multiply')
plt.legend()
```

```
   Number of vertices n  Memory used in KB
0                    32               1616
1                    64               1616
2                   128               3264
3                   256               3792
4                   512               6044
5                  1024              15408
6                  2048              52368
```

Out[22]:
```
<matplotlib.legend.Legend at 0x7f69f5b50d68>
```

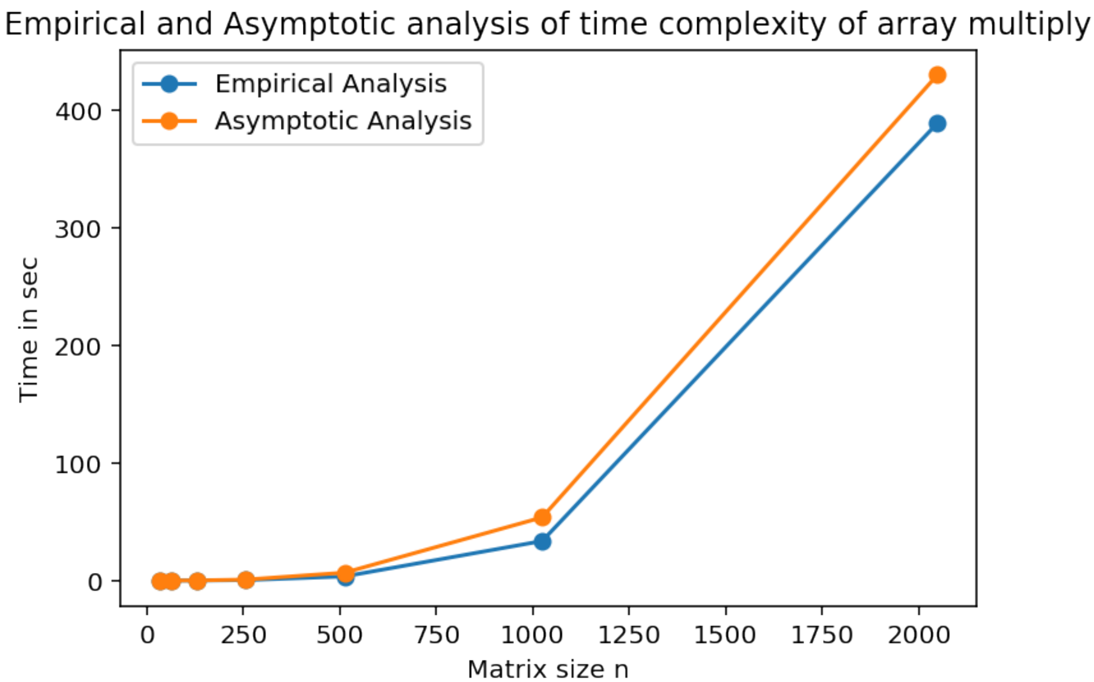



In [23]:
```
multiply_sparse_array_time = pd.read_csv('multiply_sparse_array_time.csv')
print(multiply_sparse_array_time)
multiply_sparse_array_time = multiply_sparse_array_time.values
# Empirical analysis
x = multiply_sparse_array_time[:,0]
y = multiply_sparse_array_time[:,1]/1000
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c=0.00005
y = np.power(x,3)
plt.plot(x, (c*y)/(1000), label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Time in sec')
plt.title('Empirical and Asymptotic analysis of time complexity of array multiply')
plt.legend()
```

