ENSF 694 – Summer 2024

Principles of Software Development II

University of Calgary

Lab Assignment 2

Student Name: Yael Gonzalez

Instructor: M. Moussavi, PhD, Peng

Submission Date: July 10, 2024

Part I – Algorithm Development

Exercise A: Duplicating string library functions

Source code

```
/**
 *  File Name: lab2exe_A.cpp
 *  Assignment: ENSF 694 Lab 2 Exercise A
 *  Created by: Mahmood Moussavi
 *  Completed by: Yael Gonzalez
 *  Submission Date: July 10, 2024
 */

int my_strlen(const char *s);
/**
 *  Duplicates 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.
 */

void my_strncat(char *dest, const char *source, int n);
/**
 *  Duplicates my_strncat from <cstring>, except return type is void.
 *  REQUIRES:
 *     dest and source each point to the beginning of a string.
 *     n > 0, where n is the number of characters to be concatenated
 *     from source onto dest.
 *     source strlen >= n.
 *     dest >= dest + n, i.e., enough space in dest to contain its
 *     current length plus the added characters.
 *  PROMISES:
 *     concatenates up to n characters from the source string to the
 *     end of the dest string.
 */

#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 function */
length = 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 = 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 = 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 = 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 = 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;
}

int my_strlen(const char *s)
{
    const char *e = s;

    // Move the pointer 'e' to the last char (stops at '\0')
    while (*e)
        e++;

    return (int)(e - s); // e (end) - s (start) = string length
}

void my_strncat(char *dest, const char *source, int n)
{
    // Move the pointer 'dest' to the last char (stops at '\0')
    while (*dest)
        dest++;

    // Iterate n times. For each cycle, assign value of source to dest
    // and move both pointers
```

```
    for (int i = 0; i < n; i++)
    {
        *dest = *source;
        dest++;
        source++;
    }

    // After adding n values of source to dest, make sure to return a
    // c-string, i.e., add a '\0' at the end.
    *dest = '\0';
}
```

Program output

```
PROBLEMS    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE    GITLENS

PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_A> g++ -Wall my_lab2exe_A.cpp -o my_lab2exe_A
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_A> .\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
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_A>
```

Exercise B: Understanding Recursion

Source code

```
/**
 *  File Name: lab2exe_B.cpp
 *  Assignment: ENSF 694 Lab 2 Exercise B
 *  Created by: Mahmood Moussavi
 *  Completed by: Yael Gonzalez
 *  Submission Date: July 10, 2024
 */
```

```cpp
#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)
        return 0;

    return *a + sum_of_array(a + 1, n - 1);
}
```

Program output

```
PROBLEMS   OUTPUT   TERMINAL   PORTS   DEBUG CONSOLE   GITLENS

PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_B> g++ -Wall lab2exe_B.cpp -o lab2exe_B
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_B> .\lab2exe_B
sum of integers in array a is: 100
sum of integers in array b is: 1000
sum of integers in array c is: -800
sum of integers in array d is: 280
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_B> 
```

Exercise D: A Recursive Method for Fibonacci Sequence

Source code

fibonacci.h

```
/**
 *  File Name: fibonacci.h
 *  Assignment: ENSF 694 Lab 2 Exercise D
 *  Created by: Mahmood Moussavi
 *  Completed by: Yael Gonzalez
 *  Submission Date: July 10, 2024
 */

#ifndef FIBONACCI_H
#define FIBONACCI_H

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <chrono>
using namespace std;

#define N 2


void plotMethodComparison(int* x, double *y1, double *y2, int size);
/**
 * Function to plot the algorithm time analysis.
 *
 * REQUIRES:
 *  x points to an array of integers representing the x-axis data.
```

```
 *  y1 points to an array of doubles representing the y-axis data for the first
method.
 *  y2 points to an array of doubles representing the y-axis data for the second
method.
 *  size is the number of data points.
 * PROMISES:
 *  Plots a comparison of the algorithm time analysis for two methods using the
provided data.
 */

void plotMethod(const char* Title, int* x, double *y, int size);
/**
 * REQUIRES:
 *  Title is the title of the plot.
 *  x points to an array of integers representing the x-axis data.
 *  y points to an array of doubles representing the y-axis data.
 *  size is the number of data points.
 * PROMISES:
 *  Plots the provided data with the given title.
 */

void multiplyMatrix(int a[N][N], int b[N][N], int result[N][N]);
/**
 * Function to multiply two matrices of size N x N.
 *
 * Source:
 * https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
 *
 * REQUIRES:
 *  a and b are NxN matrices to be multiplied.
 *  result is an NxN matrix to store the result of the multiplication.
 * PROMISES:
 *  Multiplies matrices a and b, storing the result in the result matrix.
 */

void powerMatrix(int base[N][N], int exp, int result[N][N]);
/**
 * Power of Matrix recursive method.
 *
 * Source:
 * https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/
 * https://robwilsondev.medium.com/bigo-and-beyond-how-to-compute-fibonacci-sequence-
efficiently-with-matrix-exponentiation-d9924545fe54
 *
 * REQUIRES:
```

```
 *  base is an NxN matrix to be exponentiated.
 *  exp is the exponent.
 *  result is an NxN matrix to store the result of the exponentiation.
 * PROMISES:
 *  Computes the power of the base matrix raised to the exp, storing the result in
the result matrix.
 */

int fibonacciRecursive(int n);
/**
 * Function to calculate the nth Fibonacci number using recursive matrix
exponentiation
 *
 * REQUIRES:
 *  n > 0, where n is the n-th number of the Fibonacci sequence.
 * PROMISES:
 *  Returns the nth Fibonacci number using recursive matrix exponentiation.
 */

int fibonacciIterative(int n);
/**
 * Function to calculate the nth Fibonacci number iteratively
 *
 * Source:
 * https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/
 *
 * REQUIRES:
 *  n > 0, where n is the n-th number of the Fibonacci sequence.
 * PROMISES:
 *  Returns the nth Fibonacci number using an iterative approach.
 */

double measureTime(int (*fibonacciFunc)(int), int n);
/**
 * Function to measure the time taken by a function to calculate the nth Fibonacci
number
 *
 * REQUIRES:
 *  fibonacciFunc is a pointer to a function that calculates the nth Fibonacci
number.
 *  n > 0, where n is the position of the desired Fibonacci number.
 * PROMISES:
 *  Measures and returns the time taken by the fibonacciFunc to calculate the nth
Fibonacci number.
 */
```

```
int* printTimeTable(const string Title, int (*fibonacciFunc)(int), double result[],
int n, int maxN, int n_stepsize, int N_value[]);
/**
 * REQUIRES:
 *  Title is the title of the table.
 *  fibonacciFunc is a pointer to a function that calculates the nth Fibonacci
number.
 *  result is an array to store the measured times.
 *  n is the initial position of the Fibonacci number.
 *  maxN is the maximum position of the Fibonacci number.
 *  n_stepsize is the step size for n.
 *  N_value is an array to store the positions of Fibonacci numbers.
 * PROMISES:
 *  Prints a table of times taken by fibonacciFunc to calculate Fibonacci numbers
from n to maxN, with step size n_stepsize.
 */


#endif
```

