**Course: ENSF 694** – Summer 2025

**Lab #:** 02

**Instructor:** Dr. Mahmood Moussavi

**Student Name:** Jack Shenfield

**Submission Date:** Wednesday, July 16th, 2025

## Part I

### Exercise A

Source code:

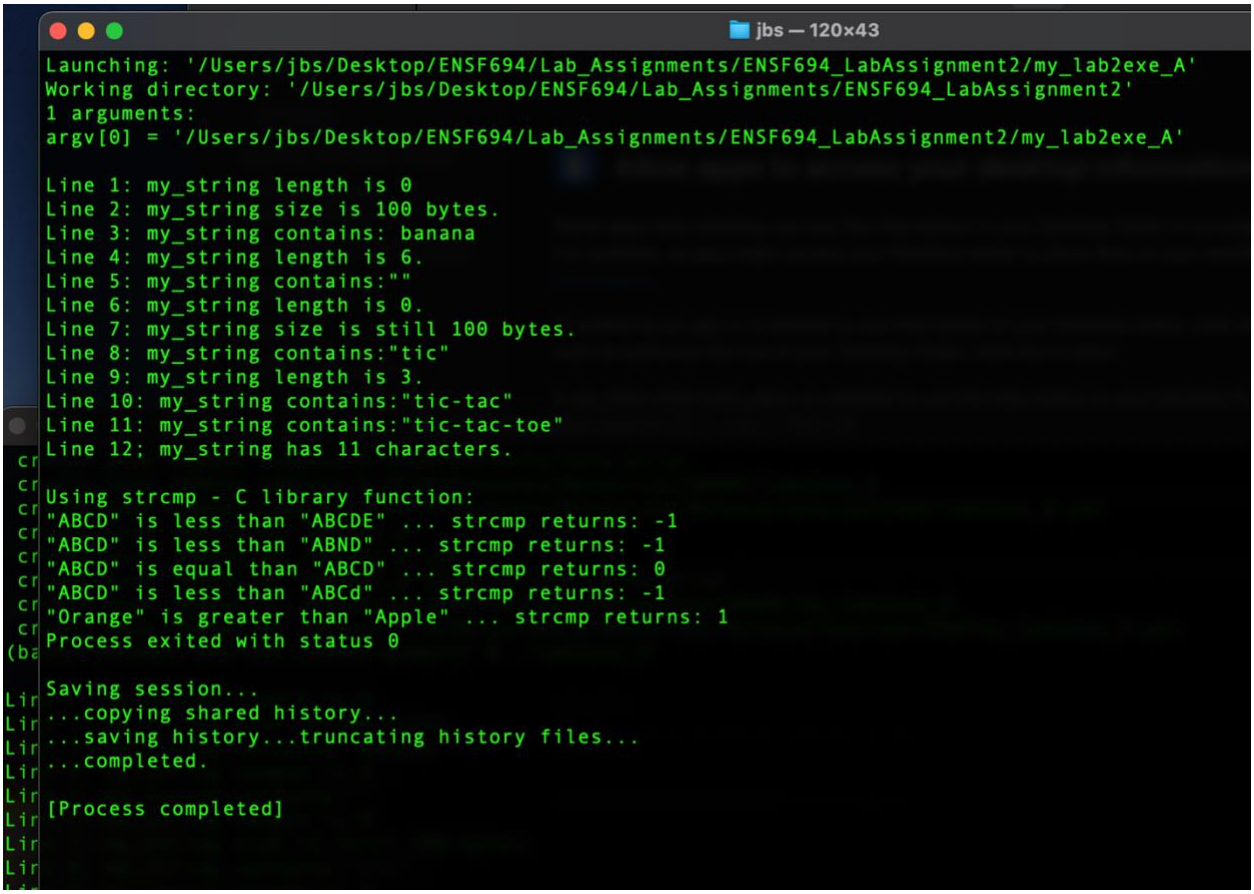
```
/*
 *  lab2exe_A.cpp
 *  ENSF 694 Lab 2 Exercise A
 *  Completed by: Jack Shenfield
 *  Development Date: July 16th, 2025
 */

int my_strlen(const char *s);
/*  Duplicates my_strlen from <cstring>, except return type is int.
 *  REQUIRES
 *    s points to the beginning of a string.
 *  PROMISES
 *    Returns the number of chars in the string, not including the
 *    terminating null.
 */

int my_strlen(const char *s){
    int counter = 0; // initialize counters
    int i = 0;
    while(*(s + i) != '\0'){ // while the value at index is not equal to the terminating character
        counter++; // increment counter and i
        i++;
    }

    return counter; // return the number of chars in the C string
}

void my_strncat(char *dest, const char *source, int n);
/*  Duplicates my_strncat from <cstring>, except return type is void.
 */

void my_strncat(char *dest, const char *source, int n){
```

```
    int i = 0;
    while(*(dest + i) != '\0'){ // while the value at index is not equal to the terminating character
        i++;
    }

    int j = 0;
    while(j < n && *(source + j) != '\0'){
        dest[i + j] = source[j]; // Add characters to destination array until terminating character is hit
        j++;
    }
    dest[i + j] = '\0'; // Add terminating character to end

}


#include <iostream>
#include <cstring>
using namespace std;

int main(void)
{
    char str1[7] = "banana";
    const char str2[] = "-tacit";
    const char* str3 = "-toe";

    /* point 1 */
    char str5[] = "ticket";
    char my_string[100]="";
    int bytes;
    int length;

    /* using my_strlen libarary function */
    length = (int) my_strlen(my_string);