```
   Matrix size n    Time in ms
0             32        1.6364
1             64       12.5010
2            128       46.7682
3            256      404.7839
4            512     3523.2335
5           1024    33592.4600
6           2048   388489.8718
```

Out[23]: <matplotlib.legend.Legend at 0x7f69f5ac5cc0>




**Observation for array implemetation**

1. For sparse data the space complexity is same as before i.e. $O(n^2)$.
2. The time complexity is also same as before i.e $O(n^3)$.

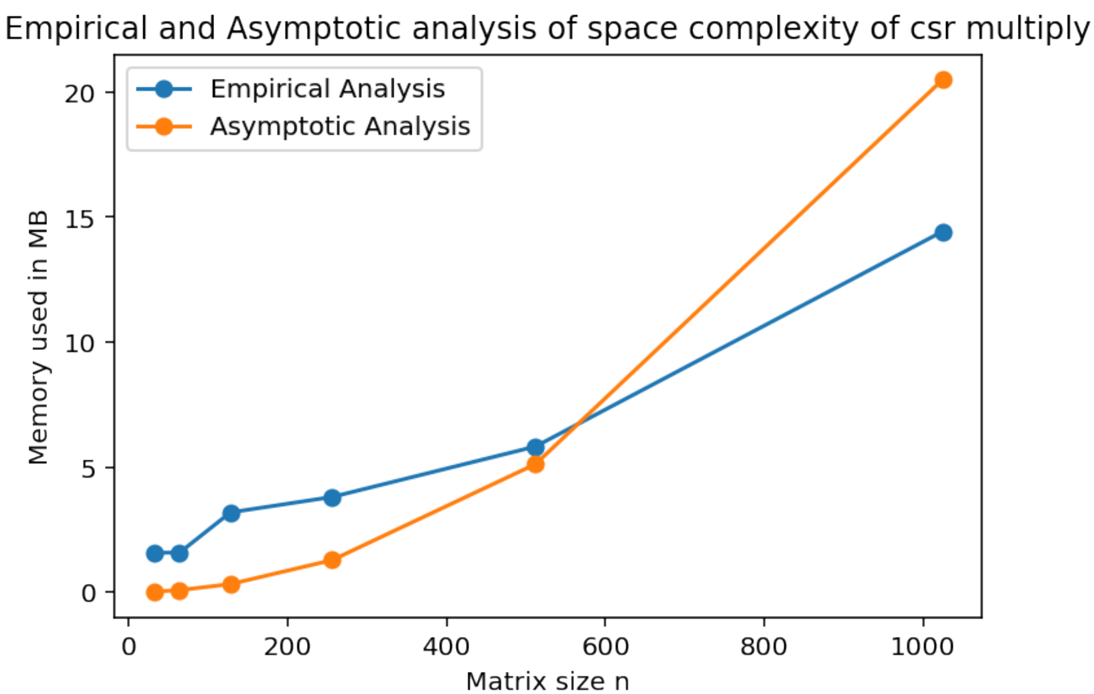
In [24]:
```
multiply_sparse_csr_mem = pd.read_csv('multiply_sparse_csr_mem.csv')
print(multiply_sparse_csr_mem)
multiply_sparse_csr_mem = multiply_sparse_csr_mem.values
# Empirical analysis
x = multiply_sparse_csr_mem[:,0]
y = multiply_sparse_csr_mem[:,1]/1024
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c=0.02
plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Memory used in MB')
plt.title('Empirical and Asymptotic analysis of space complexity of csr multiply')
plt.legend()
```

