CSS342 Data Structures, Algorithms, and Discrete Mathematics I

Autumn 2017

Assignment 3: Algorithm Analysis
Due date: Friday 17 Nov

Goal

This programming assignment empirically analyzes] the worst-case performance of the Euclidean algorithm for finding the greatest common divisor of two nonnegative integers. You will code this program, plot its running times, and validate if your output is upper-bound to a given function.

The Euclidean Algorithm

The Euclidean algorithm computes the greatest common divisor (gcd) of two positive integers A and B, based on the observation gcd(A, B) is equivalent to gcd(B, A modulus B). In other words, the greatest common divisor of positive integers A and B is equal to the greatest common divisor of the integers B and the result of A modulus B. In other words, given two positive integers, their gcd is computed by repetitions of the following three statements: R = A % B; A = B; B = R; until B becomes 0.

Example for gcd(270, 192):

78 = 270 % 192; R = 78, A = 270, B = 192; Next step: gcd (B, R) = gcd (192, 78)

36 = 192 % 78; R = 36, A = 192, B = 78; Next step: gcd (B, R) = gcd (78, 36)

6 = 78 % 36; R = 6, A = 78, B = 36; Next step: gcd (B, R) = gcd (36, 6)

0 = 36 % 6; R = 0, A = 36, B = 6; This has shown that gcd (36, 6) = 6.

Going back up the chain: 6 = gcd (36, 6) = gcd (78, 36) = gcd (192, 78) = gcd (270, 192);

Therefore, gcd(270, 192) = 6;

Another way to express this example: to find the gcd of 270 and 192,

gcd(270,192) = gcd(192,78) = gcd(78,36) = gcd(36,6) = gcd(6,0) = 6

That is, gcd(270, 192) = 6.

https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm (Links to an external site.)

The worse-case performance is $log_{3/2}2n/3$ modulus operations required for computing the gcd of two integers in the range 1 to n, where $n >= 8$.

Statement of Work

Part 1

1. Write a program that reads the integer *n (n* should be larger than 8) and measures the maximum number of modulus operations to compute the gcd of two integers in the range 8 to *n*.

   Your program must print out the following data:

   (1) at each *i* from 8 until n, print the combination of two integers a, b within 1 through i, that incurs the maximum number of modulus operations. In other words: for every b between 8 and 15, there will be a set of corresponding a's going between 1 and b.

   (2) the gcd of these two integers a and b; and

   (3) the modulus operations to obtain this gcd of a and b (a and b are going to be every combination of two integers up until I, then you will print the combination of two integers that took the most amount of modulus operations)

   For example, if n=15;

   then you have to print out the followings:

   ==========

   At i=8; gcd (5,8) = 1 took 4 modulus operations

   At i=9; gcd (5,8) = 1 took 4 modulus operations

   At i=10; gcd (5,8) = 1 took 4 modulus operations

   At i=11; gcd (5,8) = 1 took 4 modulus operations

   At i=12; gcd (5,8) = 1 took 4 modulus operations

   At i=13; gcd (8, 13) = 1 took 5 modulus operations

   At i=14; gcd (8, 13) = 1 took 5 modulus operations

   At i=15; gcd (8, 13) = 1 took 5 modulus operations

   ==========

2. Plot the maximum number of modulus operations as increasing *n* from 8 to 3,000 in a graph. You don't have to plot such number every increment of *n*. Choose an appropriate increment of *n* to view the growth of modulus operations.

3. Plot $log_{3/2}2n/3$ in the same graph and compare your empirical data with this worse-case function. Are your empirical outputs always below this Big-O? Use the following formula to compute $log_{3/2}2n/3$:

   $\log_a B = \log_{10} B / \log_{10} a$

   If you don't have a calculator that computes log_{10}, use the linux calculator.

