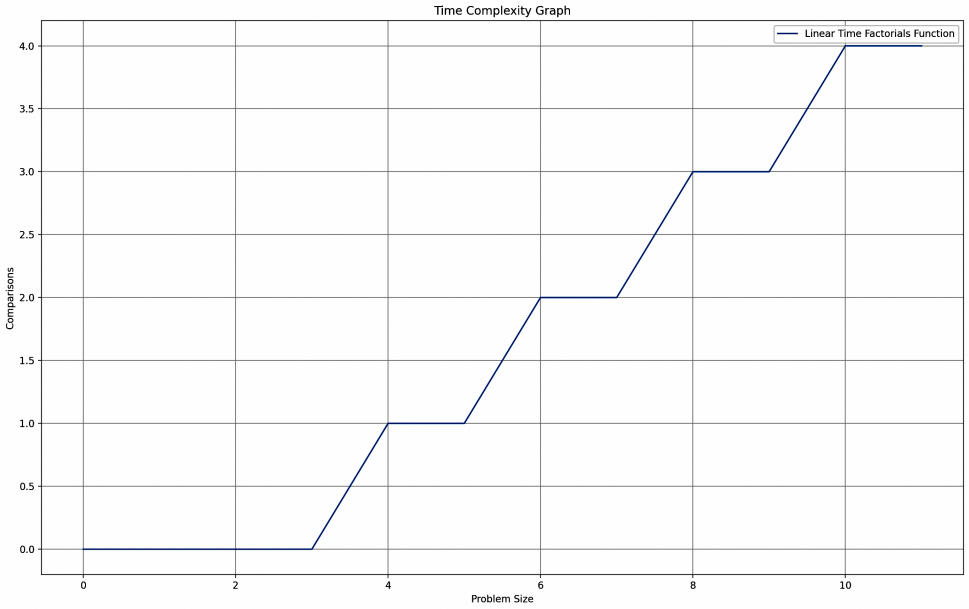
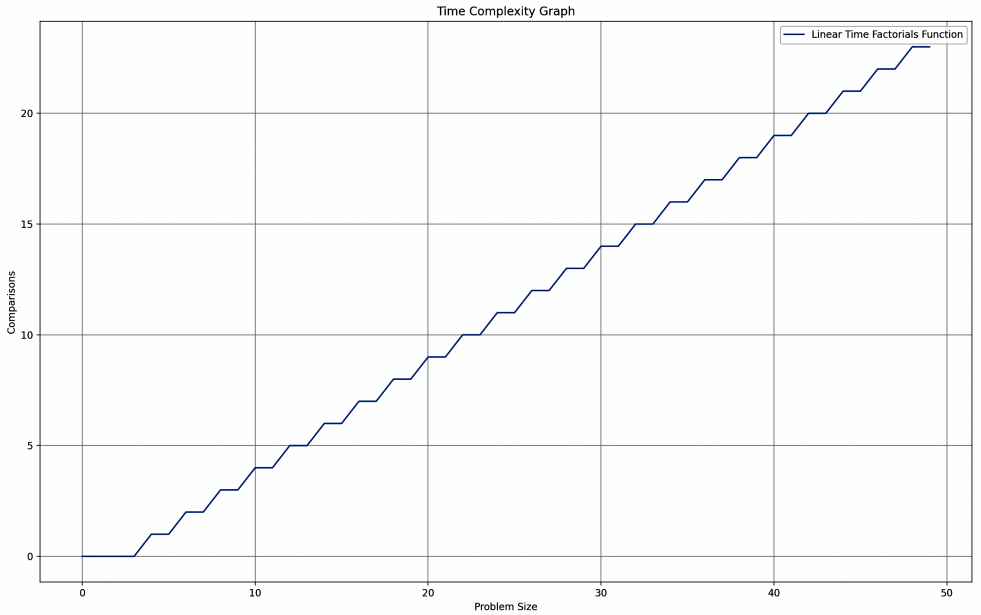
Factorials

An iterative function to calculate the factorials of a number, not including 1 and itself. This function is linear time, 0(n), due to the one for-loop being the dominate time factor in the function. The graph has a step-like appearance due to the way I set up the ‘base’ and ‘next’ variables, along with the conditional. These implementations create ‘factorial pairs’. For example, the number steps to find 4! and 5! are both equal to one, 6! and 7! both take two steps, 8! and 9! both take three steps, etc.