```
   Matrix size n  Memory used in KB
0             32               1616
```

```
1          64          1616
2         128          3264
3         256          3896
4         512          5964
5        1024         14764
```

Out[24]: <matplotlib.legend.Legend at 0x7f69f5a4bfd0>

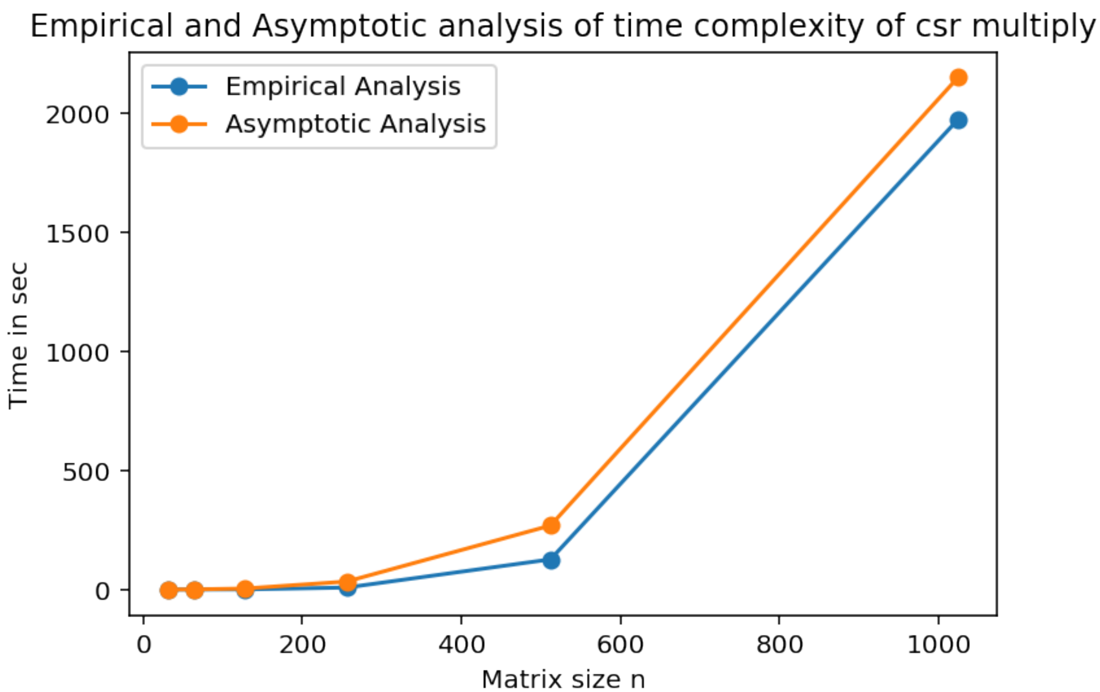



In [25]:
```
multiply_sparse_csr_time = pd.read_csv('multiply_sparse_csr_time.csv')
print(multiply_sparse_csr_time)
multiply_sparse_csr_time = multiply_sparse_csr_time.values
# Empirical analysis
x = multiply_sparse_csr_time[:,0]
y = multiply_sparse_csr_time[:,1]/1000
plt.plot(x,y,label='Empirical Analysis',marker='o')

# Asymptotic analysis
c=0.002
y = np.power(x,3)
plt.plot(x, (c*y)/(1000), label='Asymptotic Analysis',marker='o')
plt.xlabel('Matrix size n')
plt.ylabel('Time in sec')
plt.title('Empirical and Asymptotic analysis of time complexity of csr multiply')
plt.legend()
```

|   | Matrix size n | Time in ms |
|---|---|---|
| 0 | 32 | 7.498500e+00 |
| 1 | 64 | 5.110940e+01 |
| 2 | 128 | 5.845789e+02 |
| 3 | 256 | 8.344956e+03 |
| 4 | 512 | 1.265416e+05 |
| 5 | 1024 | 1.971335e+06 |