    cout << "\nLine 1: my_string length is " << length;

    /* using sizeof operator */
    bytes = sizeof (my_string);
```

```
cout << "\nLine 2: my_string size is " << bytes << " bytes.";

/* using strcpy libarary function */
strcpy(my_string, str1);
cout << "\nLine 3: my_string contains: " << my_string;

length = (int) my_strlen(my_string);
cout << "\nLine 4: my_string length is " << length << ".";

my_string[0] = '\0';
cout << "\nLine 5: my_string contains:\"" << my_string << "\"";

length = (int) my_strlen(my_string);
cout << "\nLine 6: my_string length is " <<  length << ".";

bytes = sizeof (my_string);
cout << "\nLine 7: my_string size is still " << bytes << " bytes.";

/* my_strncat append the first 3 characters of str5 to the end of my_string */
my_strncat(my_string, str5, 3);
cout << "\nLine 8: my_string contains:\"" << my_string << "\"";

length = (int) my_strlen(my_string);
cout << "\nLine 9: my_string length is " << length << ".";

my_strncat(my_string, str2,  4);
cout << "\nLine 10: my_string contains:\"" << my_string << "\"";

/* my_strncat append ONLY up ot '\0' character from str3 -- not 6 characters */
my_strncat(my_string, str3, 6);
cout << "\nLine 11: my_string contains:\"" << my_string << "\"";

length = (int) my_strlen(my_string);
cout << "\nLine 12; my_string has " << length << " characters.";

cout << "\n\nUsing strcmp - C library function: ";
```

```
    cout << "\n\"ABCD\" is less than \"ABCDE\" ... strcmp returns: " <<
    strcmp("ABCD", "ABCDE");

    cout << "\n\"ABCD\" is less than \"ABND\" ... strcmp returns: " <<
    strcmp("ABCD", "ABND");

    cout << "\n\"ABCD\" is equal than \"ABCD\" ... strcmp returns: " <<
    strcmp("ABCD", "ABCD");

    cout << "\n\"ABCD\" is less than \"ABCd\" ... strcmp returns: " <<
    strcmp("ABCD", "ABCd");

    cout << "\n\"Orange\" is greater than \"Apple\" ... strcmp returns: " <<
    strcmp("Orange", "Apple") << endl;
    return 0;
}
```

