CS3230 - Design and Analysis of Algorithm: Practice Set #1 - Correctness and Invariants

Eldon Chung
eldon.chung@u.nus.edu

National University of Singapore — March 4, 2019

Introduction

This is a practice and **completely optional** practice problem set that revolves around proving correctness of algorithms via invariants or otherwise. Solutions will not be provided! Post on the LumiNUS forums if you would like to attempt the questions, and help will be provided in the form of either verification or guidance.

A few comments about convention: The algorithms provided will be in pseudocode. Arrays here will be 1-indexed. For *this* problem set you may also assume that array access, comparison, and arithmetic is done in $\Theta(1)$ time. Arrays will be passed by reference in function calls and returns. Allocating space for or initialising n-sized arrays (to whatever value) will take $\Theta(n)$ time. Also there are no space limitations unless otherwise stated.

In terms of notation, an n-sized array A will be denoted as $A[1..n]$. Accessing the i^{th} element in an array is denoted by $A[i]$. Variable assignments are denoted by $\leftarrow$, e.g. $x \leftarrow 5$ would mean variable x is being assigned a value of 5. As is standard, ∞ will be used to denote infinity, and despite not being an actual realisable concept in most machines and programming languages, it is ostensibly `INT_MAX` in C++ or `Integer.MAX_VALUE` in Java. Meanwhile $\emptyset$ will be used to denote empty data structures (not to be confused with null pointers). E.g. $S = \emptyset$ can be a stack that contains no elements, yet it is possible to push elements into S, and invoking a pop operation on S when it is empty would just return nothing instead of crashing as most languages do.

Questions

Exercise 1 Finding the Minimum

Provided with the algorithm below that finds the minimum element in an array A of size n:

```
procedure FIND-MIN($A[1..n]$)
    $minElement \leftarrow \infty$
    for $i = 1$ to $n$ do
        if $minElement > A[i]$ then
            $minElement \leftarrow A[i]$
        end if
    end for
    return $minElement$
end procedure
```

1. Find a suitable invariant. Do this by relating the number of iterations run (call it i), the value of $minElement$ and portions of the subarray, specifically $A[1..i]$ or $A[1..i-1]$.

2. Using the invariant that you have found previously, argue that the correctness of the algorithm:

    a) *(Initialisation:)* Show that the invariant holds before the first iteration.

    b) *(Maintenance:)* Assuming that the invariant held for the i^{th} iteration, it will also hold for the $(i+1)^{th}$ iteration.

    c) *(Termination:)* Show that when the algorithm terminates, we are able to conclude that the algorithm was correct.

Exercise 2 I hope you guys know what a median is.

Consider the problem of finding the online median, that is to say the median of $A[1]$, $A[1..2]$, $A[1..3]$, until , $A[1..n]$. E.g. given $A = [2, 6, 5, 3, 8, 4]$, the medians will be:

- $A[1] = [2]$, so the median is 2.
- $A[1..2] = [2, 6]$, so the median is the average of 2 and 6.
- $A[1..3] = [2, 6, 5]$, has a median of 5.
- $A[1..4] = [2, 6, 5, 3]$, has a median which is the average 3 and 5.
- $A[1..5] = [2, 6, 5, 3, 8]$, has a median 5.
- $A[1..6] = [2, 6, 5, 3, 8, 4]$, has a median which is the average 4 and 5.

Now you're provided the heap solution below:

Algorithm A heap solution to the running median problem

```
procedure SOLUTION(Array of Integers: $A[1..n]$) ▷ Heaps will have insert, size, peek and extract-min (or extract-max) operations
    Initialise min-heap $minH = \emptyset$
    Initialise max-heap $maxH = \emptyset$
    for $A[i] = A[0]$ to $A[n]$ do
        $a \leftarrow A[i]$
        if $minH.size = maxH.size$ then
            if $a < maxH.peek()$ then
                Insert $a$ into $maxH$
            else if $a > minH.peek()$ then
                Insert $a$ into $minH$
                $min \leftarrow minH.extract\text{-}min()$
                Insert $min$ into $maxH$
            else
                Insert $a$ into $maxH$
            end if
            print $maxH.peek()$
        else if $minH.size + 1 = maxH.size$ then
            if $a < maxH.peek()$ then
                Insert $a$ into $maxH$
                $max \leftarrow maxH.extract\text{-}max()$
                Insert $max$ into $minH$
            else if $a > minH.peek()$ then
                Insert $a$ into $minH$
            else
                Insert $a$ into $maxH$
                $max \leftarrow maxH.extract\text{-}max()$
                Insert $max$ into $minH$
            end if
            print average of $minH.peek()$ and $maxH.peek()$
        end if
    end for
end procedure
```

1. Notice that the **if** clauses do not consider any other cases besides when $minH.size = maxH.size$ and $minH.size + 1 = maxH.size$. Though this might not seem exhaustive. Argue that at every iteration, only either of these two cases will hold.

2. Using the previous question, come up with a suitable invariant, then use that to prove that at every iteration, the algorithm is correct, that is that it returns the median correctly.

3. As an aside, what is an upperbound on the running time of the algorithm given n elements?

* Exercise 3 A Familiar Algorithm...

You are given a graph (with positive weights) G as an adjacency list, such that for every vertex v the list of its neighbours as well as the respective edge weights (denoted by $weight(u, v)$ for the weight from u to v) are provided. Consider the single-source shortest path algorithm Dijkstra, but with the modification that the priority queue is replaced with a normal FIFO queue instead, as specified below:

Algorithm A modified version of Dijkstra, which only returns the distances from a starting vertex to every other vertex

```
procedure WEIRD-DIJKSTRA(Adjacency List: $G$, Starting Vertex: $v$)
    Initialise FIFO queue $Q = \emptyset$
    Initialise an array $Dist[1..n]$ with $\infty$
    Enqueue $v$ into $Q$
    $Dist[v] \leftarrow 0$
    while $Q$ is not empty do
        Dequeue the next element $Q$ as $c$    ▷ Take the next element as the queue as the current node
        for all Neighbours $r$ of $c$ do
            if $Dist[r] > Dist[c] + weight(c, r)$ then
                $Dist[r] \leftarrow Dist[c] + weight(c, r)$
                Enqueue $r$ into $Q$
            end if
        end for
    end while
    return $Dist[1..n]$
end procedure
```

We will do this in parts, we will firstly need to show that it eventually terminates, and that it is correct. Note that runtime is not something we are concerned with here. For the following questions, let $MinDist[1..n]$ denote the list of actual shortest distances froSm a given source to all vertices in the graph. We will refer to the lines 6-14 as the outer loop and the lines 8-13 as the inner loop.

1. Find a suitable loop invariant that allows for you to show the correctness of the algorithm.

   a) Argue that as long as there is at least one vertex u such that $Dist[u] > MinDist[u]$, then there is at least one other vertex v in Q such that $Dist[v] = MinDist[v]$.

   b) Argue that if there exists some vertex u such that $Dist[u] > MinDist[u]$, then eventually at some point in time the Q will hold a vertex v such that $minDist[v]+weight(v, u) = minDist[u]$. (Hint: do this inductively or by contradiction.)

2. Constructing an invariant from the before arguments, prove the algorithm is correct.

   a) *(Initialisation:)* Show that the invariant holds before the first iteration.

   b) *(Maintenance:)* Assuming that the invariant held for the i^{th} iteration, it will also hold for the $(i + 1)^{th}$ iteration.

   c) *(Termination:)* Show that when the algorithm terminates, we are able to conclude that the algorithm was correct.

3. *(Optional)* Say the queue was placed with a max-heap instead. Does the algorithm still work?

* Exercise 4 A tale of two pointers.