Part 2

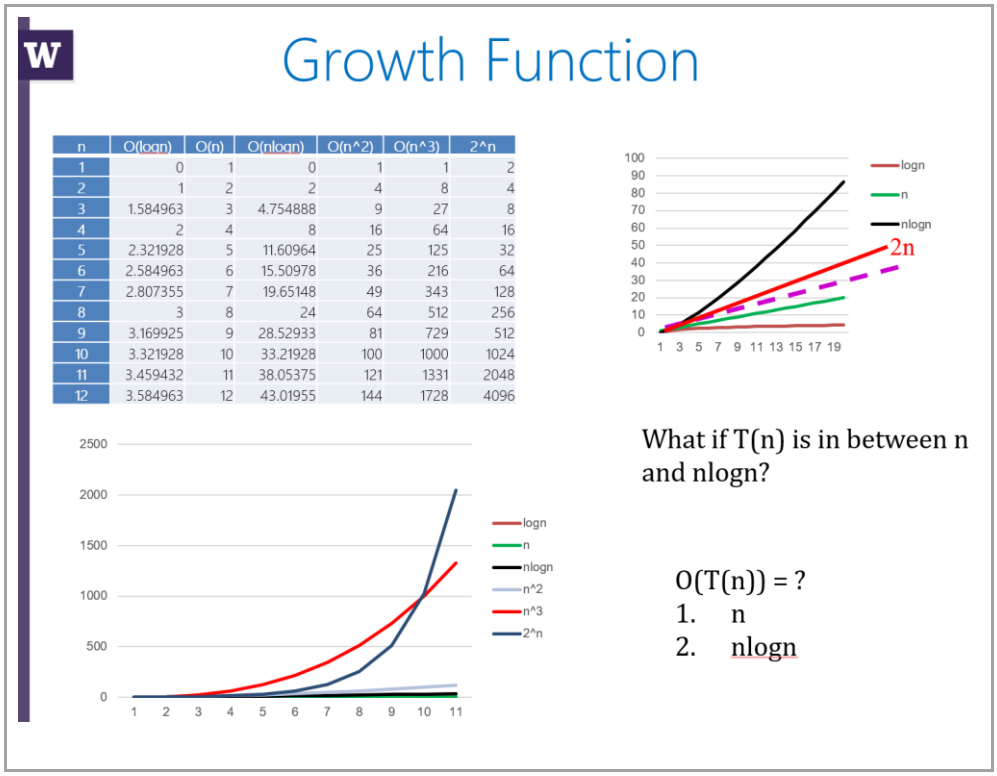
1. Measure the total running time of your program itself. For this purpose, you should use the following linux system function: #include <sys/time.h>
   int gettimeofday(struct timeval *tv, struct timezone *tz);
   The second argument to this function must be NULL. To understand the usage of this function, type *man gettimeofday* from the linux shell prompt.

2. Estimate the Big-O of your program using plots: record the actual growth time of your program with increasing n, and define it as T(n), i.e., time your program. In other words, time and collect the data for each i, for example:

   At **i=14**; **gcd (8, 13) = 1** took **5** modulus operations **time = 7ms** where the i is going to be the x-axis, and the time the y-axis. Then, **using the table and plots like the ones in the following picture**

   a. **Estimate the Big-Oh function of T(n)**
   b. **Define the estimated Big-O function as F(n).**
   c. **Verify your estimation is correct by showing T(n)/F(n) is (or nearly) constant, i.e., T(n)/F(n) <= c, where c is a constant.**

   Remember, one of the goals of this assignment is to experimentally determine the Big-Oh (and visualize the program's behavior through the plots).



n	O(logn)	O(n)	O(nlogn)	O(n^2)	O(n^3)	2^n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
3	1.584963	3	4.754888	9	27	8
4	2	4	8	16	64	16
5	2.321928	5	11.60964	25	125	32
6	2.584963	6	15.50978	36	216	64
7	2.807355	7	19.65148	49	343	128
8	3	8	24	64	512	256
9	3.169925	9	28.52933	81	729	512
10	3.321928	10	33.21928	100	1000	1024
11	3.459432	11	38.05375	121	1331	2048
12	3.584963	12	43.01955	144	1728	4096

What to Turn in

Clearly state in your code comments any other assumptions you have made.

What you have to turn in includes:

(1) euclid.cpp (i.e., your program including main()), and

(2) a separate report in .doc, .docx or .pdf format. This should include:

(2a) partial execution output in **one page**,

(2b) a graph plotting the maximum number of modules operations where *n = 8 ~ 3000, and* $log_{3/2}2n/3$

(2c) your mathematical Big-O estimation F(n) and T(n)/F(n) from part 2, and

(2d) a table and plot that verifies the Big-O estimation of your program. The table should include n, T(n), F(n), and T(n)/F(n). No program specification or algorithm explanation need to be added to your report.

NOTE: Do not submit any other file besides the ones specified above.

Grading Guide

1. Documentation (4pts)

Plot the max number of modulus as increasing n to 3,000
Yes(1pt) No(0pt)

Plot log(3/2)2n/3 in the same graph
Yes(1pt) No(0pt)

Estimate Big-O of your program from part 2 (T(n), F(n), T(n)/F(n))
Yes(1pt) No(0pt)

Verify your Big-O using a table and plot similar to the slide p13
Yes(1pt) No(0pt)

2. Correctness (12pts)
Compilation errors (0 pt)
Successful compilation (4 pt)
+ Correct Euclidean algorithm(2 pt)
+ Correct Output (2 pt)
+ Given n, examine all combination of gcd(a, b) where 1<=a, b<=i, and 8<=i<=n (2 pt)
+ Correct use of gettimeofday( ) (2 pt)

3. Program Organization (4pts)

Proper comments

Good (2) Poor(1) No explanations(0)

Coding style (proper indentations, blank lines, variable names, and non-redundant code)

Good (2) Poor(1) No explanations(0)