fibonacci.cpp

```
/**
 *  File Name: fibonacci.cpp
 *  Assignment: ENSF 694 Lab 2 Exercise D
 *  Created by: Mahmood Moussavi
 *  Completed by: Yael Gonzalez
 *  Submission Date: July 10, 2024
 */

#include "fibonacci.h"

int main(void)
{
    double recursive_result[50] = {0.0};
    double iterative_result[50] = {0.0};;
    int N_value[50] = {0};

    // Uncomment for analyzing each method separately
    // printTimeTable("Recursive Matrix Exponentiation Method", fibonacciRecursive,
recursive_result, 0, 46, 1, N_value);
    // printTimeTable("\nIterative Method", fibonacciIterative, iterative_result, 0,
46, 1, N_value);
```

```
    // plotMethod("Recursive Matrix Exponentiation Method", N_value,
recursive_result, 47);
    // plotMethod("Iterative Method", N_value, iterative_result, 47);

    // Uncomment for analyzing both methods together
    printTimeTable("\nRecursive Matrix Exponentiation Method", fibonacciRecursive,
recursive_result, 0, 10000, 500, N_value);
    printTimeTable("\nIterative Method", fibonacciIterative, iterative_result, 0,
10000, 500, N_value);
    plotMethodComparison(N_value, iterative_result,  recursive_result, 21);

    return 0;
}

void multiplyMatrix(int a[N][N], int b[N][N], int result[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            result[i][j] = 0; // Set all values of result matrix to 0
            for (int k = 0; k < N; k++)
            {
                result[i][j] += a[i][k] * b[k][j]; // Perform matrix multiplication
algorithm
            }
        }
    }
}

void powerMatrix(int base[N][N], int exp, int result[N][N])
{
    if (exp == 1) // Base case
    {
        std::copy(&base[0][0], &base[0][0] + N * N, &result[0][0]); // result = base
        return;
    }

    int half[N][N] = {0}; // Initialize half matrix with Zeroes
    powerMatrix(base, exp / 2, half); // half = A^(n/2), where 'A' is 'base', and 'n'
is 'exp'
    multiplyMatrix(half, half, result); // result = A^(n/2) * A^(n/2)

    if (exp % 2 != 0) // If it's odd multiply result by A
    {
```

```
        int temp[N][N] = {0}; // Initialize half matrix with Zeroes
        multiplyMatrix(result, base, temp); // temp = A^(n/2) * A^(n/2) * A
        std::copy(&temp[0][0], &temp[0][0] + N * N, &result[0][0]); // result = temp
    }
}

int fibonacciRecursive(int n)
{
    if (n == 0)
    {
        // std::cout << setw(12) << 0; // Uncomment to print Fibonacci num value
        return 0;
    }

    if (n == 1)
    {
        // std::cout << setw(12) << 1; // Uncomment to print Fibonacci num value
        return 1;
    }

    int base[N][N] = {{1, 1}, {1, 0}};
    int result[N][N] = {0};
    powerMatrix(base, n - 1, result);
    // std::cout << setw(12) << result[0][0]; // Uncomment to print Fibonacci num
value
    return result[0][0];
}

int fibonacciIterative(int n)
{
    int a = 0, b = 1, c = 0;

    if (n == 0)
    {
        // std::cout << setw(12) << a;  // Uncomment to print Fibonacci num value
        return a;
    }

    for (int i = 2; i <= n; i++)
    {
        c = a + b;
        a = b;
        b = c;
    }
```

```
    // std::cout << setw(12) << b; // Uncomment to print Fibonacci num value
    return b;
}

double measureTime(int (*fibonacciFunc)(int), int n)
{
    const auto start_time = std::chrono::high_resolution_clock::now();
    fibonacciFunc(n);
    const auto end_time = std::chrono::high_resolution_clock::now();
    const std::chrono::duration<double> time_diff = end_time - start_time;
    return time_diff.count() * 1000.0; // measure in miliseconds
}

int* printTimeTable(const string Title, int (*fibonacciFunc)(int), double result[],
int n, int maxN, int n_stepsize, int N_value[])
{
    std::cout << Title << "\n";
    // Uncomment to print Fibonacci num value
    // std::cout << setw(12) << "Fibonacci Num" << setw(12) << "N" << setw(12) <<
"Time (ms)" << "\n";
    std::cout << setw(12) << "N" << setw(12) << "Time (ms)" << "\n";
    for (int i = 0; n <= maxN; n += n_stepsize, i++) {
        double time = measureTime(fibonacciFunc, n);
        result[i] = time;
        std::cout << setw(12) << n << setw(12) << result[i] << endl;
        N_value[i] = n;
    }

    return N_value;
}

void plotMethodComparison(int* x, double *y1, double *y2, int size)
{
    FILE * gnuplotPipe = popen ("C:\\msys64\\ucrt64\\bin\\gnuplot.exe -persistent",
"w");

    const char* name = "Fibonacci Time Analysis";

    fprintf(gnuplotPipe, "set title '%s'\n", name);
    fprintf(gnuplotPipe, "set xlabel 'n-th Fibonacci number'\n");
    fprintf(gnuplotPipe, "set ylabel 'running time (ms)'\n");
    fprintf(gnuplotPipe, "set key top center horizontal\n");
    fprintf(gnuplotPipe, "plot '-' with linespoints pt 5 ps 1 lc 'blue' title
'Iterative', \
```

```
        '-' with linespoints pt 7 ps 1 lc 'red' title 'Recursive (Power of
Matrix)'\n");

    // Iterative
    for (int i = 0; i < size; i++)
    {
        fprintf(gnuplotPipe, "%d %f\n", x[i], y1[i]);
    }

    fprintf(gnuplotPipe, "e\n");

    // Recursive
    for (int i = 0; i < size; i++)
    {
        fprintf(gnuplotPipe, "%d %f\n", x[i], y2[i]);
    }
}

void plotMethod(const char* Title, int* x, double *y, int size)
{
    FILE * gnuplotPipe = popen ("C:\\msys64\\ucrt64\\bin\\gnuplot.exe -persistent",
"w");

    fprintf(gnuplotPipe, "set title '%s'\n", Title);
    fprintf(gnuplotPipe, "set xlabel 'n-th Fibonacci number'\n");
    fprintf(gnuplotPipe, "set ylabel 'running time (ms)'\n");
    fprintf(gnuplotPipe, "set key top center horizontal\n");
    fprintf(gnuplotPipe, "plot '-' with linespoints pt 5 ps 1 lc 'blue' title
'%s'\n", Title);

    for (int i = 0; i < size; i++)
    {
        fprintf(gnuplotPipe, "%d %f\n", x[i], y[i]);
    }
}
```

