1/12/2023

ASSIGNMENT 1 CS 566 Spring 2023

The purpose of this assignment is to have you think through the structure of algorithms[[1]](#footnote-0) as well as computational complexity—particularly runtime efficiency.

Please retain all the gray text like this. Supply the material as indicated below. The total should be 4 pages or fewer, excluding code and figures and the instructions in gray. If you need to supply additional material, please use appendices, and reference them within the text. These will be read on an as-needed basis.

Note the evaluation criteria at the end: these are the qualities we are looking for.

# Part 1

On the left bank of a river are a ferryman, a goat, a boat big enough for four, and an equal number of Bluebeards and Zombies. The latter remain at peace except when one group outnumbers the other. When the ferryman is absent, Bluebeards and Zombies eat goats. Only the ferryman can operate the boat.

## Part 1.1

Describe an algorithm, in terms of [accumulating outcomes,](https://docs.google.com/document/d/1nFfWKRABLGmCj5_t7IjOI4WGG6M7DQLq8L5n-WNlK48/edit?usp=sharing) for getting everyone to the right bank without conflict. Don’t include a description of *how* to implement these outcomes (that’s Part 1.2). Use *n* as the total number of Bluebeards (and Zombies). Remember that (1) outcomes may introduce variable(s) and (2) we assume that the code implementing each outcome will also restore all prior outcomes. Be precise and concise.

You can assume that the boat is always in the same location as the ferryman, so there is no need to mention it. The formulation anticipated below divides the preconditions and postconditions in two, and uses three outcomes, but you are free to organize them differently if you wish.

Precondition 1: The number of Bluebeards and Zombies at left bank are both n, and the number of Bluebeards and Zombies at right bank must be same

Precondition 2: Ferryman and the goat must be move together

Postcondition 1: All the Bluebeards and Zombies are transported to the right bank

Postcondition 2: The goat should be safe

Outcome 1: A pair of Bluebeards and Zombies are at the right bank

Outcome 2: Ferryman and the goat should be at the right bank

Outcome 3: Pairs of Bluebeards and Zombies are at the right bank

## Part 1.2

Describe *how* to implement these outcomes (i.e., in order). A typical line has the form

*Ferry X, Y, and Z to the right.*

Sometimes it’s convenient to implement more than one outcome at a time. *Remember to justify that prior outcomes remain true* (i.e., that they accumulate). An example of a heading below is “Implementation of outcomes 1 and 2.”

Implementation of Outcome(s) Ferry goat to the right

Implementation of Outcome(s) Ferry goat to the left, and ferry goat, 1 Bluebeard and 1 Zombie to the right, and repeat this process util all the Bluebeards and Zombies are on the right bank

Implementation of Outcome(s) Ferry goat and ferryman to the right

## Part 1.3

Calculate and explain the efficiency of your algorithm precisely.

The efficiency of this algorithm is O(n).

The number of step to ferry them accross river is proportional to the number of Bluebeards and Zombies, so when the number of Bluebeards and Zombies increase, the step to ferry them accross river will also be increase at the same rate.

# Part 2

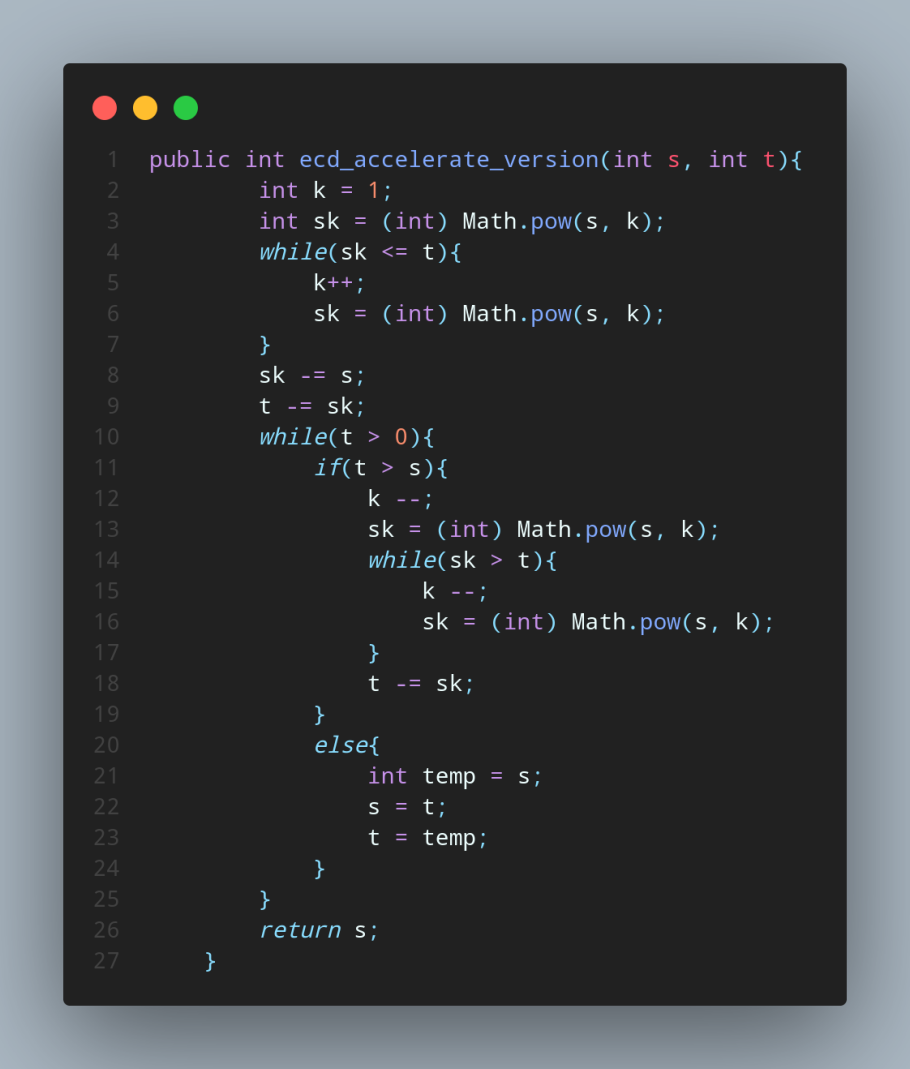
Implement the following accelerated version of Euclid’s algorithm for gcd(s, t) with precondition 0 < s <= t. Use the programming conventions outlined in the [online materials](https://docs.google.com/document/d/1nFfWKRABLGmCj5_t7IjOI4WGG6M7DQLq8L5n-WNlK48/edit?usp=sharing) and the format below. Include preconditions, postconditions, and outcomes. Recall that outcomes must accumulate, so prior outcomes must be restored. Favor loops of the form *while(!outcome)* because they are readily verifiable. Explain efficiency as requested below.