```
Out[25]: <matplotlib.legend.Legend at 0x7f69f59acfd0>
```




**Observation for csr representation**

1. The space complexity reduces to $O(1 - sparsity) * n^2)$ as there are only $(1 - sparsity) * n * n$ elements which is still $O(n^2)$ in worst case if the sparsity factor is too low.
2. The time complexity reduces to $O((1 - sparsity) * n^4)$ if the matrix is sparse in such a way that its non zero elements are present uniformly in each row i.e approx $(1 - sparsity) * n$ in each row. Still the worst case can be $O(n^2)$ in case of low sparsity.

**Conclusion**

1. Array implementation is best in case of dense matrices as it can multiply two matrices in $O(n^3)$ time as compared to csr which takes $O(n^4)$.
2. CSR works best in cases when the matrices are sufficiently sparse in such a case it reduces the space complexity to $O((1 - sparsity) * n^2)$ and time complexity to $O((1 - sparsity) * n^4)$

## 1.6 Breadth First Search

### 1.6.1 With sparse graph using csr representation

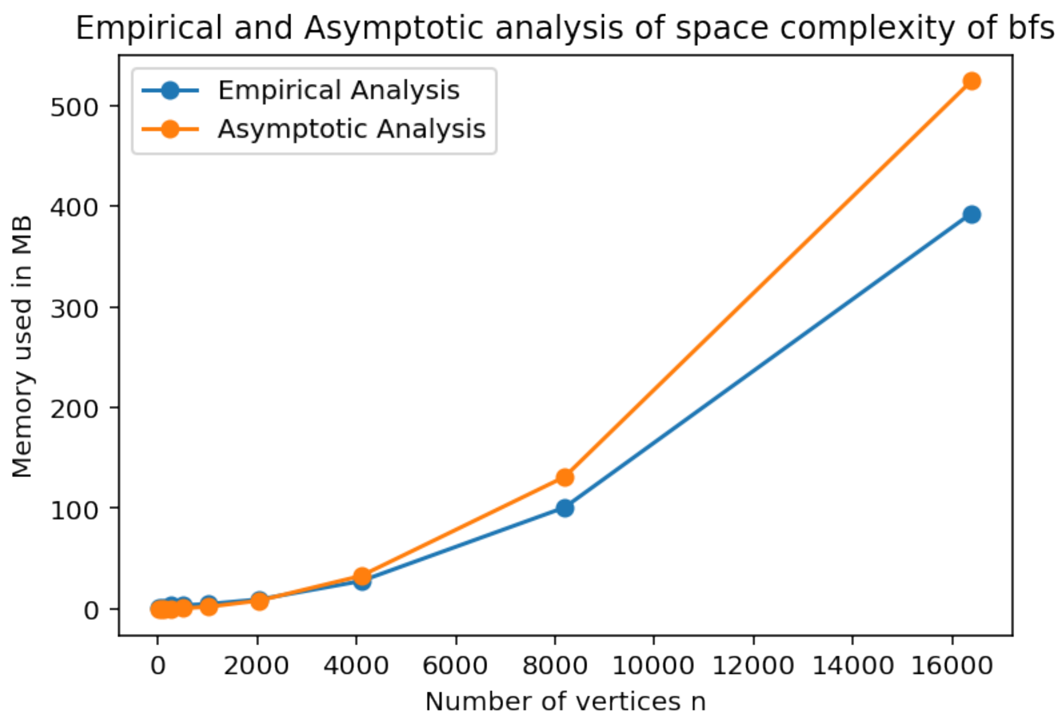
BFS has been implemented using csr matrix representation for number of vertices ranging from 32 to 16384 on a sparse graph with sparsity=0.8.

```
In [26]: bfs_mem = pd.read_csv('bfs_mem.csv')
         print(bfs_mem)
         bfs_mem = bfs_mem.values
         # Empirical analysis
         x = bfs_mem[:,0]
         y = bfs_mem[:,1]/1024
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.002
         plt.plot(x, (c*x*x)/(1024), label='Asymptotic Analysis',marker='o')
         plt.xlabel('Number of vertices n')
         plt.ylabel('Memory used in MB')
         plt.title('Empirical and Asymptotic analysis of space complexity of bfs')
         plt.legend()
```

```
   Number of vertices n  Memory used in KB
0                    32               1616
1                    64               1616
2                   128               1616
3                   256               3264
4                   512               3672
5                  1024               4940
6                  2048               9624
7                  4096              28220
8                  8192             103124
9                 16384             402296
```

```
Out[26]: <matplotlib.legend.Legend at 0x7f69f5982828>
```



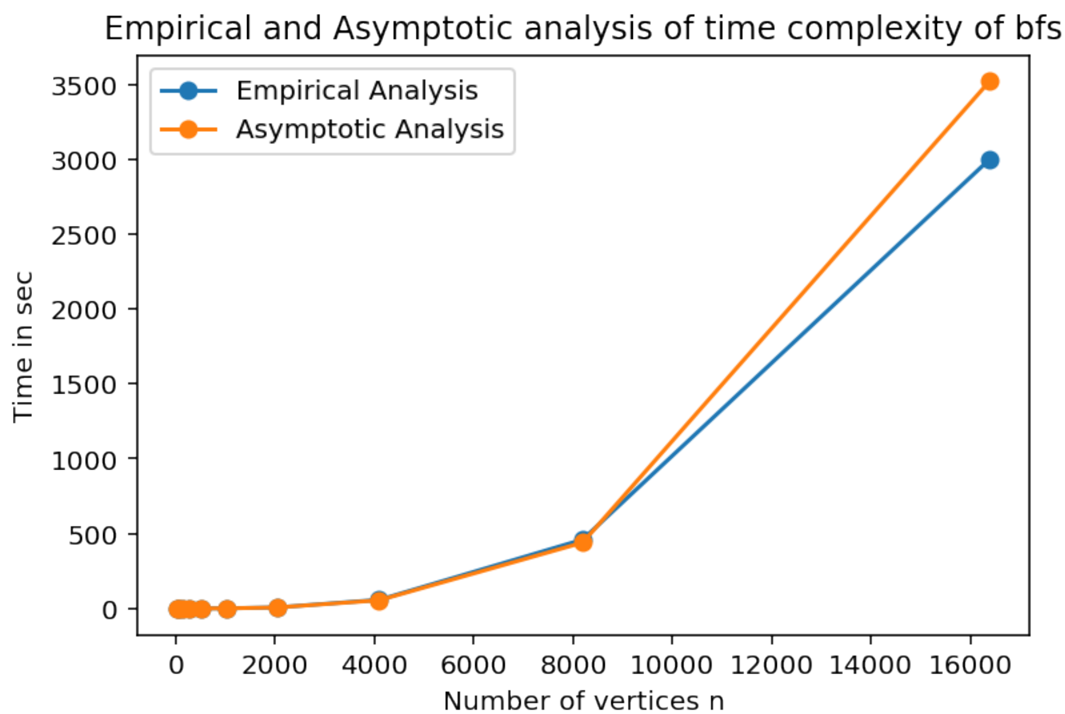