Output:


jbs — 120×43
Launching: '/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2/my_lab2exe_A'
Working directory: '/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2'
1 arguments:
argv[0] = '/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2/my_lab2exe_A'
Line 1: my_string length is 0
Line 2: my_string size is 100 bytes.
Line 3: my_string contains: banana
Line 4: my_string length is 6.
Line 5: my_string contains:""
Line 6: my_string length is 0.
Line 7: my_string size is still 100 bytes.
Line 8: my_string contains:"tic"
Line 9: my_string length is 3.
Line 10: my_string contains:"tic-tac"
Line 11: my_string contains:"tic-tac-toe"
Line 12; my_string has 11 characters.
Using strcmp - C library function:
"ABCD" is less than "ABCDE" ... strcmp returns: -1
"ABCD" is less than "ABND" ... strcmp returns: -1
"ABCD" is equal than "ABCD" ... strcmp returns: 0
"ABCD" is less than "ABCd" ... strcmp returns: -1
"Orange" is greater than "Apple" ... strcmp returns: 1
Process exited with status 0
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.
[Process completed]

**Exercise B**

Source code:

```
/*
 *  lab2exe_B.cpp
 *  ENSF 694 Lab 2 Exercise B
 *  Completed by: Jack Shenfield
 *  Development Date: July 16th, 2025
 */

#include <iostream>
#include <assert.h>
using namespace std;

int sum_of_array(const int *a, int n);
// REQUIRES
//   n > 0, and elements a[0] ... a[n-1] exist.
// PROMISES:
//  Return value is a[0] + a[1] + ... + a[n-1].


int main()
{
    int a[] = { 100 };
    int b[] = { 100, 200, 300, 400 };
    int c[] = { -100, -200, -200, -300 };
    int d[] = { 10, 20, 30, 40, 50, 60, 70 };


    int sum = sum_of_array(a, 1);
    cout << "sum of integers in array a is: " << sum << endl;

    sum = sum_of_array(b, 4);
    cout << "sum of integers in array b is: " << sum << endl;

    sum = sum_of_array(c, 4);
    cout << "sum of integers in array c is: " << sum << endl;
```

```
    sum = sum_of_array(d, 7);
    cout << "sum of integers in array d is: " << sum << endl;

    return 0;
}

int sum_of_array(const int *a, int n)
{
    if(n == 0){ // if array is empty, return 0
        return 0;
    }

    else{
        int sum = 0; // initialize sum
        sum += (a[n-1] + sum_of_array(a, n-1)); // add to sum for each recursive step
        return sum; // return the final sum
    }

}
```

**Exercise D**

Source code:

```
/*
 *  lab2exe_D.cpp
 *  ENSF 694 Lab 2 Exercise D
 *  Completed by: Jack Shenfield
 *  Development Date: July 16th, 2025
 */

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
using namespace std;
#define N 2


void myPlot(int* x, double *y1, double *y2, int size){ // created with some help from chatgpt. This is my first time using GNUplot.
    // This funcitn must be completed by the students
    // iterative plot
    FILE* gp1 = popen("gnuplot -persistent", "w"); // open gnuplot
    if (gp1 == nullptr) {
        perror("Could not open pipe to Gnuplot (iterative)");
        exit(1);
    }
    // labels
    fprintf(gp1, "set title 'Iterative Fibonacci Timing'\n");
    fprintf(gp1, "set xlabel 'Input Size (n)'\n");
    fprintf(gp1, "set ylabel 'Time (seconds)'\n");
    fprintf(gp1, "plot '-' with lines title 'Iterative'\n");
```

```
    for (int i = 0; i < size; ++i) // data points
        fprintf(gp1, "%d %f\n", x[i], y1[i]);
    fprintf(gp1, "e\n");

    pclose(gp1);

    // recursive plot
    FILE* gp2 = popen("gnuplot -persistent", "w");
    if (gp2 == nullptr) {
        perror("Could not open pipe to Gnuplot (recursive)");
        exit(1);
    }
    // labels
    fprintf(gp2, "set title 'Recursive Matrix Fibonacci Timing'\n");
    fprintf(gp2, "set xlabel 'Input Size (n)'\n");
    fprintf(gp2, "set ylabel 'Time (seconds)'\n");
    fprintf(gp2, "plot '-' with lines title 'Recursive Matrix'\n");

    for (int i = 0; i < size; ++i) // print data points
        fprintf(gp2, "%d %f\n", x[i], y2[i]);
    fprintf(gp2, "e\n");

    pclose(gp2);
}




// Function to multiply two matrices of size N x N
void multiplyMatrix(int a[N][N], int b[N][N], int result[N][N]) {
    // This funcitn must be completed by the studnets
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            result[i][j] = 0; // set before doing += operation later, may result in garbage values
            for(int z = 0; z < N; z++){
                result[i][j] += a[i][z] * b[z][j];
            }
```

```
        }
    }
}

// Recursive funciont
void powerMatrix(int base[N][N], int exp, int result[N][N]) {
    // This funcitn must be completed by the students

    if(exp == 0){ // create identity matrix if exponent = 0
        for(int i = 0; i < N; i++){
            for(int j = 0; j < N; j++){
                if(i == j){
                    result[i][j] = 1; // 1s across the diagonal
                }
                else{
                    result[i][j] = 0; // all other indices filled by 0s
                }
            }
        }
        return;
    }

    else if(exp % 2 == 0){ // If exponent is even, square two of the matrices with half the exponent
        int temp[N][N];
        powerMatrix(base, exp/2, temp); // recursively call the same function
        multiplyMatrix(temp, temp, result);

    }

    else{ // If exponent is odd, square two of the matrices with half the exponent and multiply by matrix ^ 1
        int temp2[N][N];
        int tempSq[N][N];
        powerMatrix(base, exp/2, temp2);
        multiplyMatrix(temp2, temp2, tempSq);
        multiplyMatrix(tempSq, base, result);
```

```
    }
}

// Function to calculate the nth Fibonacci number using recursive matrix exponentiation
int fibonacciRecursive(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }

    int base[N][N] = {{1, 1}, {1, 0}};
    int result[N][N];
    powerMatrix(base, n - 1, result);
    return result[0][0];
}

// Function to calculate the nth Fibonacci number iteratively
int fibonacciIterative(int n) {
    // This funcitn must be completed by the students and if necessary its return value to be corrected.
    if(n == 1){ // have to set if/elseif statements for first two numbers, as they cannot be created from previous two values.
        return 0;
    }
    else if(n == 2){
        return 1;
    }

    int last = 0, curr = 1; // initialize first two values of sequence
    for (int i = 2; i <= n; i++) { // start at i = 2
        int next = last + curr;
        last = curr;
        curr = next;
    }
    return curr;
}
```