File: linear_time_factorials_function.txt

```
1   0 0 0
2   1 0 0
3   2 0 2
4   3 0 6
5   4 1 24
6   5 1 120
7   6 2 720
8   7 2 5040
9   8 3 40320
10  9 3 362880
11  10 4 3628800
12  11 4 39916800
```

Problem size // the argument passed into the function (int factorial). x-axis

Time complexity // how many steps taken to solve - y - axis

The actual data results of the function ('result' variable).

```cpp
int factorialFunction(int factorial)
{
    int result = 1;                    // variable to save the products
    int base = factorial;              // the current factorial
    int next = base - 1;               // the next number to multiply to the base factorial
    int n = 0;                         // time complexity variable
    if (factorial > 1){
        for (int i = 0; i < factorial; i++) // loop through each
        {
            result *= base * next;
            base -= 2;    // move to the next pair of numbers to multiply
            next -= 2;    // move to the next pair of numbers to multiply
            if (next < 1) // prevent multiplying by zero
            {
                break;
            }
            n++;
        }

        // save actual experiement data
        factorialsMap[n] = result;
        allExperimentResults["factorials"] = factorialsMap;
    }
    return n;
```

Max Element

A function to find the maximum element in a 2D array of integers using nested loops and a simple comparison check. The time complexity for this function is $O(n^2)$ since there are two nested for-loops. The loop simply checks the value at each position of the matrix and compares that value to the current recorded maximum. If the current value is greater than the recorded maximum, the current value becomes the new maximum. The way the matrix is filled means the final value will always be the maximum number, though the maximum being elsewhere would not change the time complexity of this function. The data in the output text file, along with the shape of the curve, confirms the quadratic nature of this function.

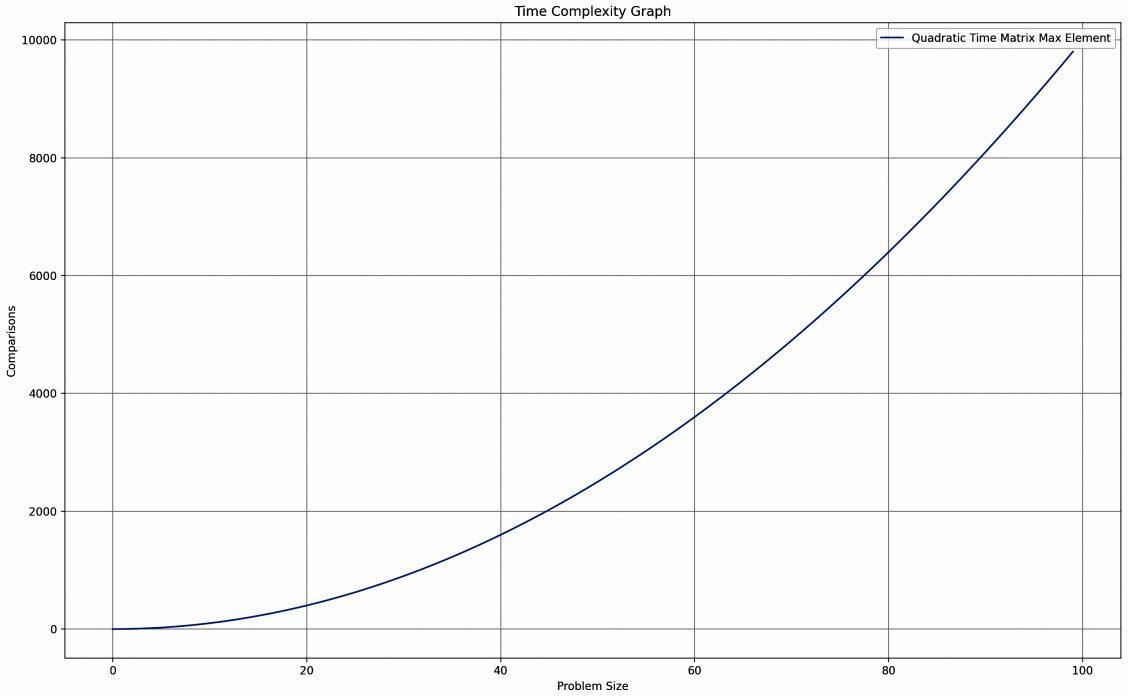
File: **quadratic_time_matrix_max_element.txt**

```
1     0 0 0
2     1 1 1
3     2 4 4
4     3 9 9
5     4 16 16
6     5 25 25
7     6 36 36
8     7 49 49
9     8 64 64
10    9 81 81
11    10 100 100
```