Output

Printing the first 50 N-th numbers of the Fibonacci sequence, we observe that both methods, Iterative and Recursive, return the correct values of Fibonacci until N = 47, where we start to get data overflow due to type int:

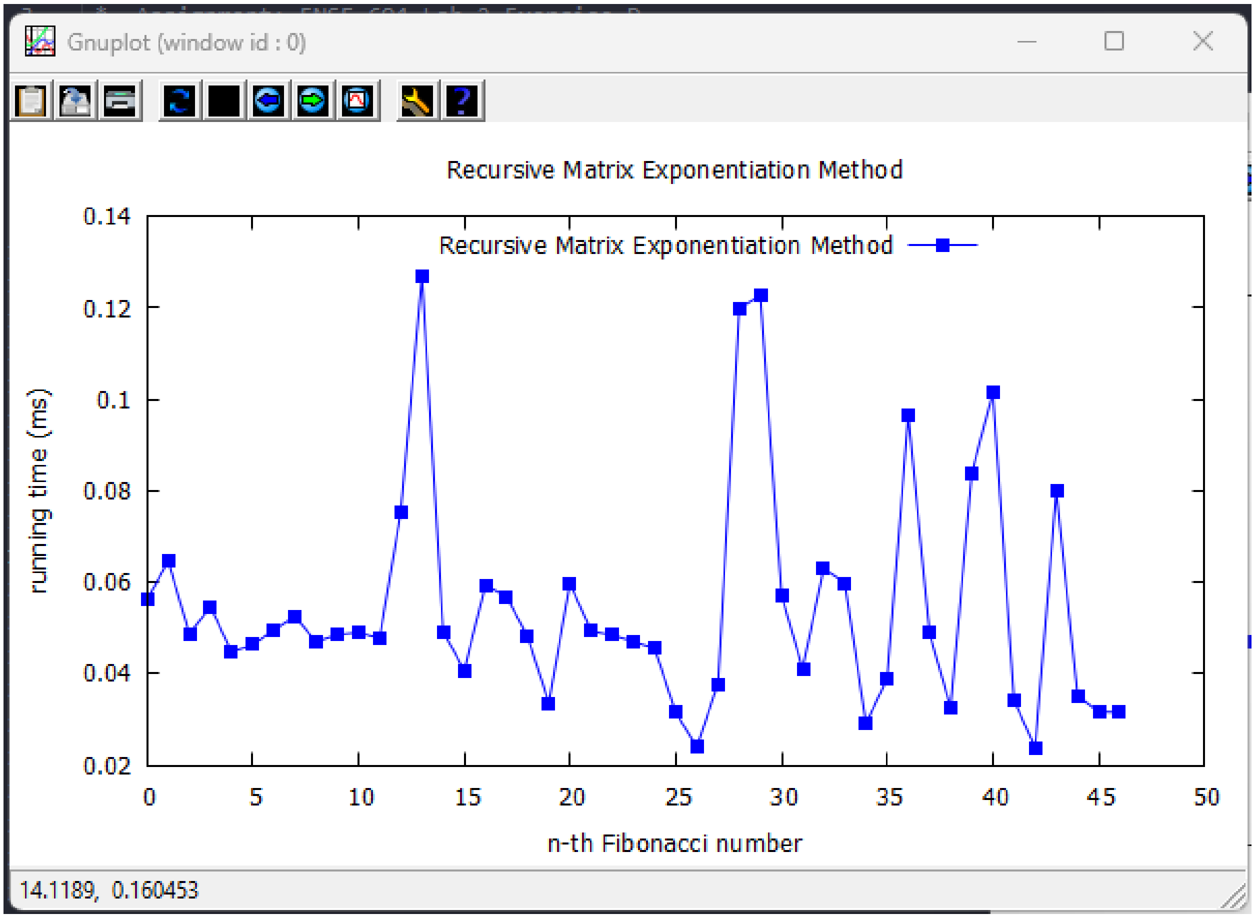
Recursive Matrix Exponentiation Method

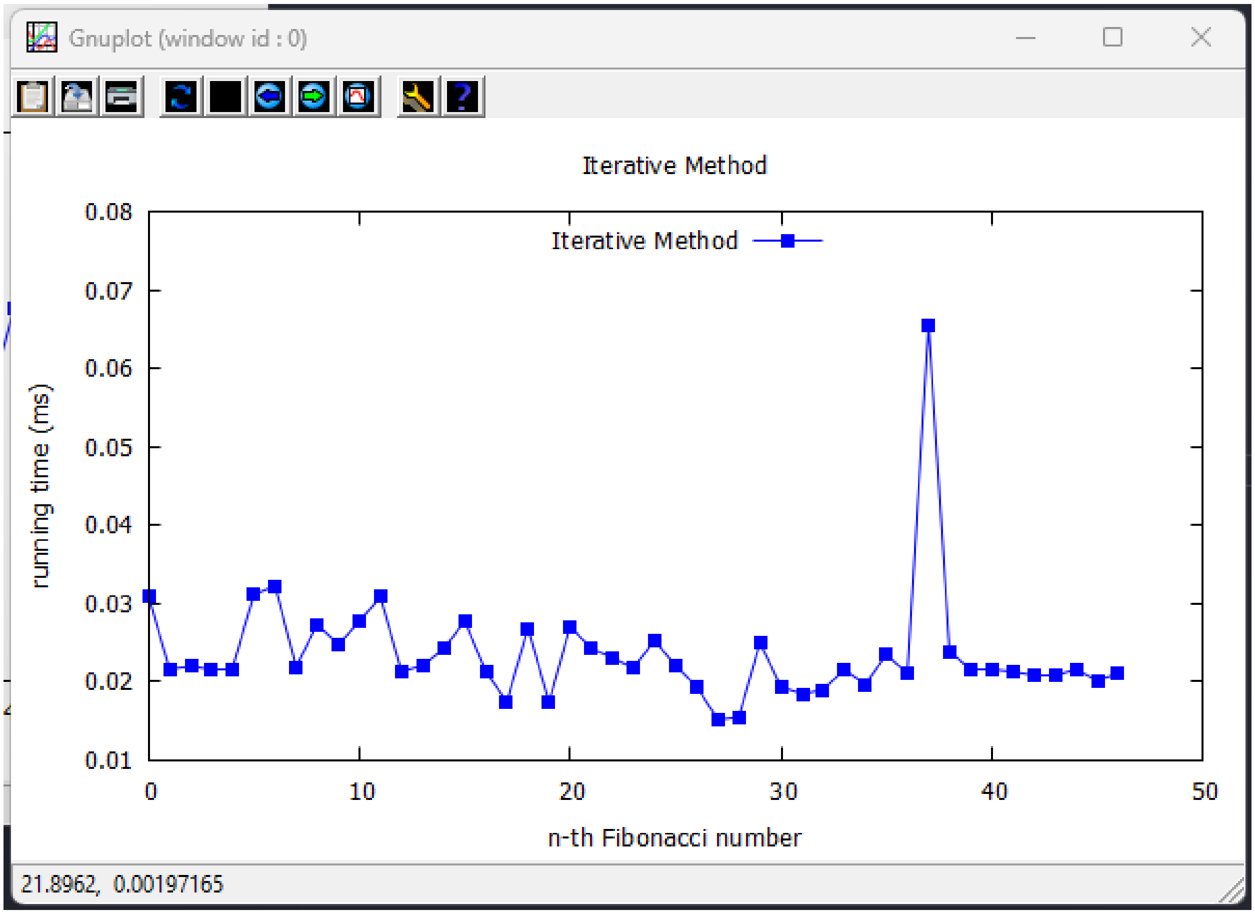
Fibonacci Num	N	Time (ms)
0	0	0.0795
1	1	0.0524
1	2	0.0531
2	3	0.0526
3	4	0.0532
5	5	0.0527
8	6	0.0533
13	7	0.0529
21	8	0.0528
34	9	0.053
55	10	0.0524
89	11	0.0526
144	12	0.0488
233	13	0.0836
377	14	0.0783
610	15	0.11
987	16	0.2252
1597	17	0.0742
2584	18	0.117
4181	19	0.0807
6765	20	0.0767
10946	21	0.074
17711	22	0.1397
28657	23	0.0662
46368	24	0.153
75025	25	0.0622
121393	26	0.0605
196418	27	0.0877
317811	28	0.0373
514229	29	0.122
832040	30	0.0647
1346269	31	0.0817
2178309	32	0.162
3524578	33	0.0714
5702887	34	0.0741
9227465	35	0.041
14930352	36	0.0328
24157817	37	0.0706
39088169	38	0.0478
63245986	39	0.0467
102334155	40	0.0669
165580141	41	0.1735
267914296	42	0.2362
433494437	43	0.1131
701408733	44	0.0657
1134903170	45	0.0699
1836311903	46	0.1988
-1323752223	47	0.0282
512559680	48	0.0385
-811192543	49	0.0287

