CSE 3101 Design and Analysis of Algorithms
Practice Test for Unit 1
Loop Invariants and Iterative Algorithms

Jeff Edmonds

First learn the steps. Then try them on your own. If you get stuck only look at a little of the answer and then try to continue on your own.

1. **Tiling Chess Board:** You are given a $2n$ by $2n$ chess board. You have many tiles each of which can cover two adjacent squares. Your goal is to place non-overlapping tiles on the board to cover each of the $2n \times 2n$ tiles except for to top-left corner and the bottom-right corner. Prove that this is impossible. To do this give a loop invariant that is general enough to work for any algorithm that places tiles. Hint: chess boards color the squares black and white.

2. **Longest Contiguous Increasing Subsequence (LCIS):** The input consists of a sequence $A[1..n]$ of integers and we want to find a longest contiguous subsequence $A[k_1..k_2]$ such that the elements are *strictly* increasing. For example, the optimal solution for $[5, 3, 1, 3, 7, 7, 9, 8]$ is $[1, 3, 7]$.

   (a) Provide a tight lower bound on the running time for this problem. Prove that your answer is indeed a tight lower bound.

   (b) Specify an appropriate loop-invariant, measure of progress and exit condition for an iterative solution to this problem. No explanations are necessary.

   (c) Provide concise pseudo-code for an algorithm LCIS(A,n) that returns the indices k_1, k_2 of the LCIS of $A[1 \ldots n]$ uses the loop-invariant, measure of progress and exit condition you specified above. Assume $n \geq 1$.

   (d) Provide an informal argument for how your code is able to maintain the loop invariant. A formal proof is not required.

   (e) What are the other key steps in proving your algorithm correct? List and provide a concise proof for each.

3. $d + 1$ **Colouring:** Given an undirected graph G such that each node has at most d neighbors, colour each node with one of $d + 1$ colours so that for each edge the two nodes have different colours. Hint: Don't think too hard. Just colour the nodes. What loop invariant do you need?
   Hint: All it will say is that you have not gone wrong yet.
   Give all the steps done in class to develop this iterative algorithm.

In Sides or Videos

4. (Answer in slides)
   Iterative Cake Cutting: The famous algorithm for fairly cutting a cake in two is for one person to cut the cake in the place that he believes is half and for the other person to choose which "half" he likes. One player may value the icing more while the other the cake more, but it does not matter. The second player is guaranteed to get a piece that he considers to be worth at least a half because he choose between two pieces whose sum worth for him is at least a one. Because the first person cut it in half according to his own criteria, he is happy which ever piece is left for him. Our goal is write an iterative algorithm which solves this same problem for n players.

   To make our life easier, we view a cake not as three dimensional thing, but as the line from zero to one. Different players value different subintervals of the cake differently. To express this, he assigns some numeric value to each subinterval. For example, if player p_i's name is written on the subinterval

$[\frac{i-1}{2n}, \frac{i}{2n}]$ of cake then he might allocate a higher numeric value to it, say $\frac{1}{2}$. The only requirement is that the sum total value of the cake is one.

Your algorithm is only allowed the following two operations. In an evaluation query, $v = Eval(p, [a, b])$, the algorithm asks a player p how much v he values a particular subinterval $[a, b]$ of the whole cake $[0, 1]$. In a cut query, $b = Cut(p, a, v)$, the protocol asks the player p to identify the shortest subinterval $[a, b]$ starting at a given left endpoint a, with a given value v. In the above example, $Eval(p_i, [\frac{i-1}{2n}, \frac{i}{2n}])$ returns $\frac{1}{2}$ and $Cut(p_i, \frac{i-1}{2n}, \frac{1}{2})$ returns $\frac{i}{2n}$. Using these the two player algorithm is as follows.

algorithm $Partition2(\{p_1, p_2\}, [a, b])$

$\langle$***pre−cond***$\rangle$**:** p_1 and p_2 are players.
$[a, b] \subseteq [0, 1]$ is a subinterval of the whole cake.

