1. SEE HANDWRITTEN PAPER

2. **Explanation:** The recursive and iterative methods matched the pseudocode presented in lecture. For the matrix method, I implemented repeated squaring and stored the squared matrices of interest in an array and multiplied those which were needed to reach the nth Fibonacci number. The recursive method, naturally, slowed rapidly. But it's worth noting that the iterative method performed better than the matrix method until we reached a high threshold (also, the increased frequency of mod operations may have slowed the matrix method).

   My Benchmarked Results:

n	**Recursive**	**Iterative**	**Matrix**
2	0.000001	0.000001	0.000011
10	0.000003	0.000001	0.000017
30	0.021920	0.000001	0.000012
40	1.973016	0.000003	0.000011
50	240.048973	0.000001	0.000013
1000	Too Long	0.000027	0.000013
10000	Too Long	0.000272	0.000017
50000	Too Long	0.001136	0.000011

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <stdint.h>
#include <vector>
#include <cmath>

//fib1

int fib1(int n) {
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        return (fib1(n-1)% 65536+fib1(n-2)% 65536)% 65536;
    }
}


//fib2

int fib2(int n) {
    int fibarray[n+1];
```

```
    fibarray[0] = 0;
    fibarray[1] = 1;

    for (int i = 2; i <= n; i++) {
        fibarray[i] = (fibarray[i-1]% 65536 + fibarray[i-2]% 65536)% 65536;
    }
    return fibarray[n];
}

//fib3

struct matrix_holder
{
    int matrix[2][2];
};

// ============================================
// Implementation of naive matrix squaring
// resulting = mat1^2
// use pass by reference
// ============================================

void naivematrixsquaring(matrix_holder input, matrix_holder &result) {
    result.matrix[0][0] = (input.matrix[0][0] % 65536 *input.matrix[0][0] % 65536 + input
    result.matrix[0][1] = (input.matrix[0][0] % 65536 *input.matrix[0][1] % 65536 + input
    result.matrix[1][0] = (input.matrix[1][0] % 65536 *input.matrix[0][0] % 65536 + input
    result.matrix[1][1] = (input.matrix[1][0] % 65536 *input.matrix[0][1] % 65536 + input
}

void naivematrixmult(matrix_holder input1, matrix_holder input2, matrix_holder &result) {
    result.matrix[0][0] = (input1.matrix[0][0]% 65536 *input2.matrix[0][0]% 65536 + input
    result.matrix[0][1] = (input1.matrix[0][0]% 65536 *input2.matrix[0][1]% 65536 + input
    result.matrix[1][0] = (input1.matrix[1][0]% 65536 *input2.matrix[0][0]% 65536 + input
    result.matrix[1][1] = (input1.matrix[1][0]% 65536 *input2.matrix[0][1]% 65536 + input
}


int fib3(int n) {

    // don't want to deal with these base cases

    if (n == 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
    }
```

```
    int arraysize = floor(log2(n)+2);

    matrix_holder base_matrix;
    matrix_holder final_matrix;
    matrix_holder identity_matrix;
    matrix_holder intermediary_matrix;

    // initialize matrices
    for  (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            base_matrix.matrix[i][j] = 0;
            final_matrix.matrix[i][j] = 0;
            identity_matrix.matrix[i][j] = 0;
        }
    }

    // fix the base matrix such that it is [0 1; 1 1]
    base_matrix.matrix[0][0] = 0;
    base_matrix.matrix[0][1] = 1;
    base_matrix.matrix[1][0] = 1;
    base_matrix.matrix[1][1] = 1;
    final_matrix.matrix[0][0] = 1;
    final_matrix.matrix[1][1] = 1;
    identity_matrix.matrix[0][0] = 1;
    identity_matrix.matrix[1][1] = 1;



    matrix_holder repeated_squares_arrray[arraysize];
    repeated_squares_arrray[0] = identity_matrix; // i.e., the identity (at this point)
    repeated_squares_arrray[1] = base_matrix;

    // populate all relevant repeating squares matrices
    for (int i = 2; i < arraysize; i++) {
        naivematrixsquaring(repeated_squares_arrray[i-1], repeated_squares_arrray[i]);
    }

    for (int i = 1; i < arraysize+1; i++) {
        // and with one and then bit shift for comparison and determining whether we use
        if ((n & 1) == 1) {
            naivematrixmult(final_matrix, repeated_squares_arrray[i], intermediary_matrix
            final_matrix = intermediary_matrix;
        }
        n = n >> 1;
    }
    return final_matrix.matrix[0][1]% 65536;// % 65536;
}
```

```
int main(int argc, char const *argv[])
{
    clock_t fib1_start_time, fib1_end_time, fib2_start_time, fib2_end_time, fib3_start_ti
    double fib1_cpu_time, fib2_cpu_time, fib3_cpu_time;

    int n = 50000;

    fib3_start_time = clock();
    int fib3ret = fib3(n);
    fib3_end_time = clock();
    fib3_cpu_time = ( (double) (fib3_end_time - fib3_start_time) ) / (double) CLOCKS_PER_
    printf("FIB3, %lf\n Fib3 returns: %i\n", fib3_cpu_time, fib3ret);

    fib2_start_time = clock();
    int fib2ret = fib2(n);
    fib2_end_time = clock();
    fib2_cpu_time = ( (double) (fib2_end_time - fib2_start_time) ) / (double) CLOCKS_PER_
    printf("FIB2, %lf\n Fib2 returns: %i\n", fib2_cpu_time, fib2ret);

    fib1_start_time = clock();
    int fib1ret = fib1(n);
    fib1_end_time = clock();
    fib1_cpu_time = ( (double) (fib1_end_time - fib1_start_time) ) / (double) CLOCKS_PER_
    printf("FIB1, %lf\n Fib1 returns: %i\n", fib1_cpu_time, fib1ret);

    return 0;
}
```

3.

A	B	O	o	Ω	ω	Θ
$\log n$	$\log(n^2)$	Y	N	Y	N	Y
$\log(n!)$	$\log(n^n)$	Y	N	Y	N	Y
$\sqrt[3]{n}$	$(\log n)^6$	N	N	Y	Y	N
$n^2 2^n$	3^n	Y	Y	N	N	N
$(n^2)!$	n^n	N	N	Y	Y	N
$\frac{n^2}{\log n}$	$n \log(n^2)$	N	N	Y	Y	N
$(\log n)^{\log n}$	$\frac{n}{\log(n)}$	N	N	Y	Y	N
$100n + \log n$	$(\log n)^3 + n$	Y	N	Y	N	Y

4. SEE HANDWRITTEN PROOFS

5. SEE HANDWRITTEN PROOFS