1/18/2023

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 left bank and right should have the number *n* as total numbers of zombies and Bluebeards, if not, they will fight.And the goat should stay with ferryman to keep it safe.

Precondition 2: The boat can only load 4 units at a time and only ferryman can operate the boat.

Postcondition 1: Move all units to the right bank

Postcondition 2: Keep goat safe, no conflict between zombies and Bluebeards

Outcome 1: Goat stay with ferryman all the times

Outcome 2: Move a pair of zombie and Bluebeard to the right side(n-2).

Outcome 3: Do the same thing repeatedly (Move a pair of zombie and Bluebeard to the right side(n-2-2-........)) until move all zombies and Bluebeards to the right side

## 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) :

Keep goat safe and move it to the right side

Implementation of Outcome(s) :

Move all zombies and Bluebeards to the right side. Achieve the question request

## Part 1.3

Calculate and explain the efficiency of your algorithm precisely.

The efficiency of the algorithm is O(n) because to need Move a pair of zombie and Bluebeard to the right side repeatedly, which mean each time we move the zombie and Bluebeard, the number of n increase. In other word, the times of moving and n are directly proportional. So this is an O(n) algorithm

# 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 250—then 64. We subtract 64 from 186, getting 122 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. There was a switching of roles in the digits in the last step.

## 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.

def Ecd\_accelerate\_version(s, t):

k = 1

sk = pow(s, k)

while sk <= t:

k += 1

sk = pow(s, k)

sk -= s

t -= sk

while t > 0:

if t > s:

k -= 1

sk = pow(s, k)

while sk > t:

k -= 1

sk = pow(s, k)

t -= sk

else:

s, t = t, s

return s

## Part 2.2 Efficiency: “n”

What is “n” in this case? Explain.

n is the efficiency of the algorithm to find the greatest common divisor. Which mean n is the number of iterations. So the efficiency of algorithm is depend on the number of iterations, in other word, it is depend on the value of n. From the example , we can see we need to increment k unit s^k <=t ans s^(k+1) >=t. Like gcd(4,250), 4^4=256,m it is bigger than t. So the best k is 3 and 4^3=64. Then we can go from gcd(4,250), to gcd(4,186), to gcd(4,122),to gcd(4,58). Because 58 is less than 64, so we should us 4^2=16. Then we can get from gcd(4,58) to gcd(4,42), to gcd(4,26), to (4,10). For the same reason, we can get from gcd(4,10) to gcd(4,4). Then we can get gcd(2,2)=2. The total step should be 9(if add the progress of finding the s^(k+1) >=t,the total step should be 14). And there is one point we should notice is the value of n is depends on s and t. And we should notice is the value of n is depends on s and t.

## Part 2.3 Best case time with explanation

Caution: don’t overthink this.

Best case time:O(1)

## 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.

To find the gcd(2,n) where n is odd, which mean gcd(2,n)=1. For example:when n=15(2^4-1=15). Then we can get gcd(2,15), Then we can get gcd(2,7) becasue k=3, then we do the same progress as 2.2. We can finally have gcd(2,15)->gcd(2,7)->gcd(2,3)->gcd(2,1).The total step should be 4. I have the same k value as 2.2 but when n is odd,it has less steps. The time complexity for this case based on the classical Euclidean algorithm—gcd(2, n) where n is odd is O(log(n))

Code for part two:

def Ecd\_accelerate\_version(s, t):

k = 1 O(1)

sk = pow(s, k) O(1)

while sk <= t: O(n)

k += 1 O(1)

sk = pow(s, k) O(1)

sk -= s O(1)

t -= sk O(1)

while t > 0: O(n)

if t > s: O(1)

k -= 1 O(1)

sk = pow(s, k) O(1)

while sk > t: O(n)

k -= 1 O(1)

sk = pow(s, k) O(1)

t -= sk O(1)

else: O(1)

s, t = t, s O(1)

return s O(1)

For this algorithm, its time efficiency is O(n^2)

Compared to the time efficiency O(n^2) and O(log(n)), O(log(n)) time efficiency is always better than the O(n^2). So we can say Classical Euclidean algorithm is better than the algorithm described for Part 2

# 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)