$\langle$***post−cond***$\rangle$**:** Returns a partitioning of $[a, b]$ into two disjoint pieces $[a_1, b_1]$ and $[a_2, b_2]$ so that player p_i values $[a_i, b_i]$ at least half as much as he values $[a, b]$.

begin
  $v_1 = Eval(p_1, [a, b])$
  $c = Cut(p_1, a, \frac{v_1}{2})$
  if($Eval(p_2, [a, c]) \leq Eval(p_2, [c, b])$) then
    $[a_1, b_1] = [a, c]$ and $[a_2, b_2] = [c, b]$
  else
    $[a_1, b_1] = [c, b]$ and $[a_2, b_2] = [a, c]$
  end if
  return($[a_1, b_1]$ and $[a_2, b_2]$)
end algorithm

The problem that you must solve is the following

algorithm $Partition(n, P)$

$\langle$***pre−cond***$\rangle$**:** P is a set of n players.
Each player in P values the whole cake $[0, 1]$ by at least one.

$\langle$***post−cond***$\rangle$**:** Returns a partitioning of $[0, 1]$ into n disjoint pieces $[a_i, b_i]$ so that for each $i \in P$, the player p_i values $[a_i, b_i]$ by at least $\frac{1}{n}$.

begin
...
end algorithm

(a) Can you cut off n pieces of cake, each of *size* strictly bigger than $\frac{1}{n}$, and have cake left over? Is it sometimes possible to allocated a disjoint piece to each player, each worth by the receiving player much more than $\frac{1}{n}$, and for there to still be cake left? Explain.

(b) Give **all** the required steps to describe this LI algorithm. (Even if you do not know how to do a step for this algorithm, minimally state the step.)

As a big hint to designing an iterative algorithm, we will tell you what the first iteration accomplishes. (Later iterations may do slightly modified things.) Each player specifies where he would cut if he were to cut off the first $\frac{1}{n}$ fraction of the $[a, b]$ cake. The player who wants the smallest amount of this first part of the cake is given this piece of the cake. The code for this is as follows.

  loop $i \in P$
    $c_i = Cut(p_i, 0, \frac{1}{n})$
  end loop
  i_{min} = the $i \in P$ that minimizes c_i
    $[a_{i_{min}}, b_{i_{min}}] = [0, c_{i_{min}}]$

As a second big hint, your loop invariant should include:

i. How the cake has been cut so far.

ii. Who has been given cake and how do they feel about it.

iii. How do the remaining players feel about the remaining cake.

5. (Answer in slides)
Sorted Matrix Search: Search a Sorted Matrix Problem: The input consists of a real number x and a matrix $A[1..n, 1..m]$ of nm real numbers such that each row $A[i, 1..m]$ is sorted from left to right and each column $A[1..n, j]$ is sorted from top to bottom. The goal is to determine if the key x appears in the matrix. Design and analyze an iterative algorithm for this problem that examines as few matrix entries as possible. Careful, if you believe that a simple binary search solves the problem. Later we will ask for a lower bound and for a recursive algorithm.

6. (Answer in slides)
Connected Components: The input is a matrix I of pixels. Each pixel is either black or white. A pixel is considered to be connected to the four pixels left, right, up, and down from it, but not diagonal to it. The algorithm must allocated a unique name to each connected black component. (The name could simply be 1,2,3,..., to the number of components.) The output consists of another matrix $Names$ where $Names(x, y) = 0$ if the pixel $I(x, y)$ is white and $Names(x, y) = i$ if the pixel $I(x, y)$ is a part of the component named i. The algorithm reads the matrix from a tape row by row as follows.

```
    loop y = 1...h
        loop x = 1...w
            ⟨loop−invariant⟩: ??
            if( I(x,y)=white )
                Names(x,y) = 0
            else
                ???
            end if
        end loop
    end loop
end algorithm
```

The image may contain spirals and funny shapes. Connected components may contain holes that contain other connected components. A particularly interesting case is the image of a comb consisting of many teeth held together at the bottom by a handle. Scanning the image row by row, one would would first see each tooth as a separate component. As the handle is read, these teeth would merge into one.

(a) Give the classic *more of the input loop invariant* algorithm. Don't worry about its running time.

(b) This version of the question is easier in that the matrix $Names$ need not be produced. The output is simply the number of connected black components in the image. However, this version of the question is harder in your computation is limited in the amount of memory it can use. For example, you don't remember pixels that you have read if you do not store them in this limited memory and you don't have nearly enough memory to store them all. The number of components may be Θ(the pixels) so you cant store all of them either. How little memory can you get away with?

