Assignment 2

Course: COMPSCI 2C03

Date: 10/03/2025

Author: Sulagna Nandi

MacID: nandis

Student Number: 400575118

Professor: Dr. Vincent Maccio

Problem 1.1: Push and Pop

After PUSH(S, 7):

7

After PUSH(S, 9):

7, 9

After POP(S):

7

9 returned.

After PUSH(S, 12):

7, 12

After PUSH(S, 13):

7, 12, 13

After POP(S):

7, 12

13 returned.

After POP(S):

7

12 returned.

After POP(S):

(empty) 7 returned.

Problem 1.2: Enqueue and Dequeue

After ENQUEUE(Q, 7):

7

After ENQUEUE(Q, 9):

9, 7

After DEQUEUE(Q):

9

7 returned

After ENQUEUE(Q, 12):

12, 9

After ENQUEUE(Q, 13):

13, 12, 9

After DEQUEUE(Q):

13, 12

9 returned.

After DEQUEUE(Q):

13

12 returned.

After DEQUEUE(Q):

(empty)

13 returned.

Problem 1.3: Two Stacks, One Array

We will essentially have two stacks growing inwards in this array from the ends. To make sure both can fit, the sum of their lengths needs to be less than the array's length. Also, each time we push or pop each stack, we must update where the top is.

Algorithm 1 TwoStacksInOneArray

```
1: Array A[0..n − 1], integers s1_index, s2_index
2: s1_index ← −1
3: s2_index ← n

4: Push1(x):
5: if s1_index + 1 == s2_index then
6:     error: overflow
7: end if
8: s1_index ← s1_index + 1
9: A[s1_index] ← x

10: Push2(x):
11: if s1_index + 1 == s2_index then
12:     error: overflow
13: end if
14: s2_index ← s2_index − 1
15: A[s2_index] ← x

16: Pop1():
17: if s1_index == −1 then
18:     error: underflow
19: end if
20: x ← A[s1_index]
21: s1_index ← s1_index − 1
22: return x

23: Pop2():
24: if s2_index == n then
25:     error: underflow
26: end if
27: x ← A[s2_index]
28: s2_index ← s2_index + 1
29: return x
```

Problem 2.1: Proving Correctness

Invariant

The ComputeSquareRoot algorithm begins with $i = 0$, increments by 1 per iteration, and will go up to a maximum of $i = \lceil \frac{n}{2} \rceil$. If the square root of n is reached before $i = \lceil \frac{n}{2} \rceil$, then that i will be returned (and the loop will of course terminate). Therefore, we can define our invariant to be the following:

$$\boxed{I(i) := 0 \leqslant i \leqslant \left\lceil \frac{n}{2} \right\rceil,\ ((i \text{ returned}) \vee (i^2 \neq n))}$$

Invariant Base Case:

This is when $i = 0$. There are two subcases to consider.

Base Subcase 1: $n = 0$

In the first iteration, $(i^2 \neq n) \equiv$ false since $i^2 = 0$ and $n = 0$. Since our if statement condition is true, the command inside it to return i will run (and the loop will terminate). Thus, i will have been returned, so (i returned) $\equiv$ true. This makes our invariant true, since $(i^2 \neq n) \vee (i$ returned$) \equiv$ false $\vee$ true $\equiv$ true.

Base Subcase 2: $n \neq 0$

In the first iteration, $(i^2 \neq n) \equiv$ true since $i^2 = 0$ and $n \neq 0$. Since our if statement condition is false, the command inside it to return i will not run (and the loop will continue). Thus, i will not have been returned, so (i returned) $\equiv$ false. This makes our invariant true, since $(i^2 \neq n) \vee$

(i returned) $\equiv$ true $\vee$ false $\equiv$ true.

Inductive Step: $I(i_k) \implies I(i_{k+1})$

Assume $I(i_k)$ and let $i_{k+1} \coloneqq i_k + 1$

Case 1: $(i_k)^2 \neq n$

According to our pseudocode, the if statement condition was not met for the k^{th} iteration, so ($i_k$returned) $\equiv$ false. Since no value was returned, our loop must continue to the $(k+1)^{th}$ iteration.

Subcase 1a: $(i_{k+1})^2 = n$

According to our pseudocode, we will enter the if statement and return i_{k+1}. Thus $I(k+1)$ will be true.

Subcase 1b: $(i_{k+1})^2 \neq n$

This makes $I(k+1)$ true.

