Midterm 1 Solutions

15-122 Principles of Imperative Computation
Saquib Razak, Rob Simmons, and Hyrum Wright
February 17, 2015

Name: _______________________________ Harry Bovik _______________________________

Andrew ID: ______ bovik ______

Recitation Section (specify letter or TA): ________ S ________

Instructions

• This exam is closed-book with one sheet of notes permitted.
• You have 80 minutes to complete the exam.
• There are 4 problems on 13 pages (including two blank pages at the end).
• 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> throughout the exam.
1 Short answer (35 points)

10 Task 1 The following questions ask about C0 expressions:

We can write \( \text{int\_min}() \) in hex as \( 0x80000000 \).

We can write 26 in hex as \( 0x1A \).

The result of computing \( \text{int\_max}() + \text{int\_max}() \), in decimal, is -2.

\( 111111101110110110000000 \) can be written in hex as \( 0xFEEDC0 \).

For any \( x \), \( x >> 30 \) is between -2 and 1 (inclusive).

10 Task 2 These four questions are about the definition of big-O.

Fill in the blank, or check the box indicating that it’s not possible to do so.

To prove that \( 2 \times n^2 + 10 \in O(n^2) \), we can set \( n_0 = 1 \) and \( c = 12 \) (or larger).
Check this box if there is no way to fill in the blank above: 

To prove that \( 2 \times n^2 + 10 \in O(n^2) \), we can set \( n_0 = \) and \( c = 2 \).
Check this box if there is no way to fill in the blank above: 

To prove that \( 500 \times n \in O(n - 500) \), we can set \( n_0 = 5 \) and \( c = \) .
Check this box if there is no way to fill in the blank above: 

To prove that \( 100 \times \log_{10} n \in O(n) \), we can set \( n_0 = 3556 \) (or larger) and \( c = 1/10 \).
Check this box if there is no way to fill in the blank above: 

To prove that \( n + 16 \in O(\sqrt{n}) \), we can set \( n_0 = 4 \) and \( c = \).
Check this box if there is no way to fill in the blank above: 

Task 3

Snapshots were taken of an array while a sorting algorithm ran. In the examples below, each line represents a different snapshot, and the snapshots are listed in the order they were taken.

Based on the snapshots, identify the algorithms as selection sort, merge sort, or quicksort.

Sort 1

10 3 0 11 1 9 5 8 7 2 14 12 4 15 6 13
0 1 2 11 3 9 5 8 7 10 14 12 4 15 6 13
0 1 2 3 4 5 6 7 8 9 14 12 11 15 10 13
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Solution: selection sort

Sort 2

10 3 0 11 1 9 5 8 7 2 14 12 4 15 6 13
2 3 0 1 4 9 5 8 7 10 14 12 11 15 6 13
0 1 2 3 4 9 5 8 7 6 10 12 11 15 14 13
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Solution: quicksort

Sort 3

10 3 0 11 1 9 5 8 7 2 14 12 4 15 6 13
0 3 10 11 1 5 8 9 7 2 14 12 4 15 6 13
0 1 3 5 8 9 10 11 2 7 12 14 4 6 13 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Solution: merge sort

Task 4

Imagine you have an array $A$ of $x$, arrays, each containing $y$ integers. That is, you have $\text{int} [] [] A, \text{\length}(A) = x$ and $\text{\length}(A[i]) = y$ for every $0 \leq i < x$.

How long would it take (in total) to use insertion sort on each of the $x$ arrays in turn, so that you have $x$ sorted arrays?

$O(xy^2)$

Once you had $x$ sorted arrays, how long would it take to find the smallest integer anywhere in any of the arrays?

$O(x)$
2 Contracts (30 points)
This question deals with using contracts and loop invariants to show safety and correctness. We will use the specification function $POW$ from lecture:

```c
int POW(int base, int exp)
//@requires 0 <= exp;
{
    if (exp == 0) return 1;
    return base * POW(base, exp - 1);
}
```

Task 1 For this task, you’ll reason about safety and correctness of the following function. Give only line numbers, don’t include unnecessary line numbers.

```c
/* 1 */ int fill(int x, int y)
/* 2 */ //@requires x >= 0;
/* 3 */ //@requires y > 0;
/* 4 */ //@ensures \result == POW(x, y-1);
/* 5 */ {
/* 6 */ int i = 0;
/* 7 */ int z = 0;
/* 8 */ int[] A = alloc_array(int, y);
/* 9 */
/* 10 */ while (i < y-1)
/* 11 */ //@loop_invariant 0 <= i;
/* 12 */ //@loop_invariant i < y;
/* 13 */ //@loop_invariant A[i] == z;
/* 14 */ //@loop_invariant z == POW(x, i) - 1;
/* 15 */ {
/* 16 */ z = (A[i] + 1) * x;
/* 17 */ i++;
/* 18 */ A[i] = z;
/* 19 */ }
/* 20 */
/* 22 */ return z + 1;
/* 23 */ }
```

To prove line 8 safe, we use line(s) 3

To prove line 13 safe, we use line(s) 8, 11, 12

To prove line 14 safe, we use line(s) 11

To prove the postcondition, we use lines 10, 12, 14 (optional: 22)
(In this last case, assume safety of the entire function, assume all loop invariants are true initially and are preserved by an arbitrary iteration of the loop)
Task 2 Given these loop invariants, give a simple implementation of reverse_dec, decimal reversal of a seven-digit decimal number. Do not use any specification functions.

```c
int reverse_dec(int x)
//@requires 0 <= x && x < POW(10,7);
//@ensures 0 <= \result && \result < POW(10,7);
{
    int res = 0;
    for (int i = 0; i < 7; i++)
        //@loop_invariant 0 <= x && x < POW(10, 7-i);
        //@loop_invariant 0 <= res && res < POW(10, i);
        {
            res = res * 10 + x % 10;
            x = x / 10;
        }
    //@assert x == 0;
    return res;
}
```

Task 3 Given this implementation of reverse_dec, give loop invariants that allow us to prove safety and correctness for all contracts.

Hints: The first missing loop invariant should establish safety of the other loop invariants. The last loop invariant is very challenging! Consider coming back to that one.

```c
int reverse_dec(int x)
//@requires 0 <= x && x < POW(10,7);
//@ensures 0 <= \result && \result < POW(10,7);
{
    int res = 0;
    int old = 1;
    int new = 10000000;
    for (int i = 0; i < 7; i++)
        //@loop_invariant 0 <= res && res <= POW(10,7);
        //@loop_invariant 0 <= i && i <= 7;
        //@loop_invariant old == POW(10, i);
        //@loop_invariant new == POW(10, 7-i);
        //@loop_invariant res % POW(10, 7-i) == 0;
        {
            int digit = (x / old) % 10;
            old = old * 10;
            new = new / 10;
            res = res + digit * new;
        }
    return res;
}
3 Sets (30 points)
A set is a mutable data structure that can store integers that are unique (no duplicates) and have no specific order. Following is a definition of a set interface that can hold