Problem size // the argument passed into the function (int factorial). x-axis

Time complexity // how many steps taken to solve - y - axis

Max element found each run. The max element always matches the number of times the loop ran.

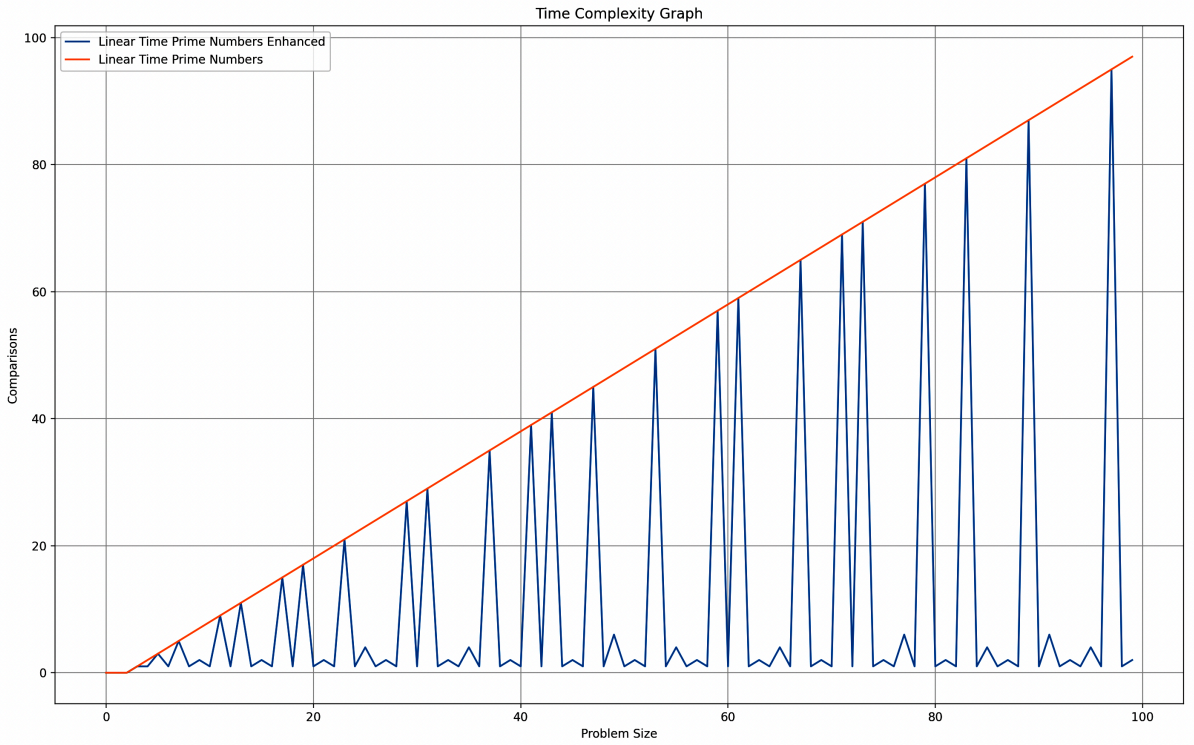



```
// find max element / calc time complexity (filling matrix not really necessary)
for (int i = 0; i < arraySize; i++)
{
    for (int j = 0; j < arraySize; j++)
    {
        currentInt = matrix[i][j];
        if (currentInt > max)
        {
            max = currentInt;
        }
        n++; // count inner iterations
    }
}
```