(c) In this this final version, in addition to a small amount of fast memory, you have a small number of tapes for storing data. Data access on tapes, however, is quite limited. A tape is in one of three modes and cannot switch modes mid operation. In the first mode, the data on a tape is read forwards one data item at a time in order. The second mode, is the same except it is read backwards. In the third mode, the tape is written to. However, the command is simply $write(data)$ which appends this data to the end of the tape. Data on a tape cannot be changed.

All you can do is to erase the entire tape and start writing again from the beginning. An algorithm must consist of a number of passes. The first pass reads the input from the input tape one pixel at a time row by row in order. As it goes, the algorithm updates what is store in fast memory and outputs data in order onto a small number output tapes. Successive passes can read what was written during a previous pass and/or the input again. The last pass must write the required output $Names$ onto a tape. You want to use as little fast memory and as few passes as possible. For each pass clearly state the loop invariant.

7. (Answer in videos)
A *tournament* is a directed graph (see Section **??**) formed by taking the complete undirected graph and assigning arbitrary directions on the edges, i.e., a graph $G = (V, E)$ such that for each $u, v \in V$, exactly one of $\langle u, v\rangle$ or $\langle v, u\rangle$ is in E. A *Hamiltonian path* is a path through a graph that can start and finish any where but must visit every node exactly once each. Design an algorithm which finds a Hamiltonian path through it given any tournament. Because this algorithm finds a Hamiltonian path for each tournament, this algorithm, in itself, acts as proof that every tournament has a Hamiltonian path.

8. (Answer in videos)
An *Euler tour* in an undirected graph is a cycle that passes through each edge exactly once. A graph contains an Eulerian cycle iff it is connected and the degree of each vertex is even. Given such a graph find such a cycle.

In Lectures

9. Pointers (Answer in slides)
"And the last shall be first!"
(For each of A-K see the figure on the next page labeled with the corresponding letter. Here $n = 6$.)

A: Precondition; The input to this problem is a linked list of n nodes in which the last node points back to the first forming a circle. The variable *last* points at the "last" node in the list. The nodes are of type $Node$ containing a field $info$ and a field $next$ of type $Node$ which is to point at the next node in the list. The values $1, 2, \ldots, 6$ in figure are just to help you and cannot be used in the code.

B: Postcondition; The required output to this problem consists of the same nodes in the same circle, except each node now points at the "previous" node instead of the "next". In other words the pointers are turned from clockwise to counter clockwise. The variable *last* still points at the same node, but this "last" node has become "first".

C: Loop Invariant$_t$; Your algorithm for this problem will be iterative (i.e. a loop taking one step at a time). Your first task is to give the loop invariant. This is not done in words but by **drawing** what the data structure will look like after the algorithm has executed it's loop $t = 2$ times. Hint: Nodes with value 1 and 2 have their pointers fixed. Hint: You may need a couple of additional pointers to hold things in place. Give them as meaningful names as possible.

D: Loop Invariant$_{t+1}$; The next task in developing an iterative algorithm is:

- Maintain the Loop Invariant:
We arrived at the top of the loop knowing only that the Loop Invariant is true and the Exit Condition is not.
We must take one step (iteration) (making some kind of progress).
And then prove that the Loop Invariant will be true when we arrive back at the top of the loop.
$\langle loop{-}invariant_t\rangle$ & $not\,\langle exit{-}cond\rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant_{t+1}\rangle$

Towards this task, **draw** what the data structure will look like after the algorithm has executed it's loop one more time, i.e. $t+1 = 3$ times.

E: Code C to D; In space E, give me the code to change the data structure in figure C into that in figure D. This will be the code within the loop that makes progress while maintaining the loop invariant. Assume every variable and field is public.

F: Loop Invariant$_0$; The next task in developing an iterative algorithm is:

- Establish the Loop Invariant:
Our computation has just begun and all we know is that we have an input instance that meets the Pre Condition.
Being lazy, we want to do the minimum amount of work.
And to prove that it follows that the Loop Invariant is then true.
$\langle pre-cond \rangle$ & $code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$

Towards this task, **draw** what the data structure will look like when the algorithm is at the top of the loop but has has executed this loop zero times. Make it as similar as possible to the precondition so that minimal work needs to be done in G.

G: Code A to F; In space G, give me the code to change the data structure in figure A into that in figure F. This will be the initial code that establishes the loop invariant.