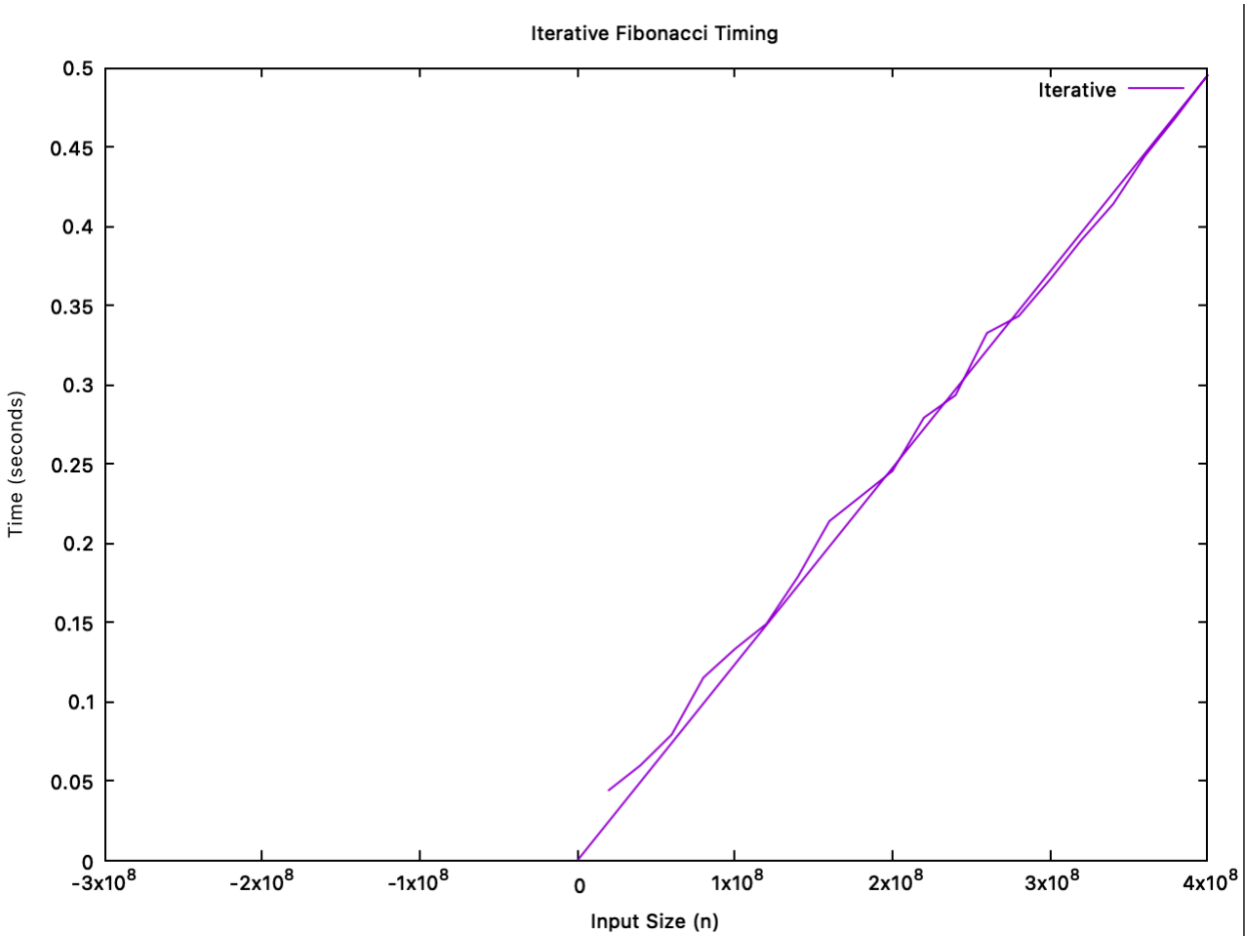
```cpp
// Function to measure the time taken by a function to calculate the nth Fibonacci number
// This function is using a pointer to a funciton called fibonacciFunc
double measureTime(int (*fibonacciFunc)(int), int n) {
    // This funcitn must be completed by the students and if necessary its return value to be corrected.

    auto start = std::chrono::high_resolution_clock::now(); // start the clock

    volatile int result = fibonacciFunc(n);  // Call fib func

    auto end = std::chrono::high_resolution_clock::now(); // end clock

    std::chrono::duration<double> duration = end - start; // calculate the duration and store in double
    return duration.count(); // return the double
}

int main(void) {
    const int maxN = 400000000; // Adjust maxN based on the range you want to test
    double recursive_result[50];
    double iterative_result[50];
    int N_value[50];

    cout << "Recursive Matrix Exponentiation Method\n";
    cout << setw(12) << "N" << setw(12) << "Time\n";
    for (int n = 20000000, i=0; n <= maxN; n+=20000000, i++) {
        double time = measureTime(fibonacciRecursive, n);
        recursive_result[i] = time;
        cout << setw(12) << n << setw(12) << recursive_result[i] << endl;
    }

    cout << "\nIterative Method\n";
    cout << setw(12) << "N" << setw(12) << "Time\n";
    for (int n = 20000000, i=0; n <= maxN; n+=20000000, i++) {
        double time = measureTime(fibonacciIterative, n);
        iterative_result[i] = time;
        cout << setw(12) << n << setw(12) << iterative_result[i] << endl;
        N_value[i] = n;
```