Prime Numbers

Functions to determine whether a number is prime. I have two functions, one that is linear time O(n) due to the one for-loop, and the other that is linear if the number is prime but has checks that will stop the loop if it is obvious that the number will not be prime (i.e. 'the square method' as well as checking to see if the prime candidate is divisible by the current number in the loop). In the 'enhanced' function, you can see that the high peaks are where primes are found and these match exactly the points on the standard function's linear line. Low points are where the function more quickly determined a number is not prime.

```cpp
int primeNumbersFunctionEnhanced(int primeNumberCandidate)
{
    bool prime = true;
    int n = 0; // time complexity variable
    if (primeNumberCandidate > 2)
    {
        // only makes sense to check numbers > 2
        // loop needs to start at 2 (1 and 0 have their own issues)
        // one for-loop = linear time
        for (int i = 2; i < primeNumberCandidate; i++)
        {
            n++;
            if (primeNumberCandidate % i == 0)
            {
                // end loop when any divisible number found
                prime = false;
                break;
            }

            if (i * i == primeNumberCandidate)
            {
                // square root strategy
                prime = false;
                break;
            }
        }
        // save actual experiement data
        primeNumbersMapEnhancedMap[n] = prime;
        allExperimentResults["primeEnhanced"] = primeNumbersMapEnhancedMap;
    }

    return n;
```



	File: **linear_time_prime_numbers_enhanced.txt**
1	0 0 0
2	1 0 0
3	2 0 0
4	3 1 1
5	4 1 0
6	5 3 1
7	6 1 0
8	7 5 1
9	8 1 0
10	9 2 0
11	10 1 0
12	11 9 1
13	12 1 0
14	13 11 1
15	14 1 0
16	15 2 0
17	16 1 0
18	17 15 1
19	18 1 0
20	19 17 1
21	20 1 0
22	21 2 0
23	22 1 0
24	23 21 1

	File: **linear_time_prime_numbers.txt**
1	0 0 0
2	1 0 0
3	2 0 0
4	3 1 1
5	4 2 0
6	5 3 1
7	6 4 0
8	7 5 1
9	8 6 0
10	9 7 0
11	10 8 0
12	11 9 1
13	12 10 0
14	13 11 1
15	14 12 0
16	15 13 0
17	16 14 0
18	17 15 1
19	18 16 0
20	19 17 1
21	20 18 0
22	21 19 0
23	22 20 0
24	23 21 1

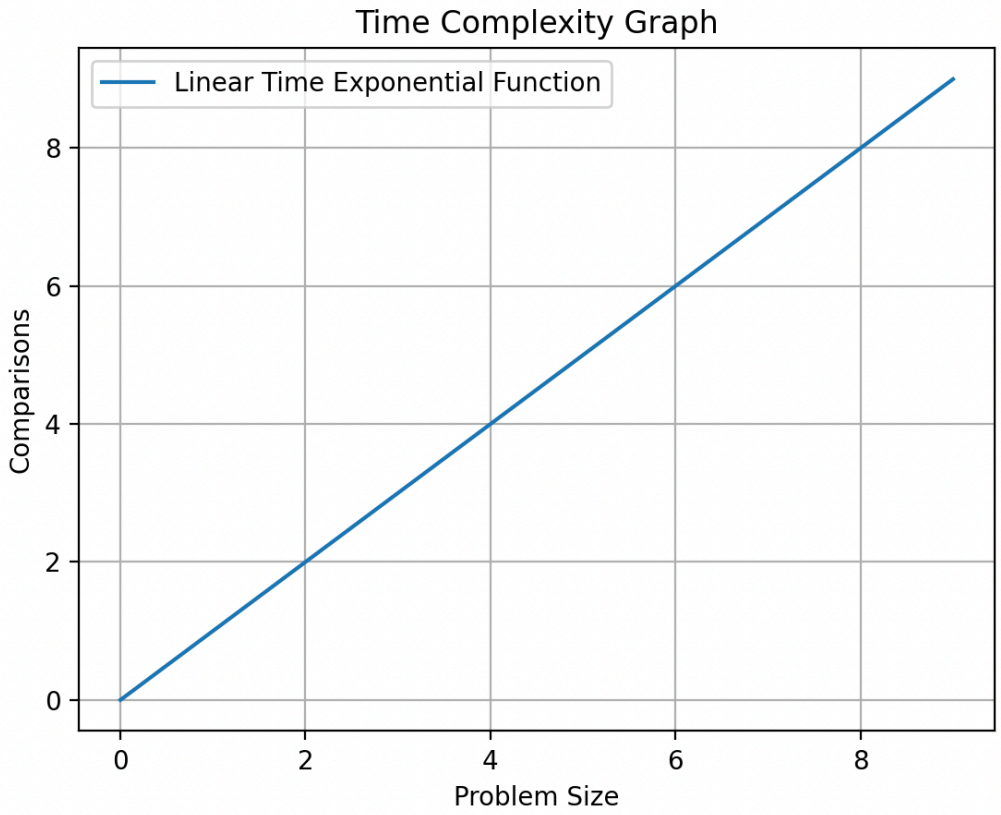
When no prime is found (bool '0' returned in 3rd column) 10 loops were required to determine '12' was not a prime with the standard function (right side data). With the enhanced loop, it took only one iteration of the loop to determine that '12' was not a prime (left side data). For '13', which is a prime number (bool 1 returned), both functions took the full iterations required.

