Problem 5.1 *Fibonacci numbers*
(a)**for full implementation, see fibonacci.h**
Naive form:

```
unsigned long long int naiveFibonacci(unsigned long long n){
        if (n < 2)
                return n;
        else
                return naiveFibonacci(n-1) + naiveFibonacci(n-2);
}
```

Closed form:

```
unsigned long long int closedFormFibonacci(int n){
        long double phi =(long double) ( ( long double) (1  + sqrtl(5))) / (long double) 2;
        long double power = powl(phi, n);
        long double root = sqrtl((long double)5);
        long double res = power / root;
        unsigned long long int actual_res  = llround(res);
        return actual_res;
}
```

Bottom up form: //uses an array to stop potential 32-bit caching in cpu

```
unsigned long long int bottomUpFibonacci(long int n){
        unsigned long long int numbers[n];
        if( n < 2)
                return n;
        numbers[0] = 1;
        numbers[1] = 1;
        for (int i = 2; i < n; i++) {
                numbers[i] = numbers[i - 1] + numbers[i - 2];
                //cout << numbers[i] << endl;
        }
        return numbers[n-1];
}
```

Matrix form:

```
template <class T>
class Matrix2x2;

template <class T>
void print_matrix(const Matrix2x2<T>& a);
template <class T>
class Matrix2x2{
private:
        T** values;
public:
        inline T* operator[](int a) const{
                return values[a];
        }

        Matrix2x2(const T arr[2][2]){
                values = new T*[2];
                for (int i = 0; i < 2; i++)
```

```
                values[i] = new T[2];
        values[0][0] = arr[0][0];
        values[0][1] = arr[0][1];
        values[1][0] = arr[1][0];
        values[1][1] = arr[1][1];

    }
    Matrix2x2(const Matrix2x2& cpy){
        values = new T*[2];
        for (int i = 0; i < 2; i++)
            values[i] = new T[2];
        values[0][0] = cpy.values[0][0];
        values[0][1] = cpy.values[0][1];
        values[1][0] = cpy.values[1][0];
        values[1][1] = cpy.values[1][1];
    }

    Matrix2x2(T** A){
        values = A;
    }

    friend Matrix2x2 operator*(const Matrix2x2<T>& a, const Matrix2x2<T>& b) {
        /*cout<<"multiplying: \n";
        print_matrix(a);
        cout<<"and : \n";
        print_matrix(b);
        cout<<"\n";
         */
        T **numbers = new T*[2];
        if (!numbers)
            throw std::bad_alloc();
        for (int i = 0; i < 2; i++) {
            numbers[i] = new T[2];
            if (!numbers[i]) throw std::bad_alloc();
        }
        T M1 = (a.values[0][0] + a.values[1][1]) * (b.values[0][0] + b.values[1][1]);
        T M2 = (a.values[1][0] + a.values[1][1]) * (b.values[0][0]);
        T M3 = a.values[0][0] * (b.values[0][1] - b.values[1][1]);
        T M4 = a.values[1][1] * (b.values[1][0] - b.values[0][0]);
        T M5 = (a.values[0][0] + a.values[0][1]) * b.values[1][1];
        T M6 = (a.values[1][0] - a.values[0][0]) * (b.values[0][0] + b.values[0][1]);
        T M7 = (a.values[0][1] - a.values[1][1]) * (b.values[1][0] + b.values[1][1]);

        numbers[0][0] = M1 + M4 - M5 + M7;
        numbers[0][1] = M3 + M5;
        numbers[1][0] = M2 + M4;
        numbers[1][1] = M1 - M2 + M3 + M6;
        Matrix2x2 res(numbers);


        return res;
    }
```

