

**Q1**

- (a) 5
- (b) 2
- (c) 3

**Q2**

- (a)(i)

	i	anInt
Before for loop	-	0
	2	1
	3	2
Exit for loop	4	2

- (ii)  $4*2 = 8$  is returned

- (b) pre: arr must be a full array with size at least 4.

**Q3**

```
function QU3(n)
{
  var a in Array (of SIZE n) of Int
  var i in Int
  for (i <-- 1 to n)
  {
    a <-- PUT(i,a,0)
  }
  return a
}
```

**Q4**

method aMeth(n in Int) state change

pre: true

post: If the spot is in state (X,Y), then its state is changed to (X+n,Y).

**Q5**

```
var t in Turtle
t.store()
t.forward(10)
t.leftTurn()
t.forward(5)
t.restore()
```

**Q6**

```
v.first() == v.last()
```

**Q7**

- (a)(i) 5 (ii) {'Q', 'Z'}

(b)  $C = \{(2,1), (3,2), (4,3)\}$

**Q8**

(a) (22 (33 () ()) (44 (44 () ()) ()))

(b) true

(c) invalid; t.rightTree().rightTree() is the empty tree, which does not have a root.

**Q9**

The columns were filled in the order shown at the bottom.  
Column 3 is the final answer.

a	b	$\neg$	$( a \wedge \neg b )$
T	T	T	F
T	F	F	T
F	T	T	F
F	F	T	F
		3	2 1

**Q10**

$\neg a \Rightarrow b \equiv \neg \neg a \vee b$  Rewrite  $\Rightarrow$   
 $\equiv a \vee b$  Negation (iii)

**Q11**

(a) {attA} is a key. It is determining and minimal.  
 Determining since FD1 says that attA determines attB, then FD2 also allows us to add AttC (transitivity), then FD3 allows us to add attD (transitivity again).  
 Minimal since there is no nonempty proper subset of {attA} to consider.

(b) {attB} is not a key; we cannot determine unique values of attA or attD.

(c) {attA, attB} is not a key because it is not minimal: {attA} is sufficient from part (a).

**Q12**

$$\begin{aligned}
 \text{QU12}([2,5,3]) &= 2 + \text{QU12}([5,3]) && //(R) \\
 &= 2 + (5 + \text{QU12}([3])) && //(R) \\
 &= 2 + (5 + (3 + \text{QU12}([\ ]))) && //(R) \\
 &= 2 + (5 + (3 + 0)) && //(S) \\
 &= 10 && \text{arithmetic}
 \end{aligned}$$

**Q13**

Before the for-loop we have a spot declared, so it will be in default state (0,0). It then moves left by one to (-1,0); it moves left again by one to (-2,0); it moves up by one to (-2,1). Now, with n = 0, the for-loop is not executed, so the spot is in the state (-2,1), which is consistent with  $p(k) \Rightarrow$  state is  $(k-2, 2k+1)$  with  $k = 0$ .

**Q14**

$$T(0) = 5 \quad (\text{S})$$

$$T(n) = T(n-1) + 7 \quad (\text{R})$$

**Q15**

(a)

```
function COUNTVEC(s, v)
{
  var n in Int
  var i in Int
  n <-- 0
  for (i <-- 1 to v.size())
  {
    if (v.at(i).equal(s)) then
    {
      n <-- n+1
    }
  }
  return n
}
```

(b)

```
function COUNTTREE(c, t)
{
  if (t.isEmpty()) then
  {
    return 0
  }
  else
  {
    if (t.getRoot() == c) then
    {
      return 1 + COUNTTREE(c, t.leftTree()) + COUNTTREE(c, t.RightTree())
    }
    else
    {
      return COUNTTREE(c, t.leftTree()) + COUNTTREE(c, t.RightTree())
    }
  }
}
```

(d) Let  $p(n)$  be the proposition that after the fragment has been executed,(i) the marker is in state  $(-n, 0, \text{"right(0)"})$ (ii) the diagram is  $\{(x, 0) : |x| \leq n\}$ 

First we establish a basis.  $p(0)$  holds since in this case the for-loop is not executed, and only the three statements before the for-loop are significant. A new turtle in default state  $(0, 0, \text{"right(0)"})$  is created, and moving `forward(0)` adds a single grid point at the origin.

Now, suppose  $p$  holds for some integer  $k-1$ ; we wish to show that  $p(k)$  follows from  $p(k-1)$ .