The pair sum problem is defined as follows: Given an array $A[1..n]$ and an integer k, you are asked to find two distinct indices i, j such that $A[i] + A[j] = k$, and NULL if no such pairing exists. Thankfully, having taken CS2040 last semester you recall that this question was covered and whip out your slides to find one of the algorithms, which is presented below:

Algorithm An algorithm that solves the pair sum problem by using two indices given a sorted array.

```
procedure PAIR-SUM-SOLUTION((Sorted) Array of Integers $A[1..n]$, Target Value: $k$)
    $right \leftarrow n$
    $left \leftarrow 1$
    while $left < right$ do
        if $A[left] + A[right] == k$ then
            return $(i, j)$
        else if $A[left] + A[right] < k$ then
            $left \leftarrow left + 1$
        else if $A[left] + A[right] > k$ then
            $right \leftarrow right - 1$
        end if
    end while
    return NULL
end procedure
```

Note: This question will now be slightly less guided, so that there's more room for you to think and figure it out yourself.

* 1. Find the loop invariant. Hint: relate $left$ and $right$ to $A[1..left - 1]$ and $A[right + 1..n]$.

2. As always, prove the following:

   a) *(Initialisation:)* Show that the invariant holds before the first iteration.

   * b) *(Maintenance:)* Assuming that the invariant held for the i^{th} iteration, it will also hold for the $(i + 1)^{th}$ iteration.

   c) *(Termination:)* Show that when the algorithm terminates, we are able to conclude that the algorithm was correct, assuming the invariant held. Then, from the three statements we are able to say that since the algorithm could start with the invariant and consistently maintain it throughout its execution until the end, it was correct.

3. Your good friend Aiken decides to code out the same algorithm, but with the **else if** statements as **if** statements instead. Does the algorithm still work?

4. *(Optional)* Aiken's best friend Dueet comes along and notices you taking a lot of effort to do this and mentions you could just do a proof by contradiction instead to show it was correct, and it'd be much easier. Come up with a proof by contradiction. (Of course from this alone we can't say one proof was easier than the other. This question is meant to just let you consider other angles for proving correctness.)

** Exercise 5 The Revenge of the Red and Blue Balls (*CS1231 AY18/19 Assignment 1 Q5.*)

In CS1231, an question was once given out to the students about blue balls and red balls. The students were expected to prove (by induction) that for any given arrangement of n blue and n red balls arranged in a circle, there exists a walk along this circle such that the number of blue balls that one passes would always be greater than or equals to the number of red balls that one would also pass, from the starting point, around the circle and back. Refer to below for the lemma:

Theorem 1. *$n > 0$ red balls and n blue balls are arranged to form a circle. A walk starts at some ball and proceeds clockwise until the last ball (the ball just before the starting ball). A valid walk around the circle is a walk such that at all times during the walk, the number of red balls that you have passed is greater than or equals to the number of blue balls that you have passed.*

For any circle formed by n blue balls and n red balls, if you can choose where to start, there is always a valid walk.

Dueet, unconvinced by this and in hopes of disproving it, decides to implement an algorithm to actually test out a given circular arrangements of n blue and n red balls. His algorithm aims to find the starting point to the arrangement so that the walk is valid. As the values of n got larger, the brute force $\mathcal{O}(n^2)$ solution was considerably slow and took time.

To help, Aiken cooks up some new linear time algorithm. Clearly Aiken and Dueet are not interested in doing any proofs right now, especially by induction. However, she is unsure of her implementation. Aiken turns to you for help. The algorithm is provided below, and you may assume no significant or meaningful deviation between her program and the algorithm.

You may assume that the input is a circular array, such that indexing is done modulo the size of the array. i.e. Given $A[1..n]$, $A[1..n]$, $A[n] = A[n+1]$, $A[-1] = A[n]$ and so on. Additionally, for ease of proof, you may assume that $A[1]$ is always blue. This is because the array is circular anyway, and you can start the algorithm at any point that has a blue ball. This takes at most $\mathcal{O}(n)$ time since you just need to remember the first index that there is a blue ball and apply an offset.