Exponentials

Iterative function to calculate the result of raising a number to a power (e.g., x^n). Uses brute force iterative multiplication. This function has the time complexity O(n) since it uses one for-loop. The function takes in an integer ‘power’, sets this integer as the base to be raised, then raises that base ‘power’-times (e.g. if the user enters 4, the function will compute 4^4. So while the outputs are exponential (n^x), the calculation remains linear.

	File: **linear_time_exponential_function.txt**
1	0 0 1
2	1 1 1
3	2 2 4
4	3 3 27
5	4 4 256
6	5 5 3125
7	6 6 46656
8	7 7 823543
9	8 8 16777216
10	9 9 387420489

Column 1: base to be raised / power to raise the base
Column 2: iterations to complete calculation/ time complexity
Column 3: output from base^power

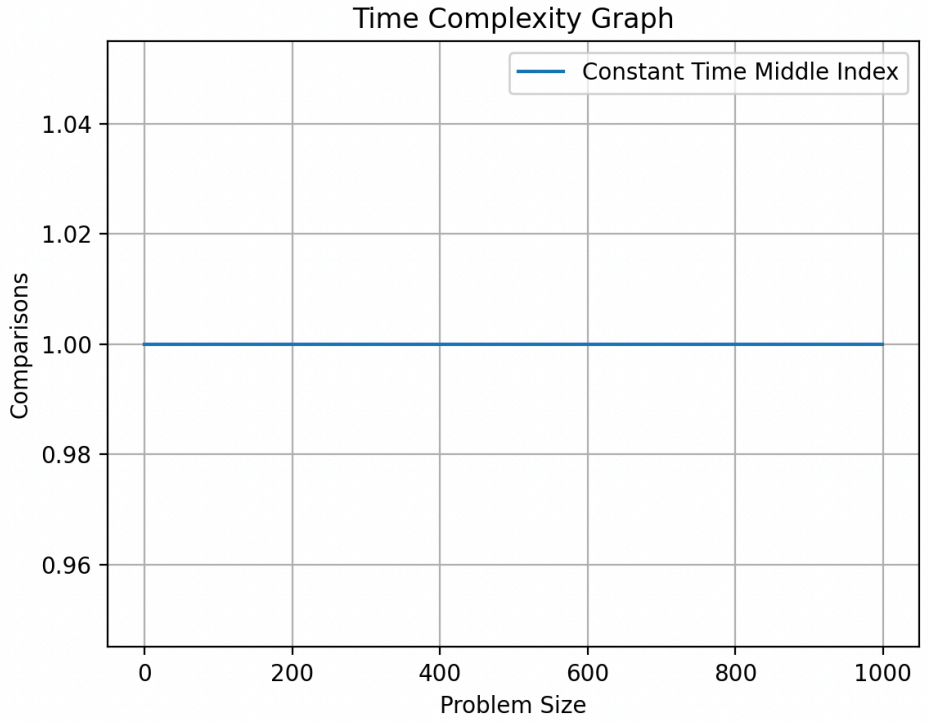



```
int exponentialFunction(int power)
{
    int n = 0; // time complexity variable
    int base = power; // base that will be raised
    int total = 1; // should you want to see that the function works correctly
    if (power == 0) // any num raised to 0 = 1
    {
        total = 1;
    }
    else
    {
        // one for-loop, so time complexity = O(n)
        for (int i = 0; i < power; i++)
        {
            n++; // linear time complexity
            total *= base;
        }
    }
    // save actual experiement data
    exponentialFunctionMap[n] =  total;
    allExperimentResults["exponential"] = exponentialFunctionMap;
    // return time complexity
    return n;
}
```