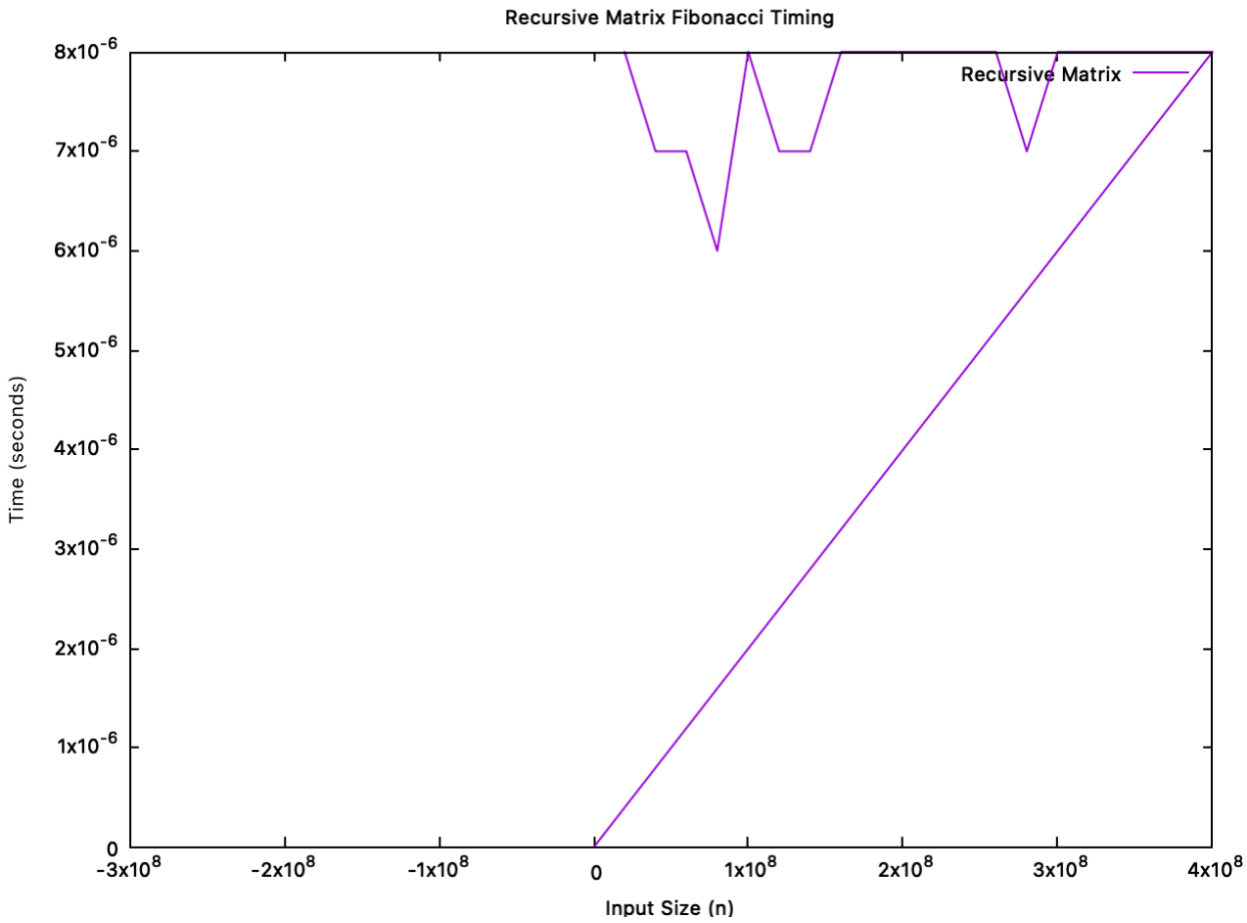
```
    }

    myPlot(N_value, iterative_result,  recursive_result, 30 );

    return 0;
}
```

