

# 15-122: Principles of Imperative Computation

## QuickCheck 3

February 1, 2013

Today, we'll do the QuickCheck at the beginning of recitation. You will have ten minutes to do this. Your TA will go over answers at the end. Then follow your TA's instructions on whether or not to hand it in

Name:

Andrew ID:

Section (circle one):      A      B      C      D      E      F      G      H

*Simplicio* and *Sagredo* (you) are a couple of 122 students. They are trying to complete an exercise, but are a bit stuck. Help them out!

*Simplicio* is trying to recollect the binary search function taught in lecture yesterday, so that he can understand the details that *Salviati* is going to go over in recitation today. He's got most of the outline, but is missing out on some critical points.

Fill in the blanks and show him that you (*Sagredo*) are awesome at algorithms like this one.

```
1 int binsearch(int x, int[] A, int n)
2 //@requires 0 <= n && n <= \length(A);
3 //@requires _____;
4 /*@ensures (-1 == \result && !is_in(x, A, 0, n))
5     ||
6     _____;
7
8 @*/
9 {
10  int lower = 0;
11  int upper = n;
12  while (lower < upper)
13    //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
14    //@loop_invariant lower == 0 || A[lower-1] < x;
15    //@loop_invariant upper == n || A[upper] > x;
16    {
17      int mid = lower + (upper-lower)/2;
18      if (A[mid] < x) {
19        // We can ignore the bottom half of the array now, since we
20        // know that every thing in that half must be less than x
21        _____;
22      } else if (A[mid] > x) {
23        // We can ignore the upper half of the array, since we know
```

```

22     // that everything in that half must be greater than x
23     _____;
24 } else {
25     // @assert A[mid] == x;
26     _____;
27 }
28 }
29 // @assert lower == upper;
30 return -1;
31 }

```

*Simplicio* wrote two functions, `mult` and `main`, and included contracts to test their correctness, as part of his homework:

```

1 int mult(int x, int y)
2 // @requires x >= 0 && y >= 0;
3 // @ensures |result| == x*y;
4 {
5     int k = x;
6     int n = y;
7     int res = 0;
8     while (n != 0)
9         // @loop_invariant x * y == k * n + res;
10    {
11        if ((k & 1) == 1) res = res + n;
12        k = k >> 1;
13        n = n << 1;
14    }
15    return res;
16 }
17
18 int main()
19 {
20     int a;
21     a = mult(3,4);
22     return a;
23 }

```

Unfortunately, *Simplicio* lost his laptop with his only implementation of C0 contract checking. Homework is due soon and he desperately needs your help to ensure that his contracts are dynamically checked. He already wrote a function to test boolean expressions, `void CHECK(bool b)`, and he marked the places in his code where he thinks it might be useful to call `CHECK` (see next page).

Help *Simplicio* out by adding explicit contract checking to his program. In the solution box, write a call to the `CHECK` function next to the number corresponding to a designated point in *Simplicio's* program. For instance, writing `CHECK(B)` next to `/* 1 */` denotes that the boolean expression `B` should be checked

at position 1 in the program. Note that since contract checking is expensive, you should only call CHECK when needed. Some positions may be left blank.

*Be careful to preserve the relative order of operations.* Assertion checks, loop guard checks, if-statement checks, and assignments should all happen in the same sequence.

```
1 int mult(int x, int y) {
2
3  /* 1 */
4  int k = x; int n = y;
5  int res = 0;
6
7  /* 2 */
8  while (n != 0) {
9    /* 3 */
10   if ((k & 1) == 1) res = res + n;
11   k = k >> 1;
12   n = n << 1;
13   /* 4 */
14  }
15  /* 5 */
16  /* 6 */
17  return res;
18  /* 7 */
19 }
20
21 int main() {
22  int a;
23
24  /* 8 */
25  a = mult(3,4);
26
27  /* 9 */
28  return a;
29 }
```

```
/* 1 */ _____
/* 2 */ _____
/* 3 */ _____
/* 4 */ _____
/* 5 */ _____
/* 6 */ _____
/* 7 */ _____
/* 8 */ _____
/* 9 */ _____
```