CSCE 221 - 504 Homework 1

Daniel Bueso-Mendoza 117005010

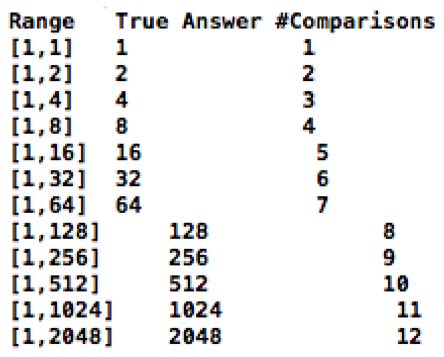
September 6, 2013

1 Random Number Game

The random number game consists of 2 players. One player selects a random number within a certain range. It is then the job of the other player to guess this number with the minimal number of questions.

1.1 Binary Search Algorithm

The implementation of the random number guessing game was accomplished through the use of the *binary search algorithm.* For a number guessing game the binary search algorithm takes a range of numbers in a sorted array and selects the middle value as its guess. One can then say if this number is higher, lower, or correct in reference to the target value. If the guess is correct then that position is simply returned. For the higher case, the lower range is eliminated and only the values to the *right* are taken into account. If lower, the reverse is enforced. Values to the right of the guess are disposed and only the *left* values are observed. The algorithm repeats in each iteration by selecting the middle term of the newly created sub-arrays as its guess until the item is found.



```
Range    True Answer #Comparisons
[1,1]    1           1
[1,2]    2           2
[1,4]    4           3
[1,8]    8           4
[1,16]   16            5
[1,32]   32            6
[1,64]   64            7
[1,128]      128             8
[1,256]      256             9
[1,512]      512             10
[1,1024]     1024              11
[1,2048]     2048              12
```