Output:

```
Launching: '/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2/fibonacci'
Working directory: '/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2'
1 arguments:
argv[0] = '/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2/fibonacci'
Recursive Matrix Exponentiation Method
           N       Time
    20000000   3.917e-06
    40000000    3.25e-06
    60000000   3.334e-06
    80000000   3.292e-06
   100000000     3.5e-06
   120000000   3.458e-06
   140000000   2.833e-06
   160000000   2.792e-06
   180000000   2.792e-06
   200000000   2.875e-06
   220000000   2.917e-06
   240000000   2.792e-06
   260000000   2.958e-06
   280000000   2.791e-06
   300000000   2.834e-06
   320000000   2.917e-06
   340000000   2.958e-06
   360000000   2.916e-06
   380000000   2.958e-06
   400000000   3.042e-06

Iterative Method
           N       Time
    20000000    0.043145
    40000000   0.0586795
    60000000   0.0802353
    80000000    0.111807
   100000000     0.12874
   120000000    0.150411
   140000000    0.173209
   160000000    0.200833
   180000000    0.221542
   200000000    0.250998
   220000000    0.283205
   240000000     0.29613
   260000000    0.321245
   280000000    0.353617
   300000000    0.382412
   320000000    0.402952
   340000000    0.434519
   360000000    0.452419
   380000000    0.467485
```

GNU plots:


Iterative Fibonacci Timing
Iterative
Time (seconds)
Input Size (n)
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-3x10^8
-2x10^8
-1x10^8
0
1x10^8
2x10^8
3x10^8
4x10^8




Pay attention to the units on the y axis, clearly the recursive version is much, much faster.

### Exercise E

Source Code:

```
/*
 *  lab2exe_E.cpp
 *  ENSF 694 Lab 2 Exercise E
 *  Completed by: Jack Shenfield
 *  Development Date: July 16th, 2025
 */

#include "compare_sorts.h"
#include <ctype.h>
```

```
void to_lower(char *str)
{
    while (*str) {
        *str = std::tolower(*str);
        ++str;
    }
}

void strip_punctuation(char *word)
{
// Students should complete the implementation of this function
    if(word == NULL){
        return;
    }

    char * initial = word; // create new pointers to track comparisons through word
    char * final = word;

    while(*initial != '\0'){ // go through the entire C-string
        if(isalnum(*initial) || *initial == '-'){ // if the initial char is alphanumeric or a hyphen, keep it
            *final = *initial;
            final++; // increment final address
        }
        initial++; // increment initial address
    }

    *final = '\0';
}

bool is_unique(char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int num_words, const char *word)
{
    // Students should complete the implementation of this function
    bool flag = true; // assume unique

    for(int i = 0; i < num_words - 1; i++){ // run through all except the last word, which is compared by 2nd loop every
time
        for(int j = i + 1; j < num_words; j++){ // skip i word, compare all others
```

```
            if(strcmp(words[i], words[j]) == 0){ // string compare
                flag = false;
                return flag;
            }
        }
    }

    return flag;
}

void quicksort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int left, int right)
{
    // Students should complete the implementation of this function
    if (left >= right) return;

    int pivot = indices[(left + right)/2]; // initial pivot location
    int i = left;
    int j = right;

    while (i <= j) { // while left is less than or equal to right

        while (strcmp(words[indices[i]], words[pivot]) < 0){
            i++; // move towards middle
        }
        while (strcmp(words[indices[j]], words[pivot]) > 0){
            j--; // move towards middle
        }

        if (i <= j) { // swap
            int temp = indices[i];
            indices[i] = indices[j];
            indices[j] = temp;
            i++;
            j--;
        }
    }
```

