Laboratory Exercise 01 Part 02

Interpolating the elevations into a higher resolution digital elevation matrix M given a lower resolution digital elevation matrix N

Kurt Brian Daine B. Punzalan

March 2, 2023

Table of Contents

Introduction

At the heart of programming lies problem-solving, which is the primary focus of a programmer's work. As such, there are a lot of interpretation and ways on how to solve a given problem. With this flexibility for solution comes the effectiveness of the program. Thus, there is a need to analyze how complex the code is and how it would be beneficial for us programmers that the algorithm implemented are efficient.

An algorithm is defined as a systematic procedure for a set of instructions to generate the desired output [3]. An algorithm that is correct is considered good, but one that is both correct and efficient is considered great [1].

There are several metrics on how to analyze the effectiveness of an algorithm, two of which are the running time and the time complexity. Although slightly different, running time refers to the actual amount of time it takes for an algorithm to complete when executed on a particular input, while time complexity provides a method for estimating how an algorithm's execution time will scale as the size of the input increases [2]. It is typically expressed as the Big O notation. Analyzing our programs helps us make well-informed choices on the algorithm to use in a specific problem.

In this exercise, we were tasked to run our program on interpolating elevations of a matrix and observe the running time as well as the time complexity of our program.

Objectives

The objectives of this exercise are as follows:

1. To run the source code on interpolating elevations of a matrix given a set of inputs;
2. To examine and evaluate the running time and time complexity of the program; and
3. Identify factors that might affect the running time and complexity of the program.

Methodology

This exercise was a continuation of the previous exercise on interpolating elevation of a lower resolution matrix N into a higher resolution matrix M. Python programming language was used in the previous exercise and Federal Communications Commission (FCC) as the method of interpolation.

In the program, an n by n matrix was initialized where n is a non-zero integer divisible by 10. All rows and columns that are both divisible by 10 were filled with randomized values from 1 to 1000 which served as the basis for the FCC.

The running time of the program from the initialized until the filled matrix was calculated. Three runs were made for each input. This was repeatedly done for varying input sizes of 100-20000.

The program was ran in the laboratory PC with Ubuntu 22.04.2 as its operating system having an Intel Core i7-8700 processor with 16.0GB of RAM and clock rate of 3.2GHz.

Results and Discussions

The source code for this program has a loop within a loop which indicates that it runs in a quadratic time complexity or $O(n^2)$. The program almost already follows the optimized code given during laboratory hours so there was no modification in the source code. The tabulation of elapsed time and calculation of the average time and complexity succeeded.

Table 1: Average Runtime and Complexity of Interpolating Elevations Program

n	Time Elapsed (seconds)			Average Runtime (seconds)	Complexity
	Run 1	Run 2	Run 3		
100	0.0244	0.0058	0.0242	0.0182	0.0182
200	0.0465	0.0179	0.0462	0.0369	0.0728
300	0.0381	0.0513	0.0387	0.0427	0.1638
400	0.0695	0.0716	0.0646	0.0686	0.2912
500	0.1002	0.1003	0.1083	0.103	0.455
600	0.1559	0.1444	0.14945	0.15	0.6552
700	0.1931	0.2108	0.1976	0.2005	0.8918
800	0.2561	0.2591	0.257	0.2574	1.1648
900	0.3218	0.3175	0.3241	0.3212	1.4742
1000	0.3966	0.3962	0.4011	0.398	1.82
2000	1.6422	1.9335	1.6012	1.7257	7.28
4000	6.2944	6.3057	6.629	6.4097	29.12
8000	25.0845	25.085	25.1549	25.1082	116.48
16000	100.4151	100.07	100.3573	100.2808	465.92
20000	157.6051	165.4293	169.107	164.0472	728

Since Python was used in this exercise, it was expected to execute more slowly than in C. During the 100-1000 range, all average time was less than a second. Inputting 20 000 in the program, on the other hand, took a toll on its average runtime, wherein it took 169 seconds to finish.

The complexity of interpolating the point's elevation of a n x n square matrix was calculated using the formula:

$$(current\_input - first\_input)^2 \cdot first\_average\_run\_time$$

where current_input is the current n size, first_input is the first n size (100, in this case), and first_average_run_time is the average run time of the program when n is 100. first_average_run_time in input size 100 also served as its complexity.