Figure 1: Tabulated Data of Guessing Game

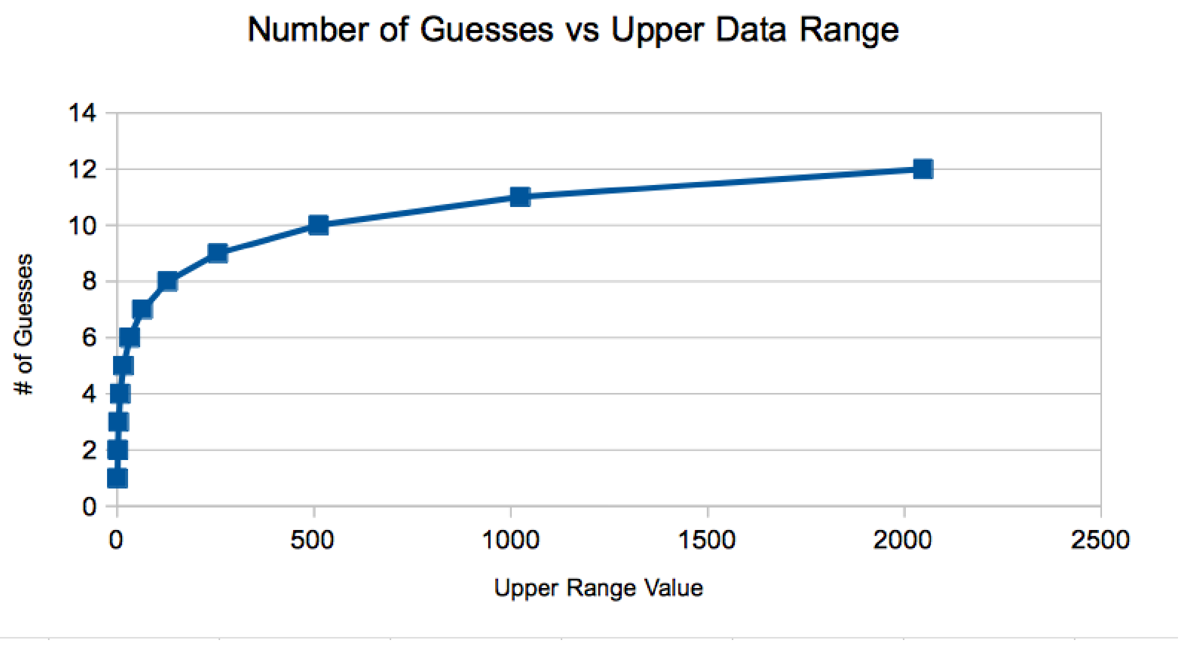


Figure 2: Number of Guesses vs Range

1.2 Guessing Game on Twelve Tests

Twelve tests were run in the range (x) as presented in Figure 1. The value to guess was given by $n = 2^k$, where $k = 0, 1, 2, ..., 11$. Figure 2 depicts the number of guesses required to reach the correct value. The figure reveals a logarithmic relationship as the range is increased.

1.3 Mathematical Formula on Twelve Tests

A formula that closely resembles the data received is: $f(n) = log_2(n) + 1$. The value n reflects the upper value for testing. Two examples follow:

$$f(64) = log_2(64) + 1 = 7$$

$$f(2048) = log_2(2048) + 1 = 12$$

One can observe Figure 1 and see that the # of comparisons results in these values.

1.4 Guessing Game on Eleven Tests

The 11 tests performed here were computed differently than the previous 12 test case. Here the value to guess is acquired from the equation $n = 2^k - 1$, where $k = 1, 2, ..., 11$. The "answer" to guess ends up being one lower than the upper range value ($n-1$). Figure 3 displays the tabulated results of this situation. The plot in Figure 4 similarly presents a logarithmic scenario as the upper range value is increased.

1.5 Mathematical Formula on Eleven Tests

The previous mathematical formula certainly applies in this 11 test case but needs to be tweaked slightly. This is because the selected answer value now is no longer a power of 2 as can be see in Figure 3. All *true answers* are odd. Thus the revised equation is as follows: $f(n) = \lfloor log_2(n) \rfloor + 1$ where n is the value for testing provided by $n = 2^k - 1$. The floor operation is necessary since 2 does not factor completely into an odd integer. Two examples are given here:

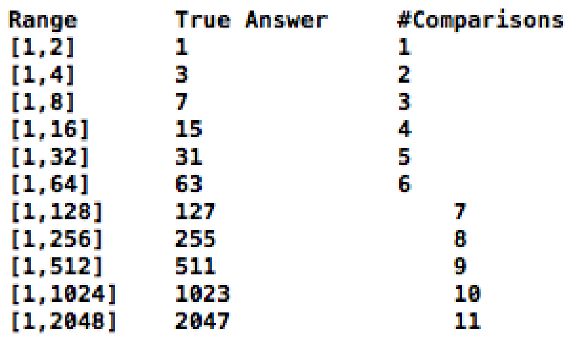
$$f(15) = \lfloor log_2(15) \rfloor + 1 = 4$$

$$f(511) = \lfloor log_2(511) \rfloor + 1 = 9$$

One can observe Figure 3 and see that the # of comparisons results in these values. The Big-0 notation for this expression is given as:

$$O(log(n))$$

The binary search algorithm peaks very quickly at the beginning and slowly saturates as the range is increased. This property is advantageous because the running time for extremely large data sets will be comparable to a data set a fraction of its size.