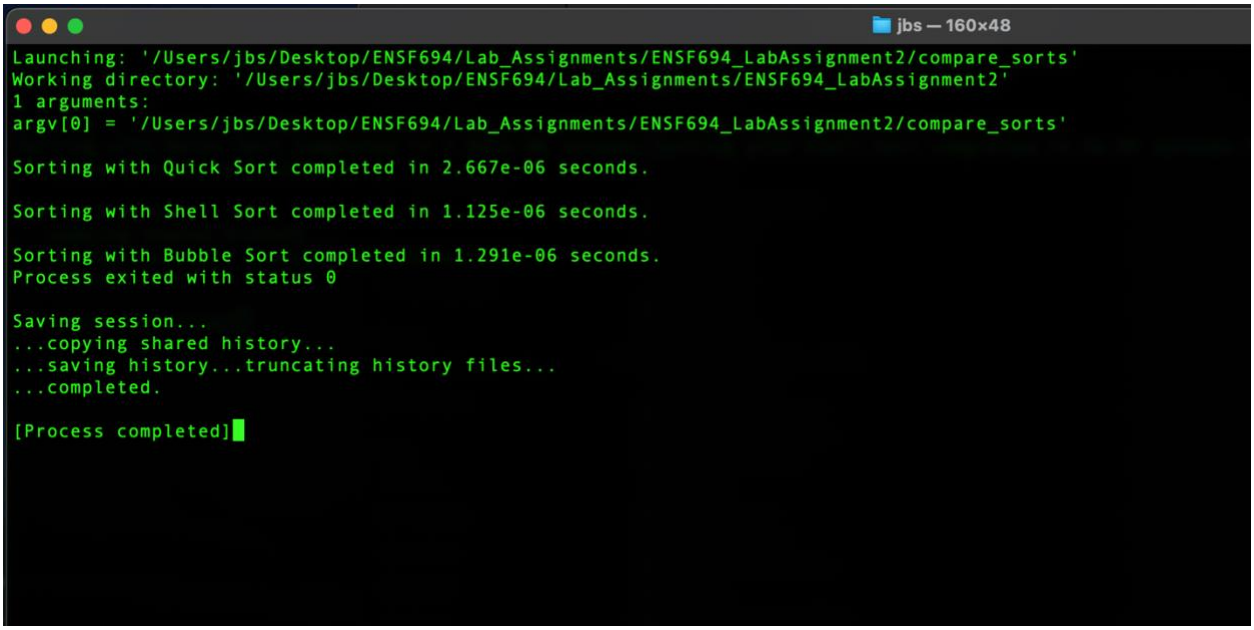
```
    quicksort(indices, words, left, j);
    quicksort(indices, words, i, right);
}

void shellsort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int size)
{
    // Students should complete the implementation of this function
    for (int gap = size / 2; gap > 0; gap /= 2) { // initialize gap at half of the size of array
        for (int i = gap; i < size; i++) {
            int temp = indices[i]; // store current value
            int j = i;

            while (j >= gap && strcmp(words[indices[j - gap]], words[temp]) > 0) { // if higher value is to the right
                indices[j] = indices[j - gap]; // save new value
                j -= gap;
            }

            indices[j] = temp; // save value for next loop
            }
        }
}

void bubblesort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int size)
{
    // Students should complete the implementation of this function
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (strcmp(words[indices[j]], words[indices[j + 1]]) > 0) { // check if the next value is less than

                int temp = indices[j]; // swap indices
                indices[j] = indices[j + 1];
                indices[j + 1] = temp;
            }
        }
    }
}
```

```
void read_words(const char *input_file, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int &num_words)
{
    std::ifstream infile(input_file);
    if (!infile) {
        std::cerr << "Error opening input file.\n";
        exit(1);
    }

    char word[MAX_WORD_SIZE + 1];
    num_words = 0;

    while (infile >> word) {
        strip_punctuation(word);
        to_lower(word);
        if (word[0] != '\0' && num_words < MAX_UNIQUE_WORDS && is_unique(words, num_words, word)) {
            std::strncpy(words[num_words++], word, MAX_WORD_SIZE);
        }
    }

    infile.close();
}

void write_words(const char *output_file, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int *indices, int
num_words)
{
    std::ofstream outfile(output_file);
    if (!outfile) {
        std::cerr << "Error opening output file.\n";
        exit(1);
    }

    for (int i = 0; i < num_words; ++i) {
        outfile << words[indices[i]] << '\n';
    }

    outfile.close();
}
```

```
void sort_and_measure_quicksort(char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int* indices, int
num_words, void (*sort_func)(int *, char [MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int, int), const char
*sort_name)
{
    // Students should complete the implementation of this function


    auto start = std::chrono::high_resolution_clock::now(); // start the clock

    sort_func(indices, words, 0, num_words - 1); // Call the function

    auto end = std::chrono::high_resolution_clock::now(); // end clock

    std::chrono::duration<double> duration = end - start; // calculate the duration and store in double

       std::cout << "\nSorting with Quick Sort completed in " << duration.count() << " seconds.\n"; // print the time
}

void sort_and_measure_shell_bubble(char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int* indices, int
num_words, void (*sort_func)(int *, char [MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int), const char *sort_name)
{
    // Students should complete the implementation of this function


    auto start = std::chrono::high_resolution_clock::now(); // start the clock

    sort_func(indices, words, num_words); // Call the function

    auto end = std::chrono::high_resolution_clock::now(); // end clock

    std::chrono::duration<double> duration = end - start; // calculate the duration and store in double
    if(sort_func == bubblesort){
        std::cout << "\nSorting with Bubble Sort completed in " << duration.count() << " seconds.\n"; // print the time
    }
    else{
        std::cout << "\nSorting with Shell Sort completed in " << duration.count() << " seconds.\n"; // print the time
```

```
    }

}

int main() {
    const char *input_file = "input.txt"; // Change this to your input file
    char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE];
    int num_words;

    read_words(input_file, words, num_words);

    int indices[num_words];
    for (int i = 0; i < num_words; ++i) {
        indices[i] = i;
    }

    sort_and_measure_quicksort(words,indices, num_words, quicksort, "Quick Sort");
    write_words("/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2/output_quicksort.txt",
words, indices, num_words);
    sort_and_measure_shell_bubble(words,indices, num_words, shellsort, "Shell Sort");
    write_words("/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2/output_shellsort.txt",
words, indices, num_words);
    sort_and_measure_shell_bubble(words, indices, num_words, bubblesort, "Bubble Sort");
    write_words("/Users/jbs/Desktop/ENSF694/Lab_Assignments/ENSF694_LabAssignment2/output_bubblesort.txt",
words, indices, num_words);
    return 0;
}
```