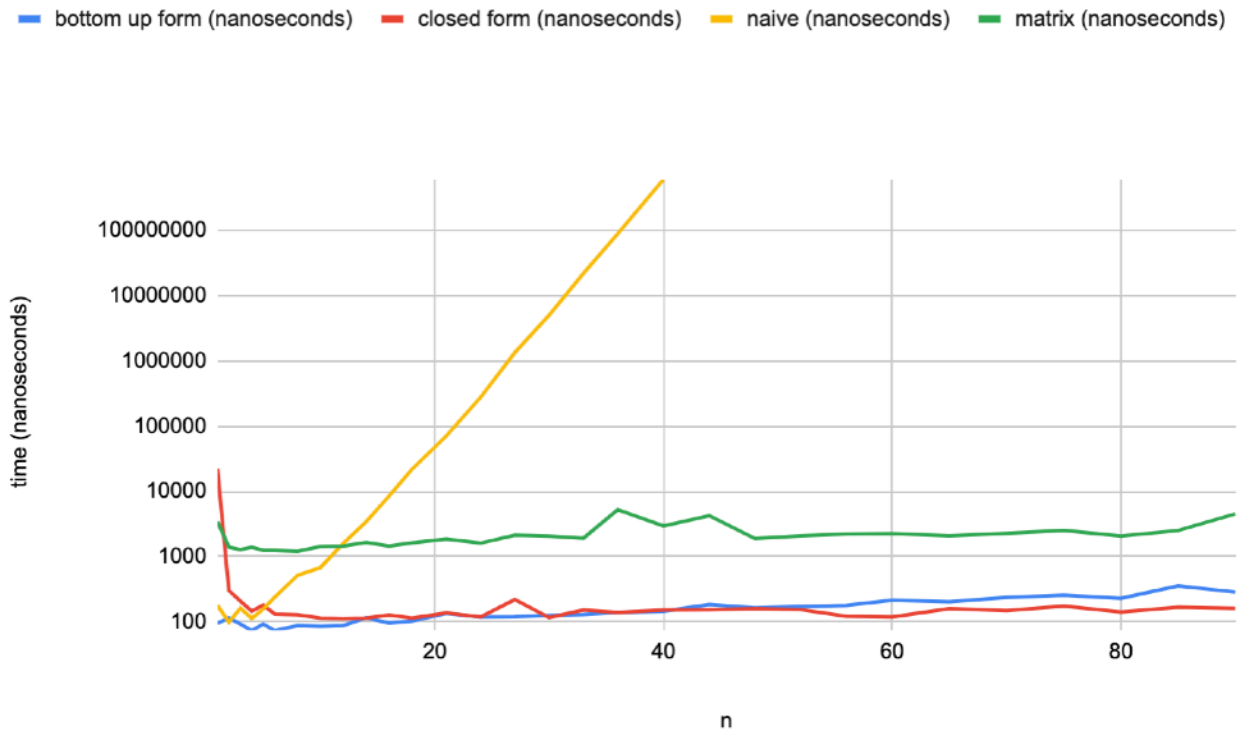
```
        Matrix2x2<T> operator^(const int& n){
                if (n == 0){
                        T numbers[2][2];
                        numbers[0][0] = 1;
                        numbers[0][1] = 0;
                        numbers[1] [0] = 0;
                        numbers[1][1] = 1;
                        Matrix2x2<T> res(numbers);
                        return res;
                }
                if (n == 1)
                        return *this;
                if (n % 2 == 0) {
                        Matrix2x2<T> res(*this^(n/2));
                        return res * res;
                }
                else {
                        Matrix2x2<T> res((*this ^ ((n - 1) / 2)));
                        return res * res * (*this);
                }
        }
        ~Matrix2x2(){
                //cout<<"freeing:"<<endl; print_matrix(*this);
                delete [] values;
        }
};
template <class T>
void print_matrix(const Matrix2x2<T>& a){
        for (int i = 0; i < 2; i++) {
                for (int j = 0; j < 2; j++) {
                        std::cout << a[i][j] << " ";
                }
                std::cout<<std::endl;
        }
}
//actual  function :)
unsigned long long int matrixFibonacci(int n){
        unsigned long long int** fib = new unsigned long long int*[2];
        if (!fib)
                throw std::bad_alloc();
        fib[0] = new unsigned long long int[2];
        if (!fib[0])
                throw std::bad_alloc();
        fib[1] = new unsigned long long int[2];
        if (!fib[1]) throw std::bad_alloc();
        fib[0][0] = 1;
        fib[0][1] = 1;
        fib[1][0] = 1;
        fib[1][1] = 0;
        Matrix2x2<unsigned long long int> mat(fib);
        Matrix2x2<unsigned long long int> result(mat^n);
        unsigned long long int res = result[0][1];
```

```
        return res;
}
```