Middle Index

Function to retrieve the element at the middle index of any sized (dynamic) array. This function has the time complexity of constant since indexing a contiguous array is essentially instantaneous. The function takes in an integer, sets the dynamic array's size to this integer's value, fills the array with ascending values, then finds the value of the element in the middle of the array.

```cpp
/**
 * CONSTANT TIME - MIDDLE ARRAY ELEMENT
 * @return time complexity of middle indexing.
 */
int middleIndexing(int arraySize)
{
    int c = 0;                          // track how many const operations are performed
    int* array = new int[arraySize];    // make array

    for (int i = 0; i < arraySize; i++) // fill array
    {
        array[i] = i;
    }
    int midElement = array[arraySize/2]; // get the middle element (constant time operation)
    c++; // increment constant operation variable

    // clean up memory
    delete [] array;

    // save actual experiement data
    middleIndexingMap[c] = midElement;
    allExperimentResults["middle"] =  middleIndexingMap;

    return c;
}
```



File: **constant_time_middle_index.txt**

```
0 1 0
1 1 0
2 1 1
3 1 1
4 1 2
5 1 2
6 1 3
7 1 3
8 1 4
9 1 4
10 1 5
11 1 5
12 1 6
13 1 6
14 1 7
15 1 7
16 1 8
17 1 8
18 1 9
19 1 9
20 1 10
21 1 10
22 1 11
23 1 11
24 1 12
25 1 12
26 1 13
27 1 13
28 1 14
29 1 14
30 1 15
31 1 15
32 1 16
33 1 16
34 1 17
35 1 17
36 1 18
37 1 18
38 1 19
39 1 19
40 1 20
```

Column 1: array size
Column 2: iterations to complete calculation - constant at 1
Column 3: value of middle index (~ half array size)

