Midterm 1 Solutions

15-122 Principles of Imperative Computation

Thursday 2nd October, 2014

Name: Harry Bovik

Andrew ID: bovik

Recitation Section: S

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 5 problems on 10 pages (including 2 blank pages at the end).
- Use a **dark** pen or pencil to write your answers.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.
- You can assume the presence of `#use <util>` and the `arrayutil.c0` library throughout the exam. The interface for some of these functions is repeated at the end of this exam.

	Max	Score
True or false	34	
Contracts	35	
Pixels revisited	30	
Images	24	
Spiral Sort	27	
Total:	150	

1 True or false (34 points)

20pts **Task 1** For each of the following C0 statements, either a) write *always true* if the statement will always evaluate to `true` or b) give *specific, concrete* values of the variables *in either* ***hex*** *or in* ***decimal*** such that the statement will either evaluate to `false` or raise an arithmetic error.

We'll deduct points for writing *always true* if there's a counterexample or for claiming a counterexample if it's always true (but not for blanks), so it's not advantageous to guess.

`x << 4 == x * 4`	x any non-zero number
`x << 2 == x * 4`	always true
`x >> 2 == x / 4`	x any odd negative number
`(~x) + 1 == -x`	always true
`y <= 0 || (x / y) * y == x`	y doesn't evenly divide x (x = 4, y = 3)
`x != -x`	x is 0, `0x80000000`, -2^{31}, -2147483648
`x != ~x`	always true
`-(-x) == x`	always true
`y <= 0 || x / y <= x`	`y > 1`, `x < 0` (x = -12, y = 3)
`x <= x + 1`	x is `0x7FFFFFFF`, 2^{31}-1, 2147483647

14pts **Task 2** Answer `true` or `false` (nothing more) to the following statements. We'll deduct points for incorrect answers (but not for blanks), so it's not advantageous to guess.

Multiplying two numbers in C0 can never cause an arithmetic error to occur.	TRUE
Creating an array in C0 with **`alloc_array`** can never cause an error to occur.	FALSE (allocating a negative-length array)
In the worst case, binary search in an array of size n will run in $O(\log n)$ time.	TRUE
In the worst case, quicksort on an array of size n will run in $O(n \log n)$ time.	FALSE (average case $O(n \log n)$, worst case $O(n^2)$)
$3n + 4 \in O(n)$	TRUE
$6n^{1.5} + n \in O(n^2)$	TRUE
$n \log n \in O(15n)$	FALSE

2 Contracts (35 points)

This function attempts to mimic the `is_sorted(A, lower, upper)` specification function.

It has all sorts of issues.

```
1 bool check_is_sorted(int[] A, int lower, int upper)
2 //@requires 0 <= lower && lower <= upper && upper <= \length(A);
3 {
4     for(int i = 0; i < upper; i++)
5     //@loop_invariant lower <= i && i <= upper;
6     {
7         if (A[i] > A[i+1]) return false;
8     }
9     return true;
10 }
```

If given the 5-element array containing the integers $[1, 2, 3, 3, 2]$, as a first argument, give *specific* values of `upper` and `lower` that meet the preconditions such that, *when compiled and run with contract checking on* (`-d`)...

5pts **Task 1** ...the function will return `false` without failing a contract or accessing an array out of bounds.

> lower = 0, upper = 4

5pts **Task 2** ...the loop invariant will fail.

> lower = 1, upper = 1,2,3,4,5 *or* lower = 2, upper = 2,3,4,5 *or* lower = 3, upper = 3,4,5 *or* lower = 4, upper = 4,5 *or* lower = 5, upper = 5

5pts **Task 3** ...an array will eventually be accessed out of bounds.

> lower = 0, upper = 5

4pts **Task 4** Rewrite line 4 so that the loop invariant is valid, all array accesses are safe, and the function correctly checks that the array is sorted from `lower` (inclusive) to `upper` (exclusive).

```
for (int i = ____lower____; ____i < upper-1____; i++)
```

The questions on this page deal with reasoning about safety. You only need to list line numbers. Do *not* list unnecessary line numbers.

```
1  int test(int[] A, int n)
2  //@requires 0 <= n;
3  //@requires n < \length(A);
4  {
5      int i = 0;
6      while (i < n)
7      //@loop_invariant 0 <= i;
8      //@loop_invariant i <= n;
9      {
10         A[i+1] = A[i+1] + 1;
11         i = i + 1;
12     }
13     return A[i] - 1;
14 }
```

3pts **Task 5** Which line(s) would we need to reference to justify that the loop invariant `0 <= i` on line 7 holds initially?