Range	True Answer	#Comparisons
[1,2]	1	1
[1,4]	3	2
[1,8]	7	3
[1,16]	15	4
[1,32]	31	5
[1,64]	63	6
[1,128]	127	7
[1,256]	255	8
[1,512]	511	9
[1,1024]	1023	10
[1,2048]	2047	11

Figure 3: Tabulated Data of Guessing Game

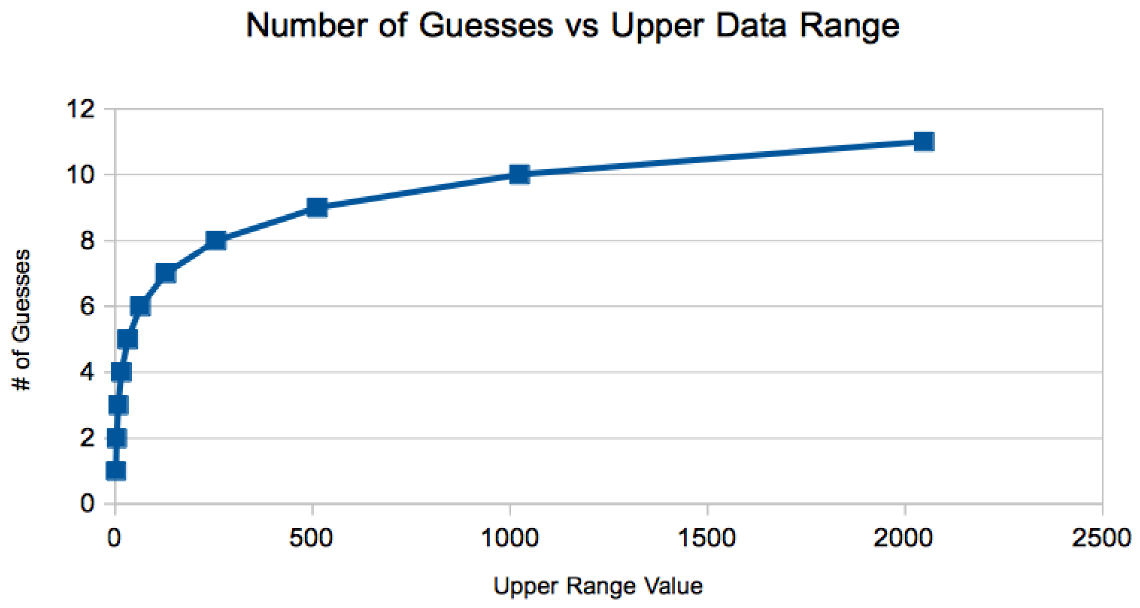


Figure 4: Number of Guesses vs Range

2 Matrix Operations

The second portion of the homework tasked the student with creating a matrix through two different implementations either the vector of vectors route or Stroustrup's Matrix class. The goal is to find a formula describing the number of additions required to compute the average of a *nxn* matrix. The vector of vectors route will be discussed first.

A 2D matrix was created by instantiating a vector of vectors like so:

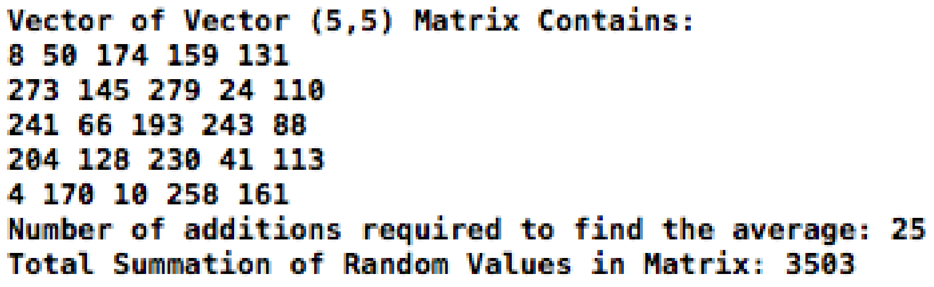
$$vector < vector < int >> vmat(n)$$

In the initialization, *vmat* is the name of the vector of vectors object while *n* specifies the vector container size. One then simply applies two for-loops to access the vector's indices to push elements into its container. The elements pushed onto the vector are selected randomly.