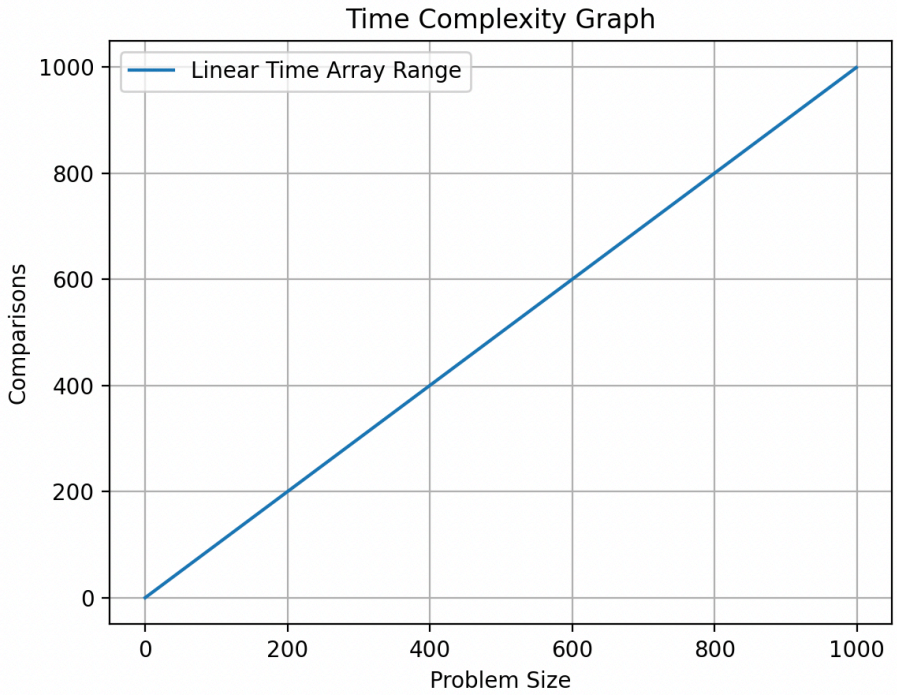
Array Range

Function to calculate the range of all the elements in a single (dynamic) array. This function has the time complexity of O(n), linear, since it requires one iteration of a for-loop over the array. The function takes in an integer, sets the dynamic array's size to this integer's value, fills the array with ascending values, finds the min and max elements, and returns the difference.

File: **linear_time_array_range.txt**

```
0 0 0
1 1 0
2 2 1
3 3 2
4 4 3
5 5 4
6 6 5
7 7 6
8 8 7
9 9 8
10 10 9
11 11 10
12 12 11
13 13 12
14 14 13
15 15 14
16 15 15
17 16 16
18 17 17
19 18 18
20 19 19
21 20 20
22 21 21
23 22 22
24 23 23
25 24 24
26 25 24
27 26 25
28 27 26
29 28 27
30 29 28
31 30 29
32 31 30
33 32 31
34 33 32
35 34 33
36 35 34
37 36 35
38 37 36
39 38 37
40 39 38
41 40 39
```

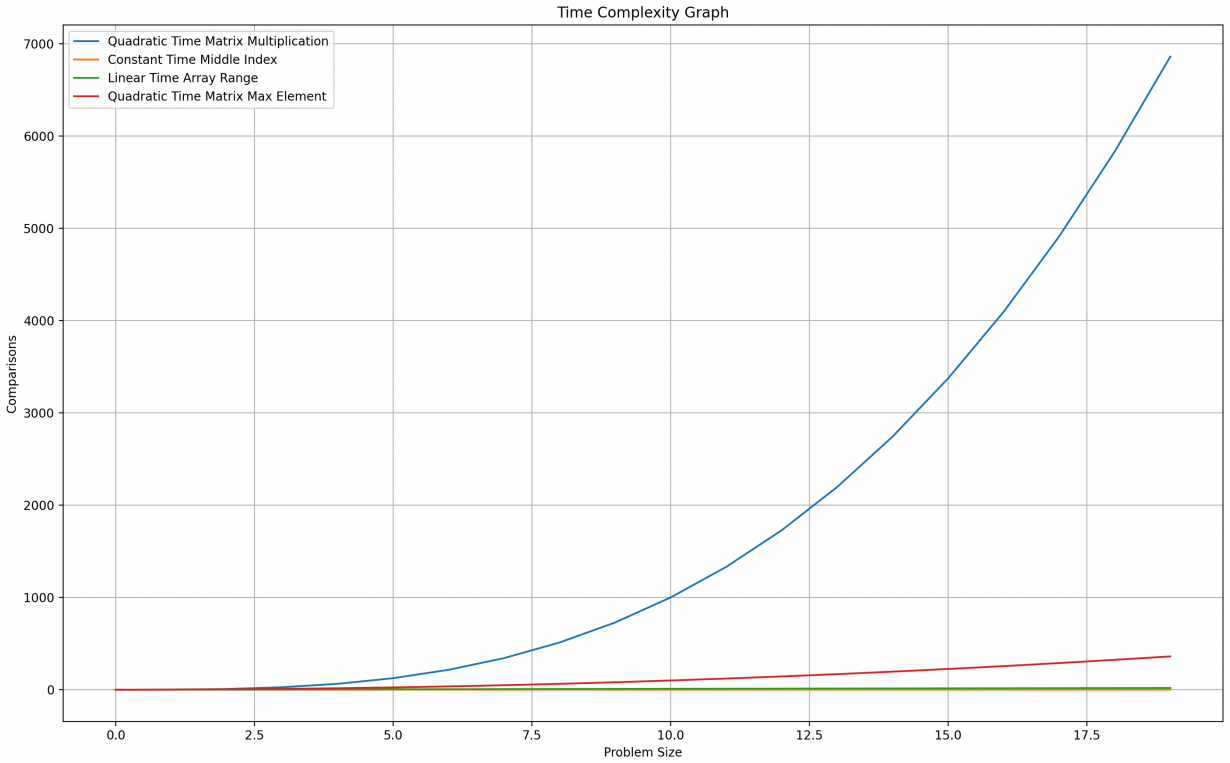
Column 1: array size
Column 2: iterations to complete calculation - linear, equals array size
Column 3: value of range, also equal to array size




```
int arrayRange(int arraySize)
{
    int n = 0;                          // time complexity variable
    int* array = new int[arraySize];    // initialize array
    int min = 0;                        // min number in array to calc range
    int max = 0;                        // max number in array to calc range
    int currentInt;                     // for use in loop / determining min & max
    int range;                          // final result but not actually needed for this project

    for (int i = 0; i < arraySize; i++)
    {
        array[i] = i;
    }
    // find min and max
    for (int i = 0; i < arraySize; i++)
    {
        currentInt = array[i];
        if (currentInt <= min)
        {
            min = currentInt;
        }
        if (currentInt > max)
        {
            max = currentInt;
        }
        n++;                            // each iteration increases time complexity
    }
     range = max - min;

    // clean up memory
    delete[] array;

    // save actual experiement data
    arrayRangeMap[n] = range;
    allExperimentResults["range"] =  arrayRangeMap;

    return n;
}
```