Just line 5

3pts **Task 6** Which line(s) would we need to reference to justify that the loop invariant `i <= n` on line 8 holds initially?

2 and 5

5pts **Task 7** Which line(s) would we need to reference to justify the safety of the array accesses `A[i+1]` on line 10?

6, 7, and 3

5pts **Task 8** Which line(s) would we need to reference to justify the safety of the array access `A[i]` on line 13?

Alternative 1: 2, 3, 6, and 8 (7 is unnecessary, we already know i == n)
Alternative 2: 3, 7, and 8
(7 tells us `0 <= i`, and the other two tell use `i <= n < \length(A)`, which suffices to show safety).

3 Pixels revisited (30 points)

Recall the interface to pixels:

```
// typedef ______ pixel;

pixel make_pixel(int A, int R, int G, int B)
/*@requires 0 <= A && A < 256; @*/
/*@requires 0 <= R && R < 256; @*/
/*@requires 0 <= G && G < 256; @*/
/*@requires 0 <= B && B < 256; @*/ ;

int get_alpha(pixel P) /*@ensures 0 <= \result && \result < 256; @*/ ;
int get_red(pixel P)   /*@ensures 0 <= \result && \result < 256; @*/ ;
int get_green(pixel P) /*@ensures 0 <= \result && \result < 256; @*/ ;
int get_blue(pixel P)  /*@ensures 0 <= \result && \result < 256; @*/ ;
```

5pts **Task 1** The *inversion* transformation leaves alpha values of pixels untouched, but for the R, G, and B color intensities, it replaces an intensity of 255 with 0, an intensity of 254 with 1, an intensity of 253 with 2... and so on to replacing an intensity of 0 with 255.

Implement the inversion transformation using only numeric constants *written in **hex*** and the numeric operations +, -, and ∗ (you don't need to use all of them!)

```
pixel invert(pixel P) {
    int A = get_alpha(P);
    int R = get_red(P);
    int G = get_green(P);
    int B = get_blue(P);

    return make_pixel(____A____, ____0xFF-R____,

                      ____0xFF-G____, ____0xFF-B____);
}
```

5pts **Task 2** Implement the inversion transformation using only numeric constants *written in **hex*** and the bitwise operations ~, ^, |, and & (you don't need to use all of them!)

```
pixel invert(pixel P) {
    int A = get_alpha(P);
    int R = get_red(P);
    int G = get_green(P);
    int B = get_blue(P);

    return make_pixel(____A____, ____0xFF^R____,

                      ____0xFF^G____, ____0xFF^B____);
}
```

15pts **Task 3** Given the following struct declaration and typedefs, fill in a data structure invariant `is_pixel` and implement `make_pixel` and `get_red`. All functions should be safe, correct, and should provably satisfy their contracts.

```
struct pixel_header {
    int A; // Stores the alpha value
    int R; // Stores the red value
    int G; // Stores the green value
    int B; // Stores the blue value
};
typedef struct pixel_header* pixel;

bool is_pixel(struct pixel_header* P) {

    if (P == NULL) return false;
    if (!(0 <= P->A && P->A < 256)) return false;
    if (!(0 <= P->R && P->R < 256)) return false;
    if (!(0 <= P->G && P->G < 256)) return false;
    if (!(0 <= P->B && P->B < 256)) return false;
    return true;

}

pixel make_pixel(int A, int R, int G, int B)
//@requires 0 <= A && A < 256 && 0 <= R && R < 256;
//@requires 0 <= G && G < 256 && 0 <= B && B < 256;
//@ensures is_pixel(\result);
{

    pixel P = alloc(struct pixel_header);
    P->A = A;
    P->R = R;
    P->G = G;
    P->B = B;
    return P;

}

int get_red(pixel P)
//@requires is_pixel(P);
//@ensures 0 <= \result && \result < 256;
{

    return P->R;

}
```

5pts **Task 4** This implementation of pixels is safe, and it provably satisfies its contracts, but it's *not* correct.

```
bool is_pixel(struct pixel_header* P) {
    return true;
}

pixel make_pixel(int A, int R, int G, int B)
//@requires 0 <= A && A < 256 && 0 <= R && R < 256;
//@requires 0 <= G && G < 256 && 0 <= B && B < 256;
//@ensures is_pixel(\result);
{
    return NULL;
}

int get_red(pixel P)
//@requires is_pixel(P);
//@ensures 0 <= \result && \result < 256;
{
    return 200;
}
```