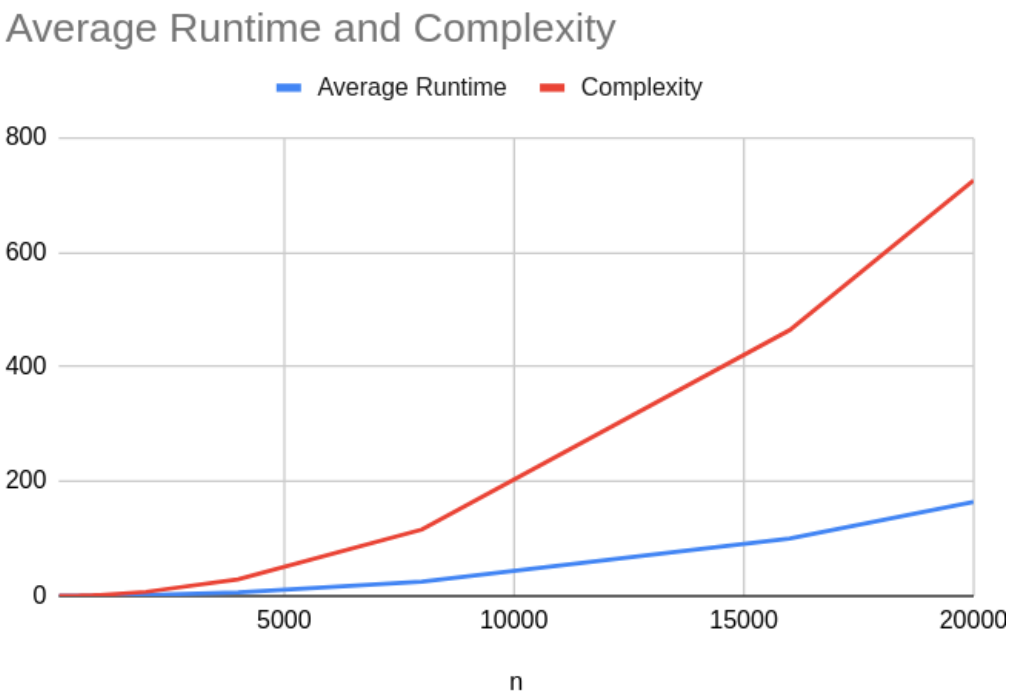


Figure 1: Input Size VS Average Runtime and Time Complexity

Figure 1 shows the graph of input size n versus the calculated average run time of as well as n versus time complexity of the program. It was observed

that the two lines agree, at least in their forms. As the input size gets larger, so do both the run time and the time complexity.

Conclusion

The running time of a program as well as its complexity depends on factors such as the input size, where it was shown that as the number of inputs increases, the longer it takes for an algorithm to finish; algorithm design, which includes the number of inner loops (and inner loops) in program; programming language used, wherein languages such as Python takes a while to process results unlike C; and even the hardware component.

As a programmer, it is our duty to optimize our code and apply our knowledge, specifically in the use of Big O notation, to make our code simpler and run faster. In order to do so, we can use better algorithms, where multiple inner loops are avoided as much as possible. Eliminating unnecessary print statements would also do the trick. In my case, the printing of the matrix also added up to the running time of the program. Lastly, choosing the appropriate data structure for the program might be beneficial.

List of Collaborators

These are the individuals who have collaborated on the program:

1. **Van Paul Dayag**: calculation of time complexity

References

[1] Khan Academy. Measuring an algorithm's efficiency, n.d.

[2] Great Learning Team. Time complexity: What is time complexity & its algorithms? *Great Learning Blog: Free Resources What Matters to Shape Your Career!*, November 7 2022.

[3] Shivam Upadhyay. What is an algorithm? characteristics, types and how to write it, 2023.