Matrix Multiplication

Function that employs dot product multiplication of square (dynamic) matrices using the naive approach with for-loops. Since this function employs three nested for-loops, the time complexity is O(n^3). The first two loops set the numbers to be multiplied from each of the matrices (matrixA and matrixB), and the third loop places the product of those values into the third matrix (matrixC). I flatten out matrixC to a single sum of all the products for the data output of the function. To make sure the function was implemented correctly, I set up a visualization of each matrix in the terminal as well.




```
matrix A
0    1    2    3    4    5    6    7    8
1    2    3    4    5    6    7    8    9
2    3    4    5    6    7    8    9    10
3    4    5    6    7    8    9    10   11
4    5    6    7    8    9    10   11   12
5    6    7    8    9    10   11   12   13
6    7    8    9    10   11   12   13   14
7    8    9    10   11   12   13   14   15
8    9    10   11   12   13   14   15   16

matrix B
0    1    2    3    4    5    6    7    8
1    2    3    4    5    6    7    8    9
2    3    4    5    6    7    8    9    10
3    4    5    6    7    8    9    10   11
4    5    6    7    8    9    10   11   12
5    6    7    8    9    10   11   12   13
6    7    8    9    10   11   12   13   14
7    8    9    10   11   12   13   14   15
8    9    10   11   12   13   14   15   16

matrix C
0    1    4    9    16   25   36   49   64
1    4    9    16   25   36   49   64   81
4    9    16   25   36   49   64   81   100
9    16   25   36   49   64   81   100  121
16   25   36   49   64   81   100  121  144
25   36   49   64   81   100  121  144  169
36   49   64   81   100  121  144  169  196
49   64   81   100  121  144  169  196  225
64   81   100  121  144  169  196  225  256

sum of products from Matrix C: 6264
```

Visualization of O(n^3) vs O(n^2) vs linear, vs constant over just 20 trials along with visualization of the three matrices in terminal.

File: quadratic_time_matrix_multiplication.txt

```
1    0 0 0
2    1 1 0
3    2 8 6
4    3 27 48
5    4 64 184
6    5 125 500
7    6 216 1110
8    7 343 2156
9    8 512 3808
10   9 729 6264
```

Column 1: 2d array size (length of one side)
Column 2: cubic iterations
Column 3: value of the sum of all the dot products