Case 2: $(i_k)^2 = n$

By our induction hypothesis, i_k must have been returned in the k^{th} iteration. In the $(k+1)^{th}$ iteration (which does not actually happen since the loop will terminate after a value has been returned), $(i_{k+1})^2 \neq n^2$ since, by Case 2, we already have $(i_k)^2 = n$. Thus, $I(i_{k+1}) \equiv$ true.

Bound Function

The for loop in ComputeSquareRoot will run from $i = 0$ to a maximum of $i = \lceil \frac{n}{2} \rceil$. It will, however, stop when $i^2 = n$. We are only looking at possible square roots in the natural numbers (since i is never negative in this algorithm).

We can define a function for the number of iterations remaining as:

$$\boxed{f(i) := \sqrt{n} - i.}$$

This is valid since $\sqrt{n} \leqslant \lceil \frac{n}{2} \rceil$ for any perfect square (number with an integer square root) n.

Proof:
$\sqrt{n} \leqslant \frac{n}{2} \leqslant \lceil \frac{n}{2} \rceil$
$\sqrt{n} \leqslant \frac{n}{2}$
$n \leqslant \frac{n^2}{4}$
$4n \leqslant n^2$
This is valid for all $n \geqslant 4$. In addition, $\sqrt{n} \leqslant \lceil \frac{n}{2} \rceil$ is clearly true for $n = 0$ and $n = 1$.

Since $\sqrt{n}, i \in \mathbb{N}$ and $0 \leqslant i \leqslant \sqrt{n}$, $f(i) \in \mathbb{N}$, and i is strictly increasing, $f(i)$ must be strictly decreasing. Thus, we know $f(i)$ will eventually reach 0,

which means the loop will terminate.

Since our base case, all cases for $I(i_k) \implies I(i_{k+1})$ have been shown, and our bound function shows that the loop will end, this algorithm is correct.

Problem 2.2: Worst-Case Time Complexity

This algorithm will run exactly $\sqrt{n}+1$ times, since the iteration where $i = 0$ is included and the loop will terminate when $i^2 = n$. Therefore, the worst-case time complexity is $\Theta(\sqrt{n})$.

Problem 2.3: BetterComputeSquareRoot(n)

Original Algorithm

Algorithm 2 ComputeSquareRoot(n)

Pre: n is a perfect square.

```
1: for i := 0 to ⌈n/2⌉ (inclusive) do
2:     if i^2 = n then
3:         return i
4:     end if
5: end for
```

Post: return (positive) $\sqrt{n}$

Improved Algorithm

Algorithm 3 BetterComputeSquareRoot(n)

Pre: n is a perfect square.

```
1: l := 0
2: h := ⌈n/2⌉
3: while l ⩽ h do
4:     m := ⌊(l+h)/2⌋
5:     if m^2 < n then
6:         l := m + 1
7:     else if m^2 > n then
8:         h := m − 1
9:     else
10:        return m
11:    end if
12: end while
```

Post: return (positive) $\sqrt{n}$

This algorithm takes $\Theta(\log(n))$ time because we are cutting the range in which we are looking for the square root of n in half each iteration. This

cutting can happen a maximum of $\log(n)$ times. It is correct because we are simply implementing a binary search algorithm between 0 and $\left\lfloor \frac{n}{2} \right\rfloor$ instead of linear search, which means we are guaranteed to find the $\sqrt{n}$ given that n is a perfect square (has an integer square root).

Problem 2.4: Non-integer Square Root

Algorithm 4 PRECISECOMPUTESQUAREROOT(n)

Pre: n is a positive value, ε is a value such that $\varepsilon > 1$.

```
1: l := 0
2: h := n/2
3: while l ⩽ h do
4:     m := (l+h)/2
5:     if m^2 < (n − ε) then
6:         l := m
7:     else if m^2 > (n + ε) then
8:         h := m
9:     else
10:        return m
11:    end if
12: end while
```

Post: return (positive) $\sqrt{n}$

This algorithm is correct because it will keep closing in on $\sqrt{n}$ until the first time it finds a real number within ε distance from $\sqrt{n}$. It will not overshoot since the search window only gets smaller each time, and we only eliminate values that are too large or too small (and not $\sqrt{n}$). The time complexity should be $O(\log(\frac{n}{\varepsilon})$ since ε, if it is very small, will significantly contribute to how many times we have to cut our search window in half (note that we're still using a similar binary search approach but just with more precision, which is why $\log(n)$ is involved in the time complexity).