Clearly, we only need to consider the last iteration of the for-loop; at  $k-1$ , the turtle is in state

$(-(k-1), 0, \text{"right(0)"})$ , and the diagram has  $\{(x, 0) : |x| \leq k-1\}$  by hypothesis. Now we do iteration  $k$  which moves

the turtle forward  $2k-1$  places, adding  $(2k,0)$  to the diagram. The turtle then turns around and moves forward  $2k$  places adding  $(-2k,0)$  to the diagram. Finally, it turns around ending in state  $(-2k,0, \text{"right}(0))$ . And this is what  $p(k)$  asserts. So, from the basis and the demonstration that  $p(k)$  follows from  $p(k-1)$ , by mathematical induction we have shown that  $p(n)$  holds for all  $n \geq 0$ .

**Q16**

(a)  $s = [1,2]; t = [1,3,4]$

	sameFirst	Ret	restOfs	restOft	Done
Before while	true	[]	[1,2]	[1,3,4]	false
	true	[1]	[2]	[3,4]	false
Exit while	false	[1]	[2]	[3,4]	false

(b) (i) Neither  $ret \oplus restOfs$  nor  $ret \oplus restOft$  are changed by a single execution of the while loop body because if their first elements agree, then  $ret$  is augmented while  $restOfs$  and  $restOft$  are diminished by one element; however if their first elements do not match, or (at least) one of them is of zero size,  $ret$  is unaffected.

(ii) Just before the while loop  $restOfs = s, restOft = t, \text{ and } ret = []$ .

(c)  $MATCH(u \oplus v, u \oplus w) = u \oplus MATCH(v, w)$  clearly holds because the specification says that the match compares elements at the front of the sequences, and here they have a common sequence ( $u$ ) at the front, so the final result will depend on how much further  $v$  and  $w$  match at their fronts.

(d)  $ret \oplus MATCH(restOfs, restOft) = MATCH(s, t)$  is a loop invariant because  $ret$  holds the vector of Ints that have successfully matched so far and  $MATCH(restOfs, restOft)$  is what we have left to do. We have already established in (b)(ii) that before the loop  $restOfs = s, restOft = t, \text{ and } ret = []$ , so we have

$$ret \oplus MATCH(restOfs, restOft) = [] \oplus MATCH(s, t) = MATCH(s, t)$$

Also, (b)(i) showed that neither  $ret \oplus restOfs$ , nor  $ret \oplus restOft$  are changed by a single execution of the while loop so  $ret \oplus MATCH(restOfs, restOft)$  is a loop invariant.

(e) The loop exits when the while condition fails so  $A_F$  and  $B_F$  either do not match at their first element, or (at least one) has become zero size. From the specification, this match is the empty vector.

(f) Upon exit,  $ret = ret_F$  and  $MATCH(A_F, B_F) = []$  so we have

$$ret \oplus MATCH(A_F, B_F) = ret_F \oplus [] = ret_F \text{ so this implementation is proven correct.}$$

**Q17**

(a)(i) (A) {author} /~> {title}                      eg tuples t3 and t4  
 (B) {agent} /~> {classification}                  eg tuples t5 and t6

(ii) R1 gives FD1 {isbn} ~> {author, title, classification}  
 R2 gives FD2 {author, title} ~> {isbn}  
 R3 gives FD3 {author} ~> {authornat}  
 R4 gives FD4 {classification, authornat} ~> {agent}

(iii)  $K = \{author, title\}$  is a key because it is determining and minimal.

Determining: because FD2 immediately adds isbn to the list; then FD1 adds classification; then FD3 adds authornat, and FD4 adds agent. Thus K is sufficient to pick out a unique tuple.

Minimal: {author} alone cannot determine title (eg see t3 and t4). Also, {title} cannot determine {author} (eg see t1, t2).

Hence K is a key.

(iv) Another key could be {isbn}. It is determining: FD1 adds author, title, and classification to the list; FD3 adds authornat, and FD4 adds agent. Hence {isbn} is determining.

There are no proper subsets of {isbn} to consider, so it is also minimal too. Hence it is a key.

(v) We do not have second normal form because with key  $K = \{\text{author}, \text{title}\}$ , a proper subset {author} determines {authornat}, and {authornat}  $\notin K$ . To remedy this, we split it into 2 schemes:

S1: attributes: author, authornat

key {author}

FD3 {author}  $\sim\>$  {authornat}

S2: attributes: isbn, author, title, classification, agent

key {author, title}

FD1 {isbn}  $\sim\>$  {author, title, classification}

FD2 {author, title}  $\sim\>$  {isbn}

FD4A {classification, author}  $\sim\>$  {agent} Note that FD4 changed. (cf U11 Ex4.2 for another example of this)

S1 and S2 are now in 2<sup>nd</sup> normal form. {author} is a foreign key of S1 in S2.

S1 is already in 3<sup>rd</sup> normal form since we only have 2 attributes and thus cannot have a transitive dependency.