2.1 Vector of Vectors Summing and Number of Additions

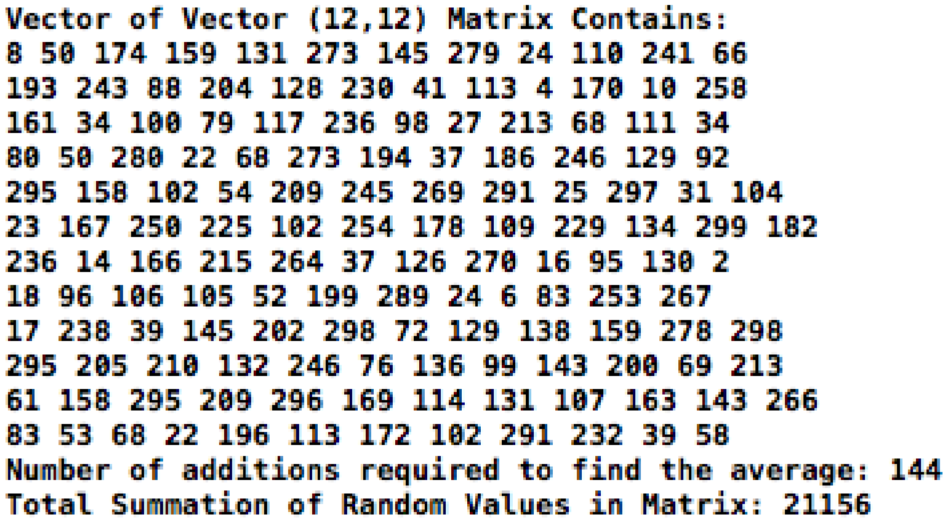
A few matrix examples are illustrated. Figure 5 displays a *(5x5)* matrix test case with its number of additions to find the average equivalent to 25 and its sum is 3503. Figure

6 displays a *(12,12)* matrix case with its additions amounting to 144 for computing an average and its summation is 21156.



```
Vector of Vector (5,5) Matrix Contains:
8 50 174 159 131
273 145 279 24 110
241 66 193 243 88
204 128 230 41 113
4 170 10 258 161
Number of additions required to find the average: 25
Total Summation of Random Values in Matrix: 3503
```

Figure 5: 5x5 Matrix

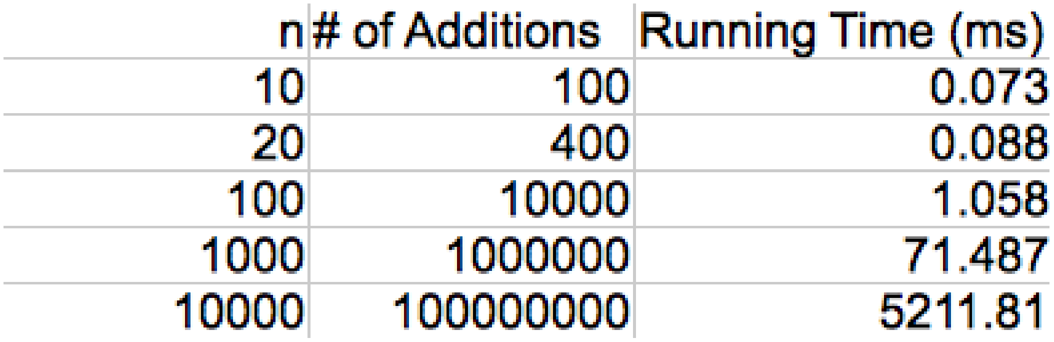


```
Vector of Vector (12,12) Matrix Contains:
8 50 174 159 131 273 145 279 24 110 241 66
193 243 88 204 128 230 41 113 4 170 10 258
161 34 100 79 117 236 98 27 213 68 111 34
80 50 280 22 68 273 194 37 186 246 129 92
295 158 102 54 209 245 269 291 25 297 31 104
23 167 250 225 102 254 178 109 229 134 299 182
236 14 166 215 264 37 126 270 16 95 130 2
18 96 106 105 52 199 289 24 6 83 253 267
17 238 39 145 202 298 72 129 138 159 278 298
295 205 210 132 246 76 136 99 143 200 69 213
61 158 295 209 296 169 114 131 107 163 143 266
83 53 68 22 196 113 172 102 291 232 39 58
Number of additions required to find the average: 144
Total Summation of Random Values in Matrix: 21156
```