Algorithm Aiken's proposed algorithm. Takes in a circular arrangement of blue and red balls and returns the starting point.

```
 1: procedure VERIFIER-1(Circular Array of Balls $A[1..n]$)
 2:     $blueCount \leftarrow 1, redCount \leftarrow 0$
 3:     $left \leftarrow A[1], right \leftarrow A[1]$
 4:     $hasGoneAroundCircle \leftarrow false$
 5:     while $right + 1 \neq left \vee !hasGoneAroundCircle$ do
 6:         $hasGoneAroundCirle \leftarrow true$
 7:         if $blueCount > redCount$ then
 8:             while $blueCount > redCount$ do
 9:                 $right \leftarrow right + 1$
10:                 if $A[right]$ is blue then
11:                     $blueCount \leftarrow blueCount + 1$
12:                 else
13:                     $redCount \leftarrow redCount + 1$
14:                 end if
15:             end while
16:         else if $blueCount <= redCount$ then
17:             while $blueCount <= redCount$ and $left - 1 \neq right$ do
18:                 $left \leftarrow left + 1$
19:                 if $A[left]$ is blue then
20:                     $blueCount \leftarrow blueCount + 1$
21:                 else
22:                     $redCount \leftarrow redCount + 1$
23:                 end if
24:             end while
25:         end if
26:     end while
27:     return $left$
28: end procedure
```

For the following questions, we will refer to the lines 8-15 as the first inner loop, lines 17-24 as the second inner loop, and lines 5-26 as the outer loop. Prove or disprove that Aiken's algorithm is correct. For one to prove that it is correct, one would need to either use invariants or some other form of proof. To show that it is incorrect, one simply need only come up with a counter example where the algorithm fails.

1. What happens if Aiken had replaced the **else if** on line 16 with **if** instead? Ignoring possible syntactical errors, would the algorithm work?

2. The **if** clauses seem quite redundant seeing how as the **while** loops have the same condition. Without the **if** clauses, would the algorithm work?

3. What about the original algorithm? Would it work? If it does, answer the four questions below and prove its correctness. Else, come up with a counter example.

    a) Find the invariant for the first inner loop and prove that it holds for the three cases.

    b) Find the invariant for the second inner loop and prove that it holds for the three cases.

    c) Find the invariant for the outer loop and prove that it holds for the thee cases.

    d) Conclude that the algorithm is correct.

Matthew, one of Aiken's frequent lunch buddies takes part in the discussion over lunch (oh surprise surprise) one weekend. As a competitive programming enthusiast, Mr. Matthew claims that a some inspiration from Kadane's algorithm should also do the trick, and thus proposes a simplified version of that algorithm. However, Matthew too, does not like to do proofs and prefers to verify his algorithm correctness by test cases (proof by **AC**), which is not an accepted proof technique in CS1231 nor CS3230... It'll be up to you to help Mr. Matthew substantiate his claims or prove him wrong.

Algorithm Matthew's proposed algorithm

```
procedure VERIFIER-2(Circular Array of Balls $A[1..n]$)
    $difference \leftarrow 0$
    $lastValidPosition \leftarrow 1$
    for $pos = 1$ to $n$ do
        if $A[pos]$ is blue then
            $difference \leftarrow difference + 1$
        else if $A[pos]$ is red then
            $difference \leftarrow difference - 1$
        end if
        if $difference < 0$ then
            $lastValidPosition \leftarrow pos + 1$
            $difference \leftarrow 0$
        end if
    end for
    return $lastValidPosition$
end procedure
```

1. Prove or disprove the algorithm is correct. (Hint provided below in case you need it.)

   a) To disprove the algorithm, provide a counter example as to why it doesn't work. In other words, find a test case that would give Matthew a **WA** (wrong answer).

   b) To prove the algorithm is correct, I trust you know what to do by now.

Additionally, since you have taken CS1231 (assuming you did so in AY17/18 Sem 1 or AY18/19 Sem 1) and have actually proven the theorem yourself, you may use it as aid in your proof that the algorithm is correct (You may also use it even if you have not proved it). Though note that just because the theorem is correct it does not immediately imply that the algorithm is correct! i.e. Existence of a starting point does not imply the algorithm returns it.