(b) the following results were obtained using a 2 second maximum time computation limit:

n	bottom up form (nanoseconds)	closed form (nanoseconds)	naive (nanoseconds)	matrix (nanoseconds)
1	94	22411	177	3431
2	114	300	98	1391
3	92	207	161	1267
4	73	145	112	1387
5	92	179	156	1244
6	73	131	237	1240
8	87	128	511	1202
10	85	113	674	1423
12	87	111	1582	1449
14	115	113	3391	1643
16	95	127	8450	1444
18	102	113	21754	1615
21	134	136	70474	1847
24	118	118	275085	1598
27	119	218	1.35E+06	2116
30	125	115	5.05E+06	2051
33	129	151	2.20E+07	1903
36	138	138	8.97E+07	5232
40	144	151	6.10E+08	2927
44	183	152		4227
48	162	158		1885
52	171	153		2069
56	178	121		2217
60	214	118		2240
65	202	158		2073
70	235	147		2259
75	255	173		2516
80	228	141		2060
85	355	167		2502
90	284	160		4504

(c) for the same n, not all of the methods may return the same answer. Assuming that integer overflow does now occur, the matrix form, naive recursive, and bottom-up algorithms should return the same integer value. However, the closed form solution may return a different value due to rounding errors arising from its implementation: the closed form implementation uses irrational numbers, which cannot be precisely stored in double or long double data types. This means that errors may propagate and lead to an incorrect result, especially for large n.



(d) As it can be seen on the graph above, the closed form algorithm took the least amount of time on average to produce a fibonacci number, however, the trade-off is that it's results may not be correct due to double rounding imprecision. The bottom-up method was the second most effective method for this data set, but towards the end it can be seen that its linear complexity increases the difference between it and the closed form, Meanwhile, the matrix form, though

taking comparatively longer than the bottom-up form for any n, should scale better with time due to its $\Theta(logn)$ time complexity. finally, the naive approach takes exponential time and is completely outclasses by all other algorithms in terms of computation time.

Problem 5.2

(a) Derive the asymptotic time complexity for a brute force implementation of n x n sized integers: Consider the brute force multiplication algorithm taught at school:

```
  123   (m)
  111   (n)
  ——
  123
 1230
12300
```

For every integer at the bottom, we multiply it times the number on top, and shift it to the left depending on what position the integer at the bottom is at. We can do the exact same thing using a binary system. The resulting time complexity of this algorithm is $\Theta(n^2)$ because it performs $n \times m$ multiplications when considering the length of both m and n (for every bit in n we multiply it fully times m). Since $n = m$, this boils down to n^2

(b) The idea for this algorithm will be shown using the decimal system, but this algorithm works for any number system:

Consider two numbers a, b that are multiplied together. For example, consider $a = 6920$ and $b = 4202$. These numbers can be represented in the following way:

$a = 69 \times 10^2 + 20; b = 42 \times 10^2 + 2$. Or, in more general terms, we can express these numbers as :

$a = a_1 \times 10^{n/2} + a_2$

$b = b_1 \times 10^{n/2} + b_2$

The product of these terms is:

$a_1b_1 \times 10^n + 10^{n/2}(a_1b_2 + b_1a_2) + a_2b_2$

The term: $a_1b_2 + b_1a_2$ can be rewritten as: $(a_1 + a_2)(b_1 + b_2) - a_1b_1 - a_2b_2$