H: Loop Invariant$_n$; The next task in developing an iterative algorithm is:

- Obtain the Post Condition:
We know the Loop Invariant is true because we have maintained it.
We know the Exit Condition is true because we exited.
We do a little extra work.
And then prove that it follows that the Post Condition is true.
$\langle loop-invariant \rangle$ & $\langle exit-cond \rangle$ & $code_{post-loop} \Rightarrow \langle post-cond \rangle$

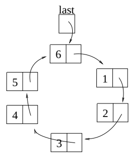
Towards this task, **draw** what the data structure will look like after the algorithm has executed it's loop n times. Make it as similar as possible to the postcondition so that minimal work needs to be done in J.

I: Exit Condition: What is the exit condition, i.e. how does your code recognize that it is in the state you drew in H, given that the algorithm does not know n (and did not count t) and does not know the values $1, 2, \ldots, 6$. Also be careful that your exit condition does not have you drop out at state D.

J: Code H to B; In space J, give me the code to change the data structure in figure H into that in figure B. This will be the final code that establishes the post condition.

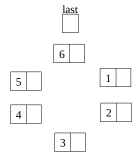
K: Give the complete code; Put all the code together into a routine solving the problem.

A: Precondition

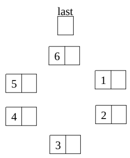


G: Code A to F

F: Loop Invariant$_0$

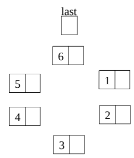


C: Loop Invariant$_t$

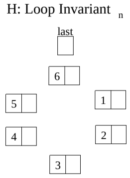


E: Code C to D

D: Loop Invariant$_{t+1}$



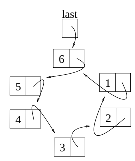
H: Loop Invariant$_n$



I: Exit Condition

J: Code H to B

B: Post Condition



K: Give the complete code

10. (Answer in slides)
Lake Problem:
You are in the middle of a lake of radius one. You can swim at a speed of one and can run infinitely fast. There is a smart monster on the shore who can't go in the water but can run at a speed of four. Your goal is to swim to shore arriving at a spot where the monster is not and then run away. If you swim directly to shore, it will take you 1 time unit. In this time, the monster will run the distance $\Pi < 4$ around to where you land and eat you. Your better strategy is to maintain the most obvious loop invariant while increasing the most obvious measure of progress for as long as possible and then swim for it. Describe how this works.

11. **Iterative 3 In A Row In An Infinite Line:**
The game board consists of a single line of squares that is infinitely long in both directions. Each iteration your opponent places a white piece and then you place a black piece. If a player gets three in a row he wins. Your primary goal is to stop him from doing this. If the game goes on forever then this is considered a tie.

Your game strategy will be to maintain the following loop invariants
After we each have played the same number of times the following are true.

(a) Every contiguous block of whites has a black on its left end.

(b) Every contiguous block of whites of length two or more also has a black on its right end.

(c) There are no contiguous blocks of white of length three or more.

(i.e. A block consisting of one white has a black to its left, a block with two whites has a black on both ends, and there are no blocks of length three or more.)

The exit condition will be that someone has won.
The post condition will be that you win.

Your task is to follow all the loop invariant steps outlined in the steps and then to prove that either the game goes on forever as a tie or you win.

Suppose your opponent has just placed a white. There are nine cases depending on whether there is a black, white, or blank on the left of this newly placed white and whether there is a black, white, or blank on the right. To help you, I group these nine into the five cases indicated in the following five lines. Then I group these further into three cases indicated by the black dots. In each of these five cases, state where you will place your black in response and in each of these three cases, prove that this maintains each of the three loop invariants.

- There is a blank to the left of where he placed his white.
  There is a black to the left.
- There is a white to the left and a blank to the right.
  There is a white to the left and a black to the right.
- There is a white to the left and a white to the right.

Is it possible to have an algorithm that guarantees you win even if you go first?

12. **Iterative n In A Row In An Infinite Line:**
The game board is an infinitely long line of squares. In a normal round, you will place a white and then your opponent a black. The goal is to build long contiguous blocks of your colour. We proved in test 1 that there is no algorithm that allows you to build a block longer than two. We will now develop an algorithm that builds arbitrarily long contiguous white blocks. To help, you will be given the advantage that every ten round you get to place a white but your opponent must skip his turn. (The regular expression is $((WB)^9W)^*$.)