Iterative Method

Fibonacci Num	N	Time (ms)
0	0	0.0768
1	1	0.0189
1	2	0.0336
2	3	0.0233
3	4	0.2985
5	5	0.0274
8	6	0.0276
13	7	0.019
21	8	0.0187
34	9	0.0263
55	10	0.1203
89	11	0.0424
144	12	0.0288
233	13	0.0294
377	14	0.0317
610	15	0.0323
987	16	0.033
1597	17	0.032
2584	18	0.037
4181	19	0.0269
6765	20	0.0228
10946	21	0.0231
17711	22	0.0273
28657	23	0.0222
46368	24	0.0841
75025	25	0.0118
121393	26	0.0169
196418	27	0.0328
317811	28	0.0243
514229	29	0.0188
832040	30	0.0159
1346269	31	0.0195
2178309	32	0.0163
3524578	33	0.0179
5702887	34	0.0162
9227465	35	0.0157
14930352	36	0.0153
24157817	37	0.0156
39088169	38	0.0168
63245986	39	0.0157
102334155	40	0.0154
165580141	41	0.0153
267914296	42	0.0153
433494437	43	0.0146
701408733	44	0.0157
1134903170	45	0.0158
1836311903	46	0.0169
-1323752223	47	0.0155
512559680	48	0.0159
-811192543	49	0.0157

When plotting the value of N against the running time in milliseconds in the span where we get correct values of the Fibonacci sequence (i.e., N from 0 to 46), we don't see a clear pattern for the time complexity analysis:

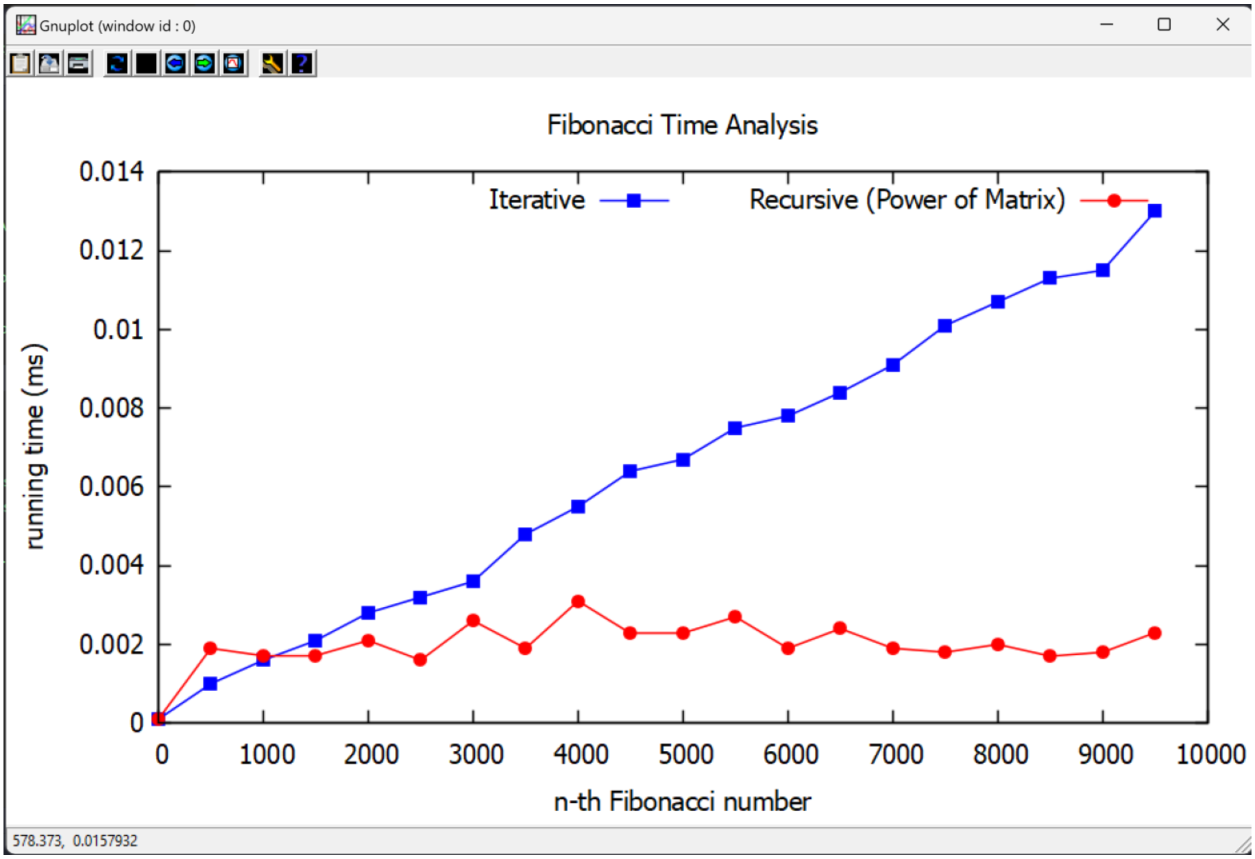




However, when plotting a wider range of N-th Fibonacci numbers (e.g., from 0 to 10,000) and a larger step size of N (e.g., 500), we can observe the expected time complexity behavior for each of the methods, where Iterative displays O(n) and Recursive by Matrix Exponentiation displays O(log2(n)):

```
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_D> .\fibonacci
Recursive Matrix Exponentiation Method
           N  Time (ms)
           0     0.0001
         500     0.0019
        1000     0.0017
        1500     0.0017
        2000     0.0021
        2500     0.0016
        3000     0.0026
        3500     0.0019
        4000     0.0031
        4500     0.0023
        5000     0.0023
        5500     0.0027
        6000     0.0019
        6500     0.0024
        7000     0.0019
        7500     0.0018
        8000      0.002
        8500     0.0017
        9000     0.0018
        9500     0.0023
       10000     0.0026

Iterative Method
           N  Time (ms)
           0     0.0001
         500      0.001
        1000     0.0016
        1500     0.0021
        2000     0.0028
        2500     0.0032
        3000     0.0036
        3500     0.0048
        4000     0.0055
        4500     0.0064
        5000     0.0067
        5500     0.0075
        6000     0.0078
        6500     0.0084
        7000     0.0091
        7500     0.0101
        8000     0.0107
        8500     0.0113
        9000     0.0115
        9500      0.013
       10000     0.0319
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_D>
```



Exercise E: Using Different Sorting Techniques

Source code

compare_sorts.h

```
//
//  compare_sorts.h
//  Compare Sort Methods
//
//  Created by Mahmood Moussavi on 2024-06-06.
//  Completed by Yael Gonzalez on 2024-07-08.
//

#ifndef compare_sorts_h
#define compare_sorts_h
#include <iostream>
#include <fstream>
#include <cstring>
#include <cstdlib>
#include <cctype>
#include <chrono>
```

```
const int MAX_WORD_SIZE = 20;
const int MAX_UNIQUE_WORDS = 10000;

void to_lower(char *str);
/* REQUIRES: str points to valid c-string terminated with a '\0'
 * PROMISES: changes any upper case character to a lowercase.
 */
void strip_punctuation(char *word);
/* REQUIRES: word points to valid c-string terminated with a '\0'
 * PROMISES: strips out any non-alphanumeric characters. Also keeps
 * hyphens.
 */
bool is_unique(char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int num_words, const char
*word);
/* REQUIRES: words refer to a 2-D arry of character, and each row is a valid
 * c-string terminated with a '\0'
 * PROMISES: returns true if words in in the arra are unique. Otherwise, returns
false
 */
void quicksort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int left,
int right);
/* REQUIRES: indices points to an arry of integer, holidng index numbers,
 * words refer to a 2-D arry of character, and each row is a valid c-string
terminated with a '\0'
 * PROMISES: uses quick sort algorithm to sort the words in ascending order.
 * Source: https://www.programiz.com/dsa/quick-sort
 */
int partition(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int left,
int right);
/* REQUIRES: indices points to an arry of integer, holidng index numbers,
 * words refer to a 2-D arry of character, and each row is a valid c-string
terminated with a '\0'