Write a unit test that *respects the pixel interface* and detects the bug in this implementation by failing an assertion.

```
int main() {

    pixel P = make_pixel(0, 0, 0, 0);
    assert(get_red(P) == 0);

    return 0;
}
```

4 Images (24 points)

In this question, we will consider two versions of the same image, which has *width* of w and *height* of h. The first is an arbitrary image, the second is a version of the original image where *each row* has been sorted by average pixel intensity. Here's one example of such a pair of images:

9pts **Task 1** Using selection sort, the time it would take to produce the image on the right from the image on the left would be in $O(hw^2)$. If this process took exactly 1 second on an image with width 500 and height 500, how long would we expect this sorting process to take...

- ...if the width was 1000 and the height was 500? 4 seconds
- ...if the width was 500 and the height was 1000? 2 seconds
- ...if the width was 1500 and the height was 2000? 36 seconds

15pts **Task 2** For each of the problems below, describe the tightest possible Big-O bounds for the time it would take to solve that problem in the worst case using the algorithms we have discussed in class. Your answer should be in terms of w and h.

	Using the original image like the one above on the left...	Using the sorted image like the one above on the right...
Deciding whether a pixel with a given average intensity i exists anywhere in the image.	$O(hw)$	$O(h \log w)$
Finding the lowest-intensity pixel (the darkest pixel) anywhere in the image.	$O(hw)$	$O(h)$
Finding the row with the lowest average intensity in the image (that is, the on-average darkest row).	$O(hw)$	$O(hw)$

5 Spiral Sort (27 points)

In this problem, we discuss *spiral sort*, a variant of insertion sort. Its chief (and perhaps its only) virtue is that its code it exceedingly short.

```
1  void spiralsort(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures is_sorted(A, 0, n);
4  {
5      for (int i = 0; i < n; i++)
6      //@loop_invariant 0 <= i && i <= n;
7      //@loop_invariant is_sorted(A, 0, i);
8          for (int k = 0; k < i; k++)
9          //@loop_invariant 0 <= k && k <= i;
10         //@loop_invariant is_sorted(A, 0, i);
11         // Another loop invariant will be needed...
12             if (A[i] < A[k])
13                 swap(A, i, k);
14     return;
15 }
```

The loop invariants given above will never fail during the actual evaluation of `spiralsort`, but the loop invariants on the *inner* loop are not strong enough to prove the correctness of the function until we add an additional loop invariant on line 11.

5pts **Task 1** What is the Big-O running time of spiral sort on an array of length n? (When compiled without `-d`, of course.)

$O($ n^2 $)$

5pts **Task 2** Show that we can't reason about the inner loop invariant being preserved: give a value for `k` and contents of an array A such that the loop invariants on lines `9` and `10` hold, the loop guard on line `8` evaluates to `true`, but the loop invariant on line `10` will not hold the next time it is checked.

`i = 3`

`k =` either 1 or 2

`A =`

0	2	3	1	whatever

A[3] must be less than BOTH A[k] and A[k-1]

5pts **Task 3** The loop invariant `A[k-1] <= A[i]` is *almost* right. What would be wrong with adding this as a loop invariant on line 11?

When `k == 0` initially this will cause an array-out-of-bounds access.

5pts **Task 4** Give a better additional loop invariant for the inner loop (which would belong on line 11) that allows us to show that all loop invariants are preserved. You can use functions from `arrayutil.c0` as discussed in class, but this is not necessary.

`k == 0 || A[k-1] <= A[i]`
or `ge_seg(A[i], A, 0, k)`

7pts **Task 5** Taking for granted that the inner loop invariants are true initially and preserved by every iteration of the loop, explain why the outer loop invariants are preserved by every iteration of the outer loop. You'll need to use your answer in part (d).

The first loop invariant is preserved because a single increment of the loop adds one to i, `i < n` before the loop (line 5) so `i' == i+1 <= n` after the loop. `0 <= i` before the loop and `0 <= i'` after the loop; the loop guard sufficies to ensure that we won't run in to overflow. (Worth 2 points if everything else goes wrong.)

Key points for the preservation of the second loop invariant:
At the end of the loop, `k == i` (lines 8 and 9)
So `A[i-1] <= A[i]` or `ge_seg(A[i], 0, i).` (line 11)
We have that `is_sorted(A,0,i)` (line 10)
And the combination of these facts gives `is_sorted(A,0,i+1)`, which is what we need to show, because `i' == i+1`.