It seemed at first to me that even if you got 100 times as many turns, that you could not build a contiguous block longer than 200, i.e. you put a 100, he caps one end by placing a black there, you place a second 100 on the other end, and he caps that end with a black. If you did manage to build a really long block, it would take you a while. Then in just two moves he can destroy it by capping both ends. Surprisingly, you can build arbitrarily long lines. (Funny though you can't just build a block that gets longer and longer arbitrarily. You must have the length you want to build in mind when you start.)

To make the algorithm easier, lets assume that the opponent is not trying to build a block of blacks. His only goal is to prevent you from building a long contiguous block of white by capping the ends of any block you are working on with a black. All blocks that we start will be far enough away from each other that they will never grow long enough to interact with each other. We will be cautious. If the opponent caps one of the ends of one of our blocks or even puts a black anywhere near that block, then we will not wait for him to cap the second end but will abandon that block completely. Hence, all we are concerned about is how many blocks we have that have no blacks near them and how long are these blocks are. The adversary each of his turns, being as greedy as he can be, will simply cap the first end of our currently longest such block. Given this, we will keep our blocks as close to being the same size as possible.

Input: The input to this problem is an integer n (say a billion).

Precondition: The board is empty.

Postcondition: You have produced a continuous block of whites of length n.

Iterations: This will be an iterative algorithm. You will have "iterations" for $i = 0, 1, \ldots, n$. Each such iteration will contain as many rounds of turns as you need to make progress and maintain the loop invariant.

Loop Invariant: After i "iterations", you have constructed many *special* blocks.

Length: Each such special block consists of i whites in a row.

Isolation: Each is far away from any black and from each other.

Number: The number of such special blocks in 10^{n-i}.

In addition to these special blocks the board will contain many white blocks that are abandoned because they have a black on one end.

- Complete Jeff's steps in completing the description/proof of this algorithm.
- Also compute the total number N of whites that you place.

13. **Multiplying:** The ancient Egyptians and the Ethiopians had advanced mathematics. Merely by halving and doubling, they could multiply any two numbers correctly. Say they want to buy 15 sheep at 13 Ethiopian dollars each. Here is how he figures out the product. Put 13 in a left column, 15 on the right. Halve the left value; you get $6\frac{1}{2}$. Ignore the $\frac{1}{2}$. Double the right value. Repeat this (keeping all intermediate values) until the left value is 1. What you have is

13	15
6	30
3	60
1	120

Even numbers in the left column are evil and, according the story, must be destroyed, along with their guilty partners. So scratch out the 6 and its partner 30. Now add the right column giving $15 + 60 + 120 = 195$, which is the correct answer.

Give the key steps to describe this LI algorithm.

Hint: Our code does not need remember all previous values. Instead, let x and y hold the original input integers, let ℓ hold the current left value, r the current right, and s the sum of all previous right values that will be included in the final answer.

Hint: Break steps within the loop (and proof that the loop invariant is maintained) into two steps. In the first step, if ℓ is odd it decreases by one. In the second step, ℓ (now even) is divided by two. These steps, must update r and s as needed. The running values will be as follows.

ℓ	r	s
13	15	0
12	15	15
6	30	15
3	60	15
2	60	75
1	120	75
0	120	195

(a) **Specification:**
$\langle pre{-}cond \rangle$ is that the input instance consists of two positive integers x and y.
$\langle post{-}cond \rangle$ is that the output is their product.

(b) *** Define Loop Invariant:** Give a meaningful loop invariant relating the current values of ℓ, r, s, x, and y. (Hint, look at the Shrinking Instance loop invariant.) (5 marks)

(c) *** Establishing the Loop Invariant:** (5 marks)

(d) **Picture:** If it is helpful for you, draw pictures to give a geometric explanation for the steps. (0 marks)

(e) **Step:** If it is helpful for you, define the Step in the loop (or even all the code). (0 marks)

(f) Maintaining the Loop Invariant for the first half of the step. (5 marks)

(g) Maintaining the Loop Invariant for the second half of the step. (5 marks)

(h) *** Ending:** Obtain the Postcondition.
Hint: What is the Ethiopian exit condition? How might you improve on this? (5 marks)

(i) **Time Complexity:** Give the time complexity of our multiplication algorithm. (5 marks)