Appendices

Appendix A: Source Code

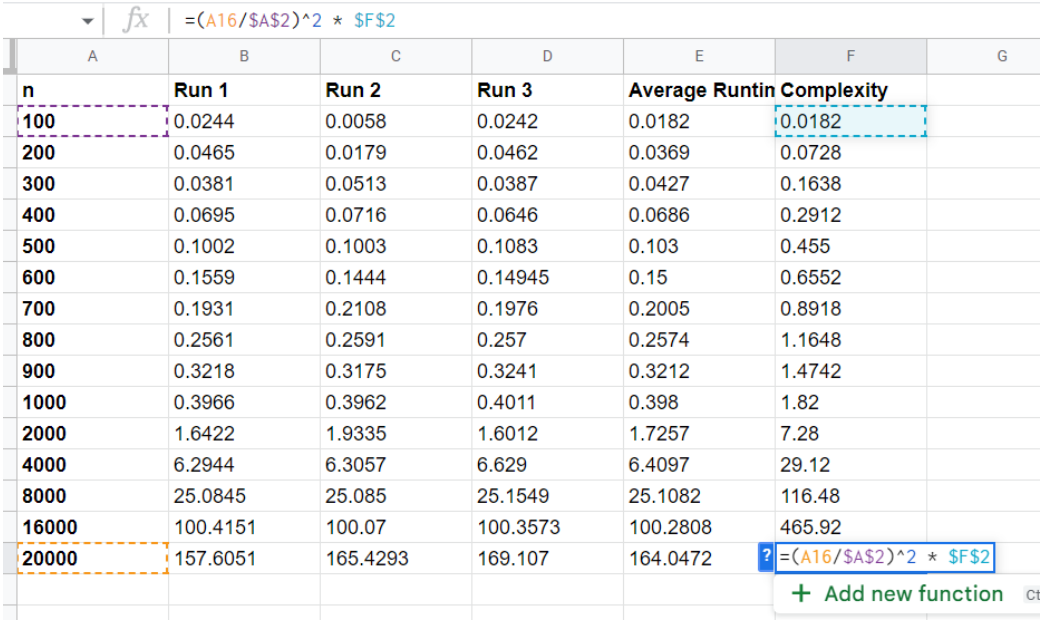
```
# Punzalan, Kurt Brian Daine B. Punzalan
# 2020-00772
# CMSC 180 - T-6L
# Exercise 01

import os
import random
import time

os.system("cls")

#! FUNCTIONS
def printMatrix(M, n):
    for i in range(n):
        for j in range(n):
            print(M[i][j], end="\t")

        print()

def randomizeDivisible(M, n):
    # randomize values of all divisible by 10
    for row in range(n):
        for col in range(n):
            if (row % 10 == 0) and (col % 10 == 0):
                M[row][col] = float(random.randint(1,1000))

    return M

def terrain_inter_row(M, n, row):
    for index in range(n):
        if M[row][index] == 0:
            x = index # provide x based on the x-coordinate
```

```
33
34              # apply the formula for FCC
35              M[row][index] = round((y1 + ((x-x1)/(x2-x1)) * (
    y2-y1)), 2)
36          else:
37              # print(index, "index")
38              x1 = index
39              x2 = index + 10
40
41              try:
42                  y1 = M[row][x1]
43                  y2 = M[row][x2]
44              except:
45                  pass
46
47      return M
48
49
50  def terrain_inter_col(M, n, col):
51      for index in range(n):
52          if M[index][col] == 0:
53              x = index
54              M[index][col] = round((y1 + ((x-x1)/(x2-x1)) * (
    y2-y1)), 2)
55
56          else:
57              x1 = index
58              x2 = index + 10
59
60              try:
61                  y1 = M[x1][col]
62                  y2 = M[x2][col]
63              except:
64                  pass
65
66      return M
```

```
67
68 #! MAIN
69
70 # user input
71 n = int(input("Input: ")) + 1
72
73 # matrix
74 M = [[0 for column in range(n)]for row in range(n)]
75 M = randomizeDivisible(M, n)
76
77 #! FCC
78 # calculate
79 time_before = time.time()
80
81 # fill the rows whose columns are divisible by 10
82 row = 0
83 while row < n:
84     M = terrain_inter_row(M, n, row)
85     row += 10
86
87
88 # fill the columns whose rows are divisible by 10
89 col = 0
90 while col < n:
91     M = terrain_inter_col(M, n, col)
92     col += 10
93
94
95 # fill remaining rows (inner box)
96 for row in range(1, n-1):
97     if row % 10 == 0:
98         continue
99     M = terrain_inter_row(M, n, row)
100
101 # for i in range(n):
102 #     for j in range(n):
```

```
#          print(f"({i}, {j})", end="\t")
#      print()

# print()

# printMatrix(M, n)

time_elapsed = time.time() - time_before

print("time elapsed:", time_elapsed)
```

Appendix B: Complexity Formula and Table



=(A16/$A$2)^2 * $F$2

	A	B	C	D	E	F	G
	n	**Run 1**	**Run 2**	**Run 3**	**Average Runtin**	**Complexity**	
	100	0.0244	0.0058	0.0242	0.0182	0.0182	
	200	0.0465	0.0179	0.0462	0.0369	0.0728	
	300	0.0381	0.0513	0.0387	0.0427	0.1638	
	400	0.0695	0.0716	0.0646	0.0686	0.2912	
	500	0.1002	0.1003	0.1083	0.103	0.455	
	600	0.1559	0.1444	0.14945	0.15	0.6552	
	700	0.1931	0.2108	0.1976	0.2005	0.8918	
	800	0.2561	0.2591	0.257	0.2574	1.1648	
	900	0.3218	0.3175	0.3241	0.3212	1.4742	
	1000	0.3966	0.3962	0.4011	0.398	1.82	
	2000	1.6422	1.9335	1.6012	1.7257	7.28	
	4000	6.2944	6.3057	6.629	6.4097	29.12	
	8000	25.0845	25.085	25.1549	25.1082	116.48	
	16000	100.4151	100.07	100.3573	100.2808	465.92	
	20000	157.6051	165.4293	169.107	164.0472	=(A16/$A$2)^2 * $F$2	

+ Add new function

Appendix C: PC Specifications