Figure 6: 12x12 Matrix

2.2 Vector of Vectors Run-Time

Here the vector of vectors *(nxn)* matrix program was evaluated with $n = 10, 20, 100, 1000, 10000$ to compute its run-time for finding its summation and counting the number of additions per given n. Figure 7 illustrates the results. It is clear from the data that the number of additions required to compute the average of the values in the matrix increases exponentially as well as the running time as the *(nxn)* matrix size increases.



n	# of Additions	Running Time (ms)
10	100	0.073
20	400	0.088
100	10000	1.058
1000	1000000	71.487
10000	100000000	5211.81

Figure 7: NxN Matrix, Number of Additions for Average, and Run-Time

2.3 Vector of Vectors Formula

The formula relating matrix size n and the corresponding number of additions:

$$f(n) = n^2$$

Two examples using this equation:

$$f(20) = 20^2 = 400$$

$$f(100) = 100^2 = 10,000$$

The results from these equations can be verified from Figure 7.

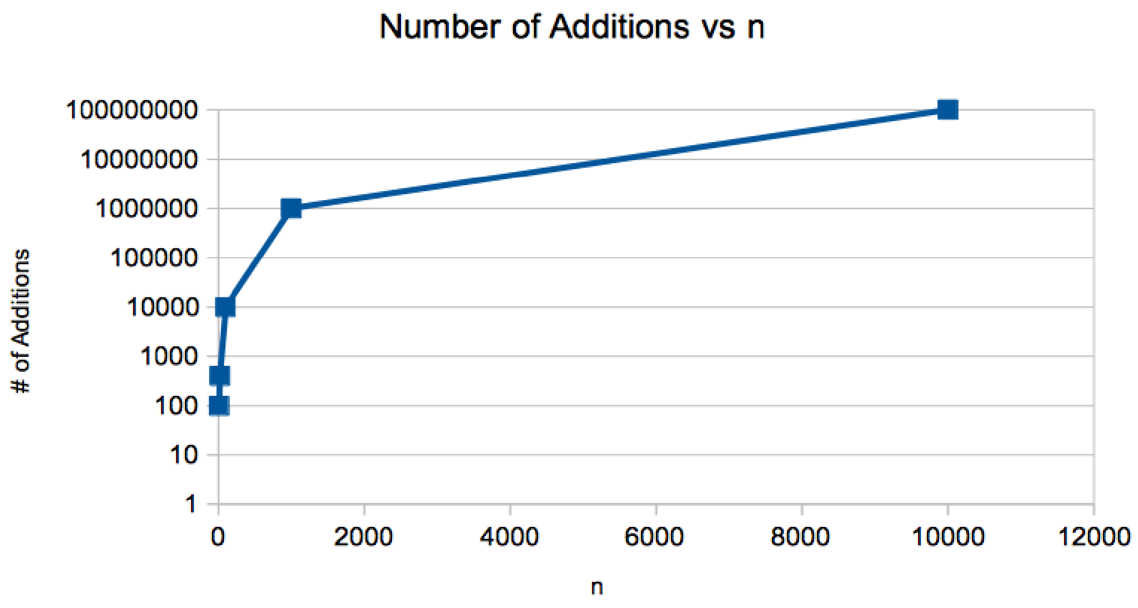


Figure 8: Number of Additions vs N

Figure 8 also shows that the # of additions to compute an average for an *(nxn)* matrix increases exponentially. The Big-O notation to relate # of additions and matrix size *(nxn)*:

$$O(n^2)$$

The n^2 term is the worst-case limiting factor for this matrix program.