 * PROMISES: finds the partition position to be used in the quicksort pivot
 * Source: https://www.programiz.com/dsa/quick-sort
*/
void shellsort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int size);
/* REQUIRES: indices points to an arry of integer, holidng index numbers,
 * words refer to a 2-D arry of character, and each row is a valid c-string
terminated with a '\0'
 * PROMISES: uses shell sort algorithm to sort the words in ascending order.
 * Source: Slide "14_More on Algorithms & Complexity Analysis" created by Mahmood
Moussavi
 */
void bubblesort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int size);
/* REQUIRES: indices points to an arry of integer, holidng index numbers,
```

```
 * words refer to a 2-D arry of character, and each row is a valid c-string
terminated with a '\0'
 * PROMISES: uses bubble sort algorithm to sort the words in ascending order.
 * Source: Slide "14_More on Algorithms & Complexity Analysis" created by Mahmood
Moussavi
 */
void read_words(const char *input_file, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE],
int &num_words);
/* REQUIRES: words refer to a 2-D arry of character, and each row is a valid c-string
 * terminated with a '\0'
 * PROMISES: opens an input file, reads each word from the file, and saves the word
with
 * no punctuations, all lowercase, into array of words, and updates numbers of words
to
 * assure they are less than MAX_UNIQUE_WORDS
 */
void write_words(const char *output_file, char
words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int *indices, int num_words);
/* REQUIRES: words refer to a 2-D arry of character (number of rows: num_words),
 * and each row is a valid c-string terminated with a '\0'
 * PROMISES: opens an output file, writess the word referred by order of indecies
into the
 * output file.
 */
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);
/* REQUIRES: words refer to a 2-D arry of character, and each row is a valid c-string
 * terminated with a '\0', num_words refers to number of words in the 2-D array,
sort_func
 * points to a quicksort function.
 * PROMISES: uses std::chrono::high_resolution_clock::now, before and after call to
the sort
 * function
 */
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);
/* REQUIRES: words refer to a 2-D arry of character, and each row is a valid c-string
 * terminated with a '\0', num_words refers to number of words in the 2-D array,
sort_func
 * points to a shell or bubble sort function.
 * PROMISES: uses std::chrono::high_resolution_clock::now, before and after call to
the sort
 * function
```

```
 */


#endif /* compare_sorts_h */
```

compare_sorts.cpp

```
/**
 *  File Name: compare_sorts.cpp
 *  Assignment: ENSF 694 Lab 2 Exercise E
 *  Created by: Mahmood Moussavi
 *  Completed by: Yael Gonzalez
 *  Submission Date: July 10, 2024
 */

#include "compare_sorts.h"

int main() {
    const char *input_file = "feynman.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("output_quicksort.txt", words, indices, num_words);
    sort_and_measure_shell_bubble(words, indices, num_words, shellsort, "Shell
Sort");
    write_words("output_shellsort.txt", words, indices, num_words);
    sort_and_measure_shell_bubble(words, indices, num_words, bubblesort, "Bubble
Sort");
    write_words("output_bubblesort.txt", words, indices, num_words);
    return 0;
}

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

