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

A screenshot of a computer

Description automatically generated

## 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

 \*/

#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 outputA screenshot of a computer code Description automatically generated

## 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:

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

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:

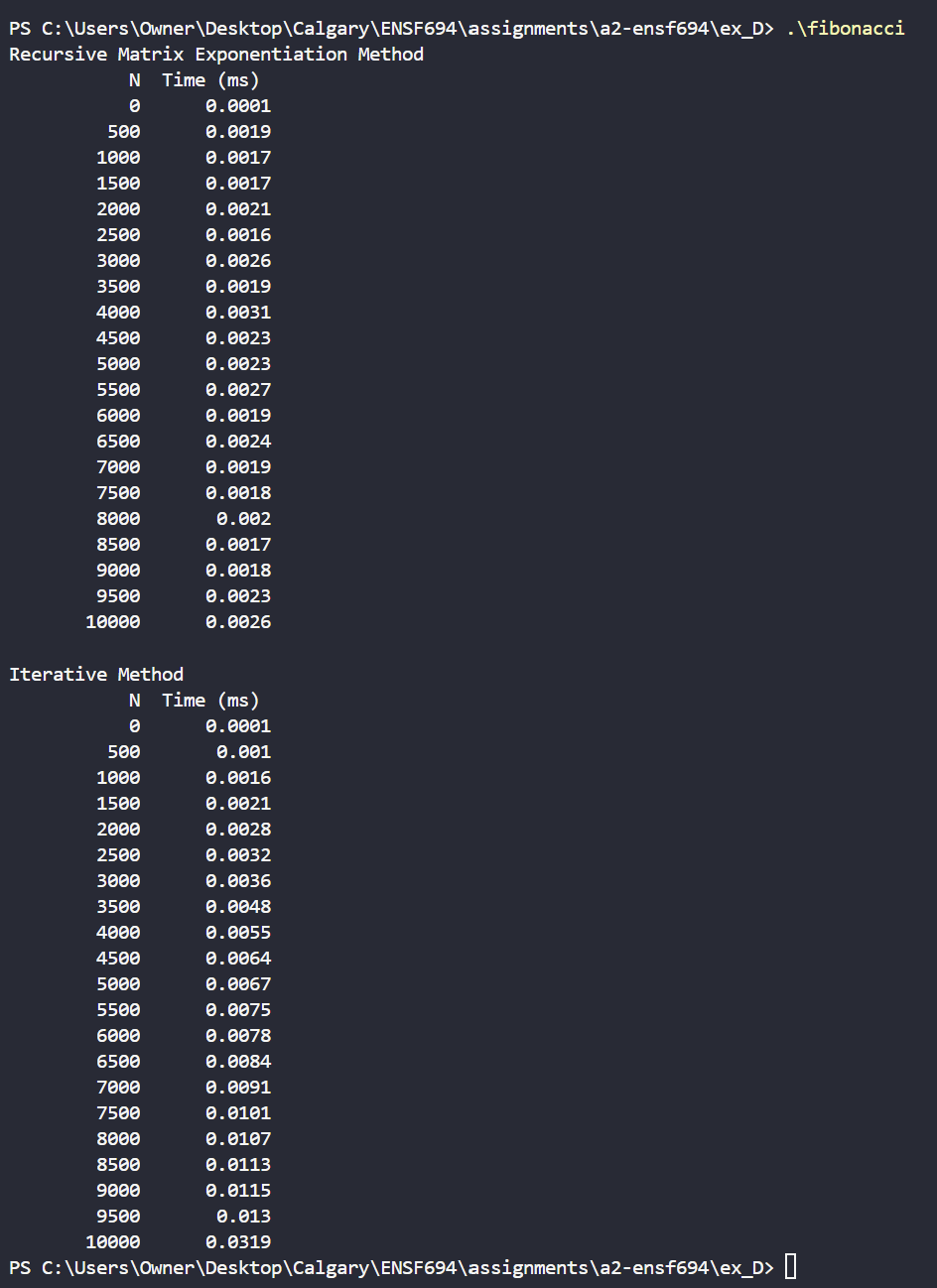
A screen shot of a graph

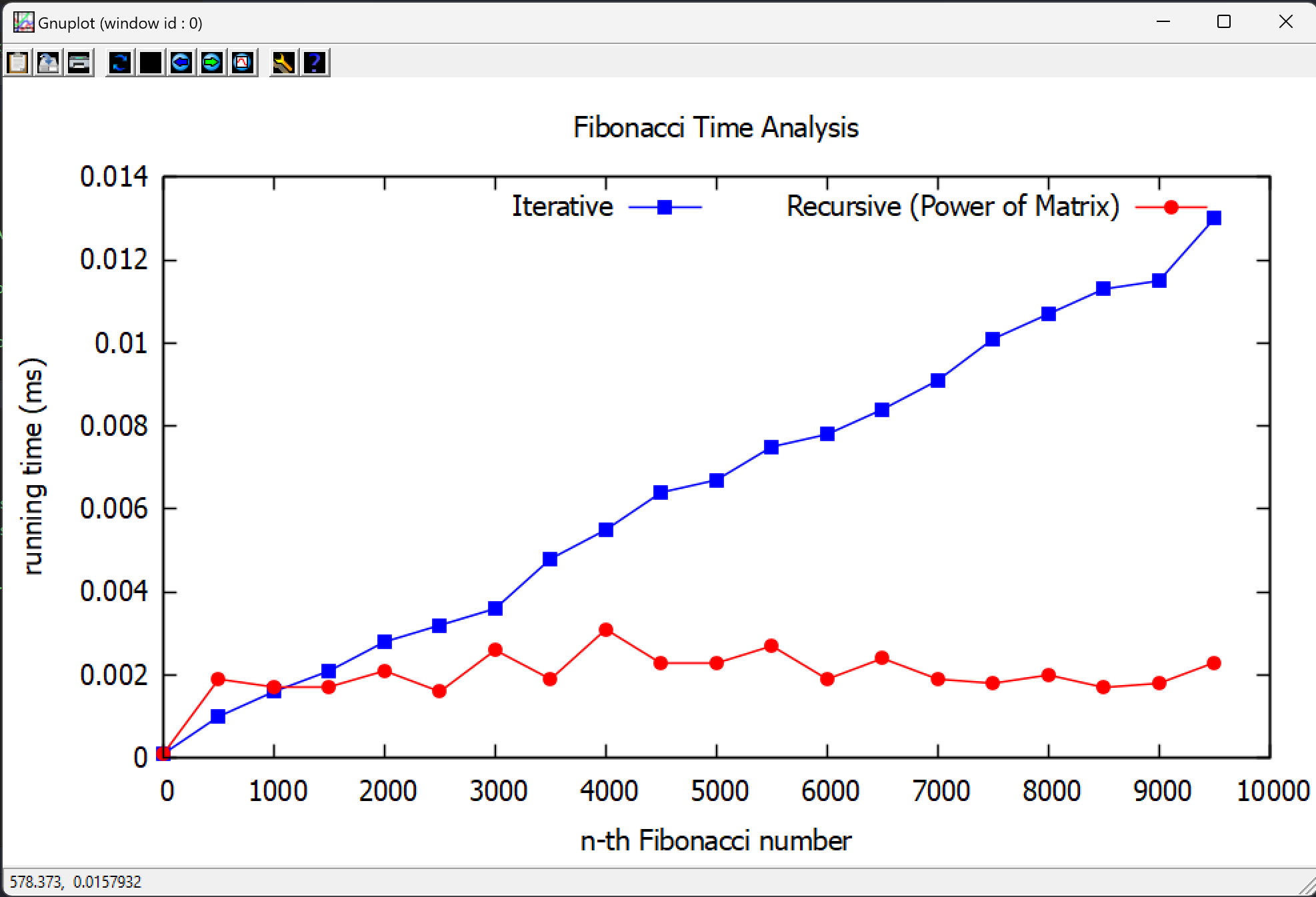
Description automatically generated

A screen shot of a graph

Description automatically generated

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)):





## 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

A screenshot of a computer program

Description automatically generated

# Part II – Complexity Analysis

## Exercise A

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

1. – 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. – This is O(). 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. – 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. – This is O(). The growth rate is exponential, but slower than O() because the exponent is halved.
7. – This is O(). 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:

A graph of a function

Description automatically generated

## Exercise B

A black and white screen with blue text

Description automatically generated

A black and white screen with blue text

Description automatically generated

A screen shot of a computer

Description automatically generated

A close-up of a computer screen

Description automatically generated

A blackboard with a black and white background with a black and white background with a black and white background with a black and white background with a black and white background with a red circle and a

Description automatically generated

A screen shot of a computer screen

Description automatically generated

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

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

Best Worst