2.4 Stroustrup's Matrix Class

Stroustrup's matrix class implementation is tested here. A 2D matrix is created by instantiating as follows:

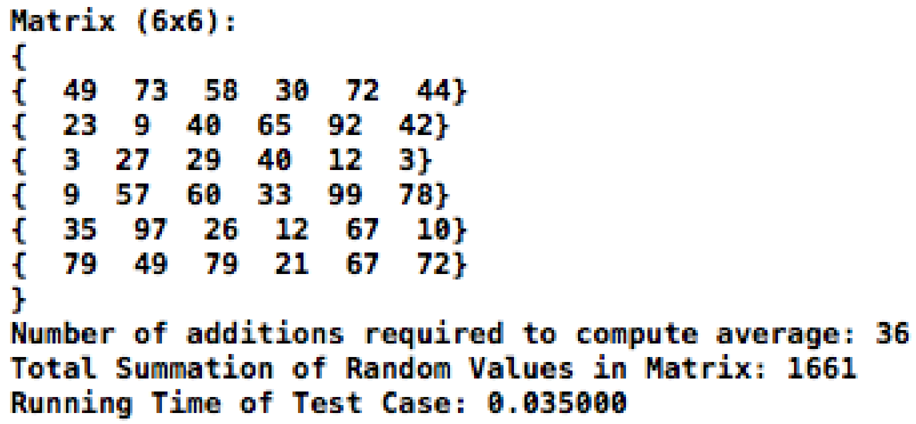
$$Numeric\_lib :: Matrix < int, 2 > mat(n, n)$$

$Numeric\_lib$ is the class to call to create a matrix object and to describe a 2D matrix of type int use this $< int, 2 >$. The name of the matrix object being created is mat while (n, n) is its size.

2.5 Matrix Summing and Number of Additions

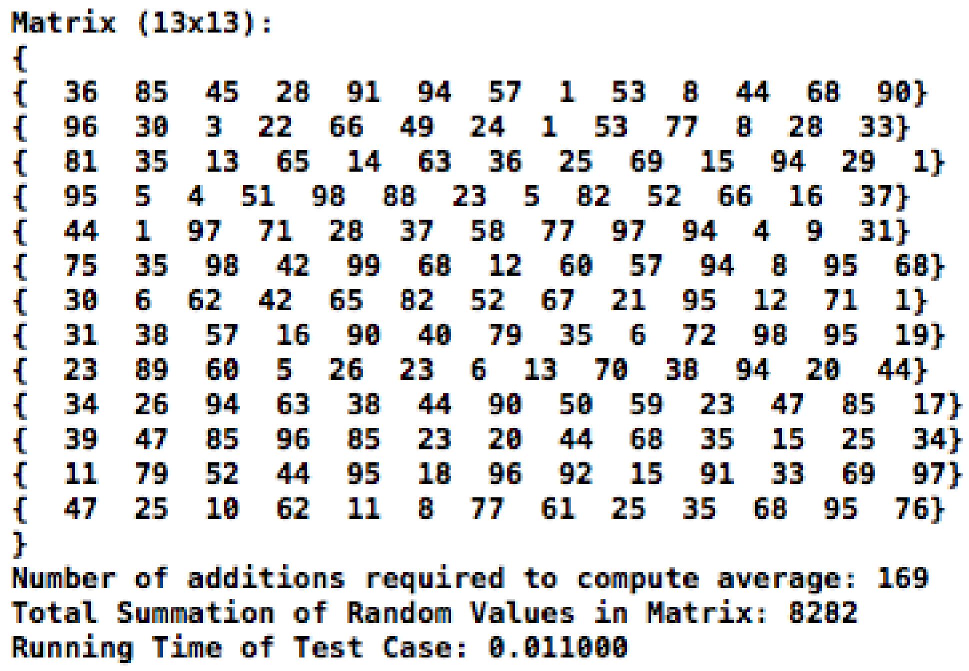
A few matrix test cases will be shown. Figure 9 displays a *(6x6)* matrix test case with its number of additions to find the average equivalent to 36 and its sum is 1661. Figure