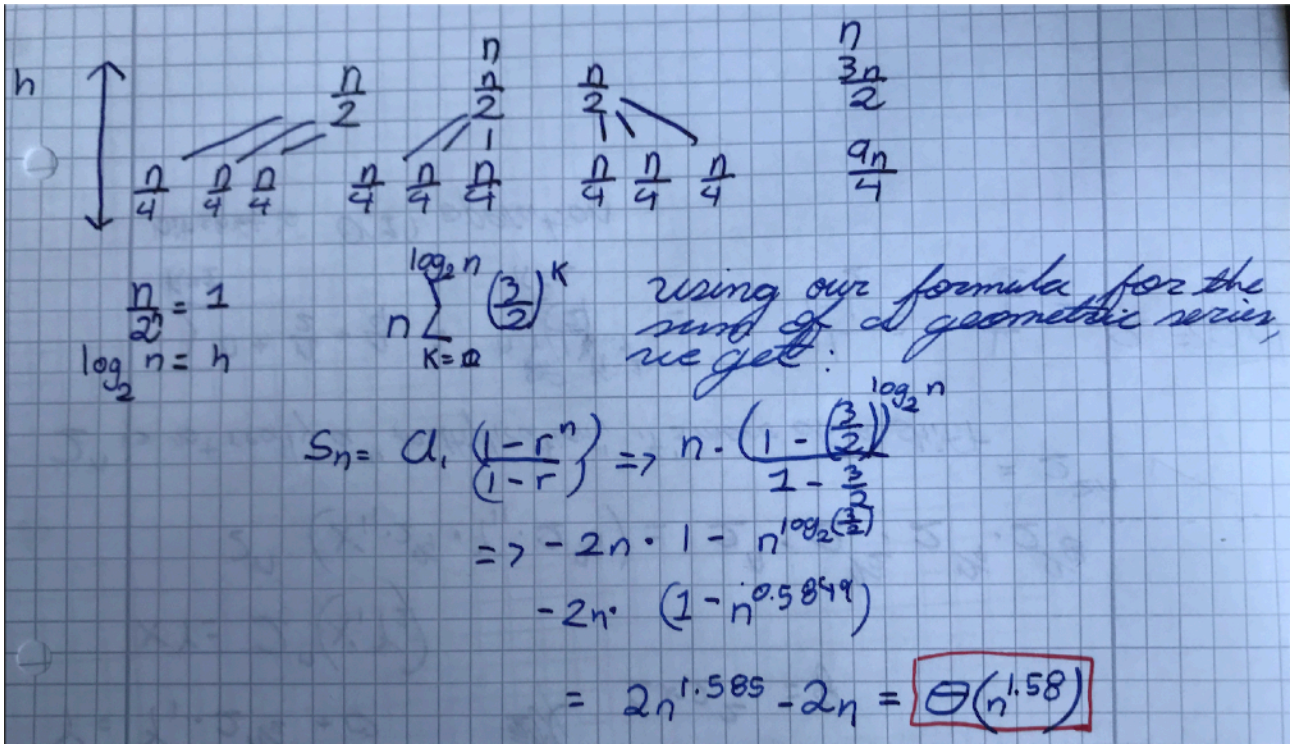
finally, we get that:

$a \times b = a_1b_1 \times 10^n + 10^{n/2}((a_1 + a_2)(b_1 + b_2) - a_1b_1 - a_2b_2) + a_2b_2$. This results in 3 new multiplications: $a_1b_1, a_2b_2,$ and $(a_1 + a_2)(b_1 + b_2)$, each using n/2-sized numbers: furthermore, we take summation to be $\Theta(n)$ because each bit must be compared to sum the two numbers. (Note that the cost of multiplying to the power of 2 (the base) is the equivalent to binary shifting, which usually has a complexity of $\Theta(1)$ and at most a complexity of $\Theta(n)$ for modern cpu architecture.) Therefore: the recurrence becomes

(c)

$T(n) = 3T(n/2) + \Theta(n)$

(d)



(e) using the Master theorem:

$n = O(n^{log_2 3-\epsilon}) = O(n^{1.58-0.58})$
As shown by the first case of the master theorem, then
$T(n) = \Theta(n^{log_2 3})$