```
        ++str;
    }
}

void strip_punctuation(char *word) {
    char *src_word = word;

    while (*src_word) {
        if(isalnum(*src_word) || *src_word == '-') {
            *word = *src_word;
            word++;
        }
        src_word++;
    }
    *word = '\0';
}

bool is_unique(char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int num_words, const
char *word) {
    for (int i = 0; i < num_words; i++) {
        if (strcmp(words[i], word) == 0) {
            return false;
        }
    }
    return true;
}

void quicksort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int
left, int right) {
    if (left < right) {
        // Find the partitoin index (pi) such that:
        // - Elements smaller than pivot are on left of pivot
        // - Elements greater than pivot are on righ of pivot
        int pi = partition(indices, words, left, right);

        // Recursive call on the left of pi
        quicksort(indices, words, left, pi - 1);

        // Recursive call on the right of pi
        quicksort(indices, words, pi + 1, right);
    }
}

int partition(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int
left, int right) {
```

```
    // Select the rightmost element as pivot
    int pivot = indices[right];

    // pointer for greater element
    int i = (left - 1);

    // traverse each element of the array
    // compare them with the pivot
    for (int j = left; j < right; j++) {
        if (strcmp(words[indices[j]], words[pivot]) <= 0) {
            // if element smaller than pivot is found
            // swap it with the greater element pointed by i
            i++;

            // swap element at i with element at j
            std::swap(indices[i], indices[j]);
        }
    }
    // swap pivot with the greater element at i
    std::swap(indices[i + 1], indices[right]);

    // return the partition point
    return (i + 1);
}

void shellsort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int
size) {
    // Start with a big gap, then reduce the gap
    for (int gap = size / 2; gap > 0; gap /= 2) {
        // Perform a gapped insertion
        for (int i = gap; i < size; i++) {
            int temp = indices[i];
            int j;
            for (j = i; j >= gap && strcmp(words[indices[j - gap]], words[temp])
> 0; j -= gap) {
                indices[j] = indices[j - gap];
            }
            indices[j] = temp;
        }
    }
}

void bubblesort(int *indices, char words[MAX_UNIQUE_WORDS][MAX_WORD_SIZE], int
size) {
    for (int i = 0; i < size - 1; i++) {
```

```
        for (int j = 0; j < size - 1 - i; j++) {
            // Compare the words at the current indices
            if (strcmp(words[indices[j]], words[indices[j + 1]]) > 0) {
                // Swap the indices if out of order
                int temp = indices[j];
                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) {
    const auto start_time = std::chrono::high_resolution_clock::now();
    sort_func(indices, words, 0, num_words - 1);
    const auto end_time = std::chrono::high_resolution_clock::now();
    const std::chrono::duration<double> time_diff = end_time - start_time;
    std::cout << "Sorting with " << sort_name << " completed in " << time_diff.count() << " seconds.\n";
}

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) {
    const auto start_time = std::chrono::high_resolution_clock::now();
    sort_func(indices, words, num_words);
    const auto end_time = std::chrono::high_resolution_clock::now();
    const std::chrono::duration<double> time_diff = end_time - start_time;
    std::cout << "Sorting with " << sort_name << " completed in " << time_diff.count() << " seconds.\n";
}
```

Program output

```
PROBLEMS    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE    GITLENS

PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_E> g++ -Wall -std=gnu++23 .\compare_sorts.cpp -o .\compare_sorts
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_E> .\compare_sorts
Sorting with Quick Sort completed in 8.3e-05 seconds.
Sorting with Shell Sort completed in 8.31e-05 seconds.
Sorting with Bubble Sort completed in 0.0005212 seconds.
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a2-ensf694\ex_E>
```

Part II – Complexity Analysis

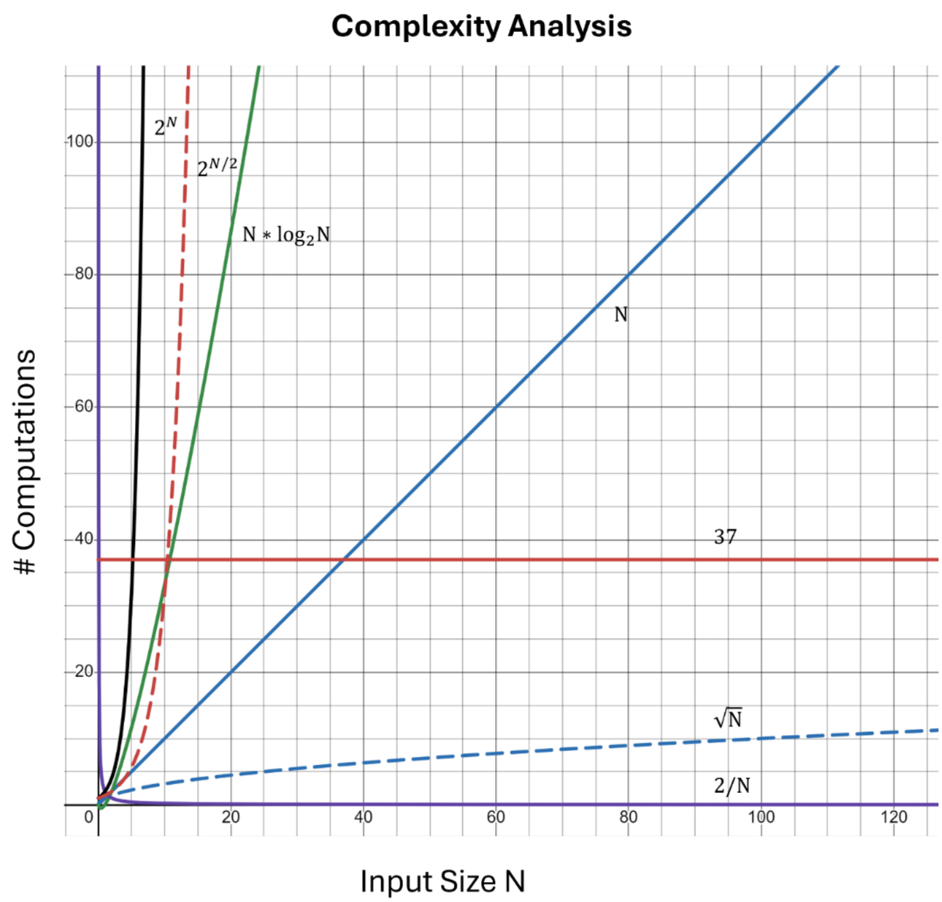
Exercise A

In order of performing best to worst based on their growth rate:

1. $\mathbf{2/N}$ – This is O(1/N). The growth rate decreases exponentially as N increases, approaching 0. For large N, it grows very slowly.

2. **37** – This is O(1). It is a constant, meaning its growth rate does not change regardless of N. It remains fixed and performs better than O(√N) for all N > 1369.
3. $\sqrt{\mathbf{N}}$ – This is O($\sqrt{\mathrm{N}}$). The growth rate is sublinear, i.e., it's faster than a constant but slower than linear.
4. **N** – This is O(N). It has linear growth, so it increases proportionally with N.
5. $\mathbf{N} * \mathbf{log_2 N}$ – This is O(NlogN). The growth rate is linear times a logarithmic factor, making it grow slightly faster than linear but still much slower than polynomial growth rates.
6. $\mathbf{2^{N/2}}$ – This is O($2^{\mathrm{N}/2}$). The growth rate is exponential, but slower than O(2^{N}) because the exponent is halved.
7. $\mathbf{2^N}$ – This is O(2^{N}). It has exponential growth, so it doubles with each increment of N. This is the fastest-growing rate among the given expressions.

The analysis can be visualized in the plot below:



Exercise B

```
(1)
sum = 0;
for( i = 0; i < n; ++i )
    ++sum;
```

initialization of sum: 1 time

for loop from i=0 to i<n: $n + n+1 = 2n+1$ times

summation statement of sum: n times

$1 + 2n+1 + n = 3n + 2 \quad \therefore O(n)$

```
(2)
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        ++sum;
```

initialization of sum: 1 time

for loop from i=0 to i<n: $n+n+1 = 2n+1$ times

for loop from j=0 to j<n: $n(n+1) + n^2 = 2n^2 + n$ times

summation statement of sum: n^2 times

$1 + 2n+1 + 2n^2 + n + n^2 = 3n^2 + 3n + 2 \quad \therefore O(n^2)$

```
(3)
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n * n; ++j )
        ++sum;
```

initialization of sum: 1 time

for loop from i=0 to i<n: $n+1+n = 2n+1$ times

for loop from j=0 to j<n²: $n^2(n+1) + n^2 = n^3 + 2n^2$

summation statement of sum: n^3 times

$1 + 2n+1 + n^3 + 2n^2 + n^3 = 2n^3 + 2n^2 + 2n + 2 \quad \therefore O(n^3)$

```
(4)
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < i; ++j )
        ++sum;
```

initialization of sum. 1 time

for loop from i=0 to i<n: $2n+1$ times

for loop from j=0 to j<i: $\frac{n(n+1)}{2} + \frac{n^2}{2} = \frac{2n^2+n}{2}$

summation statement of sum: $\frac{n^2}{2}$ times

$1 + 2n+1 + n^2 + \frac{n}{2} + \frac{n^2}{2} = \frac{3}{2}n^2 + \frac{5}{2}n + 2 \quad \therefore O(n^2)$

(5)

```
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < i; ++j )
        for( k = 0; k < j; ++k )
            ++sum;
```

initialization of sum: 1 time

for loop from i=0 to i<n: 2n+1 times

for loop from j=0 to j<i: $\frac{n(n+1)}{2} + \frac{n^2}{2} = \frac{2n^2+n}{2}$

for loop from k=0 to k<j: $\frac{n(n+1)(n+2)}{6} = \frac{(n^2+n)(n+2)}{6} = \frac{n^3+3n^2+2n}{6}$

summation statement of sum: $\frac{n^3}{6}$ times

$$1 + 2n+1 + n^2 + \frac{n}{2} + \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3} = \frac{n^3}{6} + \frac{3n^2}{2} + \frac{17}{6}n + 2$$

$\therefore O(n^3)$

(6)

```
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        for( k = 0; k < n; ++k )
            ++sum;
```

initialization of sum: 1 time

for loop from i=0 to i<n: n+1+n = 2n+1 times

for loop from j=0 to j<n: $n(n+1) + n^2 = 2n^2 + n$

for loop from k=0 to k<n: $n^2(n+1) + n^3 = 2n^3 + n^2$

summation statement of sum: n^3

$$1 + 2n+1 + 2n^2+n + 2n^3+n^2 + n^3 = 3n^3 + 3n^2 + 3n + 2$$

$\therefore O(n^3)$

Order of performing best to worst based on their growth rate:

(1) > (4) > (2) > (5) > (3) > (6)

Best Worst