1.
(a)
Command used - cbmc-5.1/dist/bin/cbmc --bounds-check --function queue_test prob1.c --unwind 5 --no-unwinding-assertions

Since the size of the buffer is set to 4, so the smaller unwinding bound that causes a buffer overflow is (size + 1) = 5.

Counterexample:

```
State 29 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

State 30 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

State 31 file prob1.c line 20 function dequeue thread 0
----------------------------------------------------
  lo=1 (00000000000000000000000000000001)

State 44 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

State 47 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

State 48 file prob1.c line 20 function dequeue thread 0
----------------------------------------------------
  lo=2 (00000000000000000000000000000010)

State 61 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

State 64 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

State 65 file prob1.c line 20 function dequeue thread 0
----------------------------------------------------
  lo=3 (00000000000000000000000000000011)

State 78 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)
```

```
State 81 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

State 82 file prob1.c line 20 function dequeue thread 0
----------------------------------------------------
  lo=4 (00000000000000000000000000000100)

State 95 file prob1.c line 19 function dequeue thread 0
----------------------------------------------------
  res=0 (00000000000000000000000000000000)

Violated property:
  file prob1.c line 19 function dequeue
  array `buf' upper bound in buf[(signed long int)lo]
  (signed long int)lo < 4l

VERIFICATION FAILED
```

(b)
Changed the code to fix the bounded buffer overflow problem (hw4_1(b).c).
Verified it works for unrolling up to 40 times.
Yes, it can be used to verify that the program has no buffer overflows.
This is because we have found a loop invariant which will keep the index count of accessed element within the array between 0 to size-1, irrespective of the number of times the loop executes (due to the percentile operation).

(c)
Upon checking CBMC on bin search function with unwinding of 5 or less, it always returns successful verification and there is no bounded buffer issue (with no unwinding assertions). If there are unwinding assertions however, it fails verification. I think this is related to the fact that the loop hasn't been unrolled sufficiently so we need not worry about verification failures due to unwinding assertions. We need to disable them to make sense since at least 6 iterations of the of the loop must occur before we know that the loop has been come out of, so I will present the results here wrt CBMC with no unwinding assertions.
Yes, no array access happens out of bounds for 5 or less unrolling of the loop.
We learn that the function is safe. This is because it doesn't take more than log(16) times to search for an element in an array of size 16 at the worst case, and the last time of the loop following that (the 5th iteration) is when we will come out of the loop since the loop condition will not have been met. So, if within 5 unrollings of the loop we don't get a bounded buffer problem, we will not anytime after that and hence the function is safe.

(d)
Repeating part(c) with an unwinding bound of 10, we get no buffer overflow. We can conclude that the function is safe from buffer overflows. Also note that since the loop has been sufficiently unrolled enough times, we need not use the “no-unwinding-assertions” flag.
This is because with a size of 16 only loop unrolling of up to 6 matters. The rest of the loop unrolling beyond 6 don't matter, and in terms of safety of the program, we have already established in (c) why the function is safe.

2.

(a) Instrumented code is hw4_2_abc.c).
(b)
Upon executing the following command on CBMC (provided code is hw4_2_abc.c) -
`cbmc-5.1/dist/bin/cbmc --function lock1 prob2.c`
We get a Verification Failed.

```
Building error trace

Counterexample:

State 18 file prob2.c line 14 function lock1 thread 0
----------------------------------------------------
  in_irq=64 (00000000000000000000000001000000)

State 19 file prob2.c line 15 function lock1 thread 0
----------------------------------------------------
  buf={ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
({ 00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000 })

State 20 file prob2.c line 16 function lock1 thread 0
----------------------------------------------------
  i=0 (00000000000000000000000000000000)

State 24 file prob2.c line 5 function lock thread 0
----------------------------------------------------
  l=1 (00000000000000000000000000000001)

State 26 file prob2.c line 26 function lock1 thread 0
----------------------------------------------------
  i=0 (00000000000000000000000000000000)

State 34 file prob2.c line 27 function lock1 thread 0
----------------------------------------------------
  buf[0]=0 (00000000000000000000000000000000)
```

```
State 35 file prob2.c line 26 function lock1 thread 0
----------------------------------------------------
  i=1 (00000000000000000000000000000001)

State 44 file prob2.c line 27 function lock1 thread 0
----------------------------------------------------
  buf[1]=0 (00000000000000000000000000000000)

State 45 file prob2.c line 26 function lock1 thread 0
----------------------------------------------------
  i=2 (00000000000000000000000000000010)

State 54 file prob2.c line 27 function lock1 thread 0
----------------------------------------------------
  buf[2]=0 (00000000000000000000000000000000)

State 55 file prob2.c line 26 function lock1 thread 0
----------------------------------------------------
  i=3 (00000000000000000000000000000011)

State 64 file prob2.c line 27 function lock1 thread 0
----------------------------------------------------
  buf[3]=0 (00000000000000000000000000000000)

State 65 file prob2.c line 26 function lock1 thread 0
----------------------------------------------------
  i=4 (00000000000000000000000000000100)

State 74 file prob2.c line 27 function lock1 thread 0
----------------------------------------------------
  buf[4]=0 (00000000000000000000000000000000)

State 75 file prob2.c line 26 function lock1 thread 0
----------------------------------------------------
  i=5 (00000000000000000000000000000101)

State 81 file prob2.c line 5 function lock thread 0
----------------------------------------------------
  l=2 (00000000000000000000000000000010)

Violated property:
  file prob2.c line 34 function lock1
  assertion l != 2
  l != 2

VERIFICATION FAILED
```

Thus, lock1 is accessing the lock() function consecutively (l is the lock status variable).

(c)
No, upon checking with CBMC for iterations of loop up to 20 (without unwinding assertions), lock2 seems to exhibit the proper behavior. I believe we can conclude from this analysis that under no execution locks will be acquired or released consecutively. Thus, the program can be considered as being safe. Increasing the number of times the loop executes (unwinding bound) wouldn't be too useful since only the total and old values determine the number of times that the loop executes, and we can essentially derive a loop invariant off of them. In other words, the sequence of operations that lead to wrong use of locks or unlocks that CBMC is trying to detect would take place irrespective of the number of times we explicitly run the loop. Therefore, a bound of upto 20 can be considered to deduce that the program is safe.

(d)
Wrote code in hw4_2_de.c file. Checked with the following statement (until unwind bound of 300 and it passed successfully each time) -

```
cbmc-5.1/dist/bin/cbmc --function lock2 prob2.c --unwind 300 —no-
unwinding-assertions
```

(e)
No, this does not indicate that the program is correct since you can have a situation where the lock is held before the loop executes, and then the lock is called on it again, which just keep l = 1. And then this would mean that the check for non-consecutiveness of locks of unlock uses would not happen, and only our given assertion is checked for which passes. But the question of correct use of lock and unlock state and the alternative nature of their state transitions has still not been checked with this assertion asked in part (d), thus the program is safe acc to this assertion, however it is not a correct program.