S2 is not in 3<sup>rd</sup> normal form, since we have

{isbn} $\sim\>$ {author, classification}	from FD1,
{author, classification} $\sim\>$ {agent}	from FD4A

However, {author, classification}  $\not\sim\>$  {isbn} (eg see t3 and t4)

Also agent  $\notin$  {author, classification, isbn} so we split S2 to remove the transitive dependency:

S3: attributes: author, classification, agent

key {classification, author}

FD4A {classification, author}  $\sim\>$  {agent}

S2A: attributes: isbn, author, title, classification

key {author, title}

FD1 {isbn}  $\sim\>$  {author, title, classification}

FD2 {author, title}  $\sim\>$  {isbn}

Now all schemes S1, S2A and S3 are in 3<sup>rd</sup> normal form.

(vii) We have {author} is a foreign key of S1 in S2A (there are many S1 tuples for one S2A tuple).

Also {author} is a foreign key of S1 in S3 (there are many S1 tuples for one S3 tuple).

Finally, {author, classification} is a foreign key of S3 in S2A (many S3 tuples for one S2A tuple).

(b) (i) setTree:  $\text{Int} \times \text{T} \times \text{T} \rightarrow \text{T}$

(ii) `newTree().getLeft() == newTree()` could be appropriate since `newTree()` creates an empty binary tree, and the left tree of that is defined (by specification) to be another empty tree.

(iii)  $\forall t \text{ in } T [ \forall l \text{ in } T [ \forall r \text{ in } T [ \forall x \text{ in } \text{Int} [ t.\text{setTree}(x,l,r).\text{getLeft}() == l ] ] ] ] ]$

**Q18**

(a) A somewhat wordy explanation is easier than a tabular one.

We enter `INSTRING('b', "abcdabc")` and execute lines, 1 to 7, and 9 to 13 (ie 12 lines in total). However 10 and 11 are recursive subcalls to `INSTRING('b', "abc")` and these two execute 1 to 9 and then 13 (ie 10 lines in total).

Thus we have  $12 + 2(10) = 32$  lines in total executed. This is not the worst case. Think of the string as being the result of pancaking a binary tree; the worst case is when each of the leaves of that tree are the element being sought since that leads to line 8 being executed as many times as there are leaves in the tree.

(b)  $T(1) = 10$  In this case, lines 1 to 9 and 13 are executed.

$$T(n) = 12 + 2T(\text{DIV}(n+1,2) - 1) \text{ for } n > 1$$

This must be so since if `c` is not the middle character then we execute lines 1 to 7, and 9 to 13 with lines 10 and 11 being recursive calls to problems of half the size.

A worst case `s` and `c` for `s` of length 7 could be `s = "abacaba"` and `c = 'a'`

(c)

n	T(n)
1	$T(1) = 10$
3	$T(3) = 12 + 2T(1) = 12 + 2(10) = 32$
7	$T(7) = 12 + 2T(3) = 12 + 2(32) = 76$
15	$T(15) = 12 + 2T(7) = 12 + 2(76) = 164$

By inspection, the order of  $T(n)$  is linear. As `n` goes up exponentially, so does  $T(n)$ . It's not constant, and not fast enough for quadratic. Very roughly, it looks like  $T(n) \approx 10n$ , so it's linear.

(d) `s = "abcdab"` is not valid input. It fails the precondition so any result, even if apparently plausible, would be invalid. In this case, we get a pair of recursive calls `INSTRING('b', "ab")` and `INSTRING('b', "da")`. When we attempt to access the middle element we have `mid <-- DIV(2+1,2)-1`, so `mid` is set to zero, and that causes the access violation on line 7. This does not indicate an invalid implementation.

(e)

$$U(0) = 5$$

$$U(n) = U(n-1)+3 \quad n \geq 1$$

We expand the recursion thus:

$$U(n) = U(n-1)+3$$

$$U(n-1) = U(n-2)+3$$

$$U(n-2) = U(n-3)+3$$

...

$$U(1) = U(0)+3$$

Summing and cancelling, we get  $U(n) = 3n + 5$ .

Let  $p(n)$  be the proposition that  $U(n) = F(n) = 3n+5$ .

The basis is established for  $p(0)$ , since  $U(0) = F(0) = 3(0)+5 = 5$ .

Now suppose  $U(k-1) = F(k-1)$  is true for some  $k > 1$ , then we are assuming  $p(k-1)$  is true and wish to show  $p(k)$  follows.

$$\begin{aligned} \text{Now, } U(k) &= U(k-1) + 3 && \text{(from Recurrence)} \\ &= (3(k-1)+5)+3 && \text{(by hypothesis)} \\ &= 3k - 3 + 5 + 3 \\ &= 3k + 5 \\ &= F(k) \end{aligned}$$

Hence by mathematical induction  $U(n) = 3n+5$  for  $n \geq 0$ .