Program Output:




**Part II**

**Exercise A**

In order from lowest to highest growth rates:

2/N (approx. N^-1) – Decays towards 0 as N increases. Negative growth.

37 – Constant. Inbetween negative and positive growths.

sqrt(N) – Square root – less than just N.

N – Constant linear growth.

Nlog(N) – Linear*Logarithmic. Greater than previous growth rate of N.

2^0.5N – First exponential.

2^N – Similar to previous growth rate, but would obviously grow faster.

**Exercise B**

(1)

O(n)

There is one loop that runs up to n times (e.g. n potential iterations).

(2)

O(n^2)

There is a loop that runs n times with an embedded loop that also runs n times (e.g. n*n potential iterations).

(3)

O(n^3)

There is a loop that runs n times with an embedded loop that runs n*n times (e.g. n*n*n potential iterations).

(4)

O(n^2)

There is a loop that runs n times with an embedded loop that runs i times (e.g. n*i potential iterations). This i runs until n-1 potentially. Thus, n^2 would be the highest order term in the equation.

(5)

O(n^3)

There is a loop that runs n times with an embedded loop that runs i times with an additional embedded loop that runs j times (e.g. n*i*j potential iterations). Similar to the previous part, i runs until n-1 and j until n-2, thus the highest order term would be n*n*n.

(6)

O(n^3)

Two embedded loops that run n times, and the base loop also runs n times. Thus, n*n*n potential iterations.