```
In [27]: bfs_time = pd.read_csv('bfs_time.csv')
         print(bfs_time)
         bfs_time = bfs_time.values
         # Empirical analysis
         x = bfs_time[:,0]
         y = bfs_time[:,1]/1000
         plt.plot(x,y,label='Empirical Analysis',marker='o')

         # Asymptotic analysis
         c=0.0000008
         y = np.power(x,3)
         plt.plot(x, (c*y)/(1000), label='Asymptotic Analysis',marker='o')
         plt.xlabel('Number of vertices n')
         plt.ylabel('Time in sec')
         plt.title('Empirical and Asymptotic analysis of time complexity of bfs')
         plt.legend()
```

```
   Number of vertices n    Time in ms
0                    32  1.665000e-01
1                    64  8.498000e-01
2                   128  2.232000e+00
3                   256  2.154980e+01
4                   512  1.205141e+02
5                  1024  9.255263e+02
```

```
6                    2048   7.281977e+03
7                    4096   5.772812e+04
8                    8192   4.599964e+05
9                   16384   2.997502e+06
```

Out[27]: <matplotlib.legend.Legend at 0x7f69f5905f98>




**Observations for BFS**

1. As we have seen in the case of csr the space complexity is $O((1 - sparsity) * n^2)$ the worst case being $O(n^2)$ without sparsity. Therefore depending on the number of edges in the graph we would have those many entries in the csr matrix which in worst case is $O(n^2)$. The curve for memory against number of vertices captures that relation. For eg. In load csr case for dense data, for input size 8192 the memory used is $527820KB$ now here we have a sparse graph with sparsity factor of 0.8. Therefore we should expect that the memory used in this case should be $527820 * 0.2 = 105564$ which is actually the case as the result obtained empirically for $n = 8192$ is $103124KB$

2. From the memory vs number of vertices graph we see that space complexity is $O(0.002 * n^2)$ which is $O(n^2)$

3. From the time vs number of vertices plot we see that time complexity is $O(0.0000008 * n^3)$ which is $O(n^2)$. This is expected because for a given vertex v we check all n vertices if it

has an edge with v. And checking if it has an edge with a vertex i requires us to call the get function which we have seen has worst case of $O(n)$. So there are n vertices in total and every vertex get pushed to the queue if we assume that all n nodes are connected. Therefore n checks for n vertices and $O(n)$ for get operation in worst case leads to $O(n^3)$. In the code I have implemented there is another loop within the while(!q.empty()) loop that searches for the set to which the popped vertex belongs to which is $O(depth)$ of the graph which in worst case could be $O(n)$. But that doesnt affect the overall time complexity as it is still $O(n^3)$.

4. The largest graph on which I could run bfs within reasonable time was for n=16384 which took roughly 50 mins to run.

**Conclusion**

1. Worst case space complexity of implementing bfs with csr implementation of graph is $O(n^2)$ and worst case time complexity is $O(n^3)$ where n being the number of vertices.