10 displays a *(13,13)* matrix case with its additions amounting to 169 for computing an average and its summation is 8282.



```
Matrix (6x6):
{
{  49  73  58  30  72  44}
{  23   9  40  65  92  42}
{   3  27  29  40  12   3}
{   9  57  60  33  99  78}
{  35  97  26  12  67  10}
{  79  49  79  21  67  72}
}
Number of additions required to compute average: 36
Total Summation of Random Values in Matrix: 1661
Running Time of Test Case: 0.035000
```

Figure 9: 6x6 Matrix



```
Matrix (13x13):
{
{  36  85  45  28  91  94  57   1  53   8  44  68  90}
{  96  30   3  22  66  49  24   1  53  77   8  28  33}
{  81  35  13  65  14  63  36  25  69  15  94  29   1}
{  95   5   4  51  98  88  23   5  82  52  66  16  37}
{  44   1  97  71  28  37  58  77  97  94   4   9  31}
{  75  35  98  42  99  68  12  60  57  94   8  95  68}
{  30   6  62  42  65  82  52  67  21  95  12  71   1}
{  31  38  57  16  90  40  79  35   6  72  98  95  19}
{  23  89  60   5  26  23   6  13  70  38  94  20  44}
{  34  26  94  63  38  44  90  50  59  23  47  85  17}
{  39  47  85  96  85  23  20  44  68  35  15  25  34}
{  11  79  52  44  95  18  96  92  15  91  33  69  97}
{  47  25  10  62  11   8  77  61  25  35  68  95  76}
}
Number of additions required to compute average: 169
Total Summation of Random Values in Matrix: 8282
Running Time of Test Case: 0.011000
```

Figure 10: 13x13 Matrix

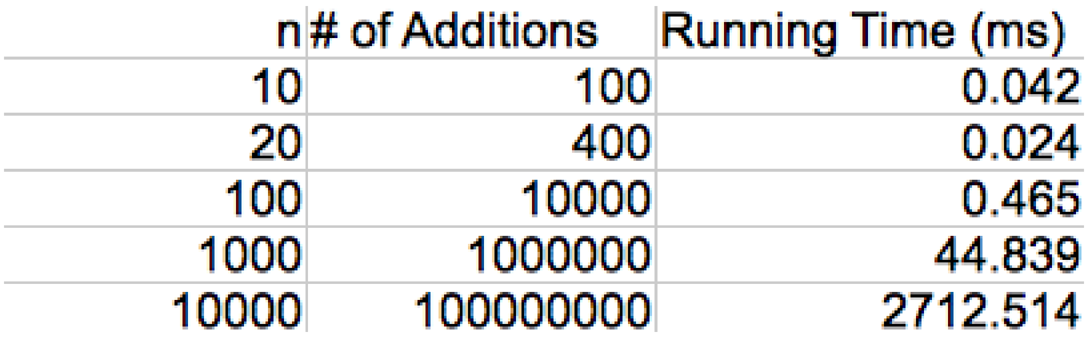
The formula to compute the # of additions follows the same equation as discovered previously: $f(n) = n^2$. Both cases follow the rule:

$$f(6) = 6^2 = 36$$

$$f(13) = 13^2 = 169$$

2.6 Matrix Additions and Run-Time

The *(nxn)* matrix program was evaluated with $n = 10, 20, 100, 1000, 10000$ as done for vector of vectors to compute its run-time for finding its summation and counting the number of additions per given n. Figure 11 displays the same exponentially increase in # of additions as discovered for vector of vectors. The run-time also increases dramatically with matrix size.



n	# of Additions	Running Time (ms)
10	100	0.042
20	400	0.024
100	10000	0.465
1000	1000000	44.839
10000	100000000	2712.514

Figure 11: NxN Matrix, Number of Additions for Average, and Run-Time