Increment k until sk <= t and sk+1 > t, retaining s2, s3, … sk. Diminish t by sk. If (the new) t is still the larger of s and t, iterate down s2, s3, … sk starting with sk, repeating the process of finding the highest power of s to subtract from t. If the new t is the smaller of s and t, switch the roles.

For example, to find gcd(4, 250), s=4, and we consider 4, 42=16, 64, 256. We subtract 64 from 250. The new t is 186. We start by considering 256—which is too big to subtract from 256—then 64. We subtract 64 from 186, getting 124 for t, and repeat the process. So we go from gcd(4, 250) to gcd(4, 186) to gcd(4, 122) to gcd(4, 58) to gcd(4, 42) to gcd(4, 26) to gcd(4, 10) to gcd(4, 6) to gcd(4, 2) to gcd(2, 2) = 2. (No switching of roles took place in this example.)

## Part 2.1 Implementation (preferably Python, but Java is OK)

If you don’t use the programming conventions outlined in the online materials, explain substantively why yours are easier to verify and analyze.



## Part 2.2 Efficiency: “n”

What is “n” in this case? Explain.

n is the efficiency for the algorithm to find the greatest common divisor by fewer number of iterations. So the efficiency if this algorithm is determined by the value of n.

## Part 2.3 Best case time with explanation

Caution: don’t overthink this.

The best case is O(1) when the s is already the greatest common divisor.

## Part 2.4 Worst case exploration

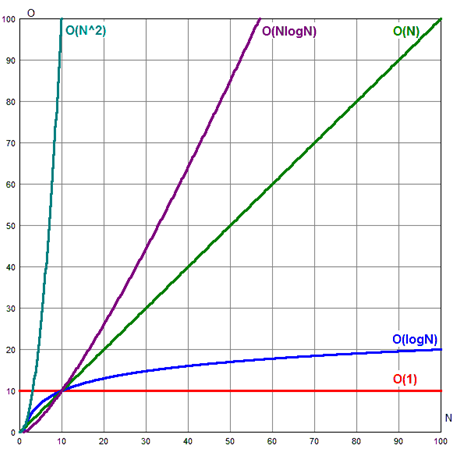
Take a worst case for the Euclidean algorithm—gcd(2, n) where n is odd. *For this case* only, compare the time efficiency of the classical Euclidean algorithm with the efficiency of the algorithm described for Part 2.

The time complexity for the worst case for the classical Euclidean algorithm—gcd(2, n) where n is odd is O(log(n))

For part 2

|  |  |
| --- | --- |
| Code | Big-oh |
| public int ecd\_accelerate\_version(int s, int t){ |  |
| int k = 1; | O(1) |
| int sk = (int) Math.pow(s, k); | O(1) |
| while(sk <= t){ | O(n) |
| k++; | O(1) |
| sk = (int) Math.pow(s, k);} | O(1) |
| sk-=s; | O(1) |
| t-=sk; | O(1) |
| while(t > 0){ | O(n) |
| if (t > s){ | O(1) |
| k --; | O(1) |
| sk = (int) Math.pow(s, k); | O(1) |
| while(sk > t){ | O(n) |
| k --; | O(1) |
| sk = (int) Math.pow(s, k);} | O(1) |
| t -= sk;} | O(1) |
| else{ | O(1) |
| int temp = s; | O(1) |
| s = t; | O(1) |
| t = temp;}} | O(1) |
| return s;} | O(1) |
| For the code of Part 2, we can know the big-oh notation is O( which is equal to O() | |

So, the time complexity for the worst case for the algorithm described in part 2 is O(n^2)



Compared to this two algorithm, according to the graph above, the big-oh graph of the classical one has the smaller slope than the graph for algorithm in Part 2.

As we all know that the lower slope represents faster, so, the classical one is faster than the algorithm described in part 2 in the worst case.

# Evaluation



# Appendix 1 (if needed)

# Appendix 2 (if needed)

1. Vardi [points out](https://cacm.acm.org/magazines/2012/3/146261-what-is-an-algorithm/fulltext?mobile=false) that there are two basic ways to view an algorithm, analogously to the way that light can be thought of as a wave as well as a particle. The commonly understood view is as code (whether executable or pseudocode). The other way is as a sequence of intersecting states—accumulating outcomes. Both ways inform our understanding. [↑](#footnote-ref-0)