14. (Answer in slides)
Multiplying using Adding: My son and I wrote a JAVA compiler for his grade 10 project. We needed the following. Suppose you want to multiply $X \times Y$ two n bit positive integers X and Y. The challenge is that the only operations you have are adding, subtracting, and tests like $\leq$. The standard high school algorithm seems to require looking at the bits of X and Y and hence can't obviously be implemented here. The Ethiopian algorithm requires dividing by two, so can't be implemented either. A few years after developing this algorithm, I noticed that it is in fact identical to this high school multiplication algorithm, but I don't want to tell you this because I don't want you thinking about the algorithm as a whole. This will only frighten you. All you need to do is to take it one step at a time. I want only want you to establish and maintain the loop invariant and use it to get the post condition. (The flavor is very similar to what was done in the Ethiopian problem on the practice test.) To help, I will give you the loop invariants, the measure of progress, and the exit condition. I also want you to explain what the time complexity of this algorithm is, i.e. the number of iterations and the total number of bit operations as a function of the size of the input.

We have values i, x, a, $u[]$, and $v[]$ such that

Useful Arrays: Before the main loop, I will set up two arrays for you with the following values. Then they will not be changed.

LI0': $u[j] = 2^j$, (for all j until $u[j] > X$)

LI0": $v[j] = u[j] \times Y$. Note $v[j] - u[j] \times Y = 0$

My Code: For completion, I include my code, but it is not necessary for you to understand it.

$u[0] = 1$
$v[0] = Y$
$j = 0$
while($u[j] \leq X$)
    $u[j+1] = u[j] + u[j]$
    $v[j+1] = v[j] + v[j]$
    $j = j + 1$
end while

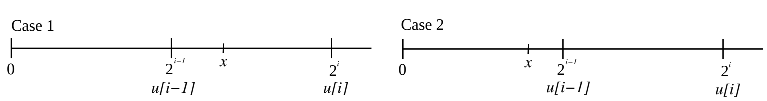
For Main Loop: These are the loop invariants for you to deal with.

LI0: Neither X nor Y change.

LI1: $X \times Y = x \times Y + a$ (i.e. Shrinking Instance)

LI2: $x \geq 0$

LI3: $x < 2^i = u[i]$ (below are two cases)



Measure of progress: i. It decreases by 1 each iteration.

Exit Condition: $i = 0$

Use the steps laid out in class to complete the description of the algorithm.

15. Computing square root digit by digit: The following is the algorithm for taking square roots that I learned in high school. The loop invariant is both the classic more of the input and the classic more of the output. As such, you should be able to quickly complete the steps. Do not panic. Your job is simply to trust and follow the steps. It should all fall out simply.

I wrote a long and fascinating explanation of how this algorithm looks like long division. But on reflection it was not necessary to do the test and only distracted. It will be included in the solution set.

As I have often said — do not try to understand this algorithm.

algorithm $SquareRoot(p)$

$\langle$***pre−cond***$\rangle$**:** P is an integer
but we pair its digits as $P = \langle a_1, \ldots, a_t, \ldots, a_n \rangle = \langle 07, 80, 14, 71, 23 \rangle = 0780147123$.

$\langle$***post−cond***$\rangle$**:** Q is the square root of P rounded down to the nearest integer.
We consider its digits as $Q = \langle b_1, \ldots, b_t, \ldots, b_n \rangle = \langle 2, 7, 9, 3, 1 \rangle = 27931$.
$R = P - Q^2$ is the remainder.

```
begin
    ??
    loop
        ⟨loop−invariant⟩: From more of the input we have p = ⟨a_1, ..., a_t⟩.
            From more of the output we have q = ⟨b_1, ..., b_t⟩.
            As integers, q is the largest integer for which r = p − q^2 and r ≥ 0
        exit when ??
        a = a_{t+1} = next pair of digits in P
        p = 100p + a
        R = 100r + a
        loop b = 9, 8, ..., 0
            s = (10(2q) + b) × b
            r = R − s
            exit when r ≥ 0
        end loop
        q = 10q + b
    end loop
    return(⟨q, r⟩)
end algorithm
```

Hint: Ignore the loop over b. Just assume that it is setting b to be the biggest value in $\{0, .., 9\}$ for which r is non-negative.

(a) ($8 + 8 + 24 = 40$ marks) Complete the three key loop invariant steps for this algorithm.

(b) (8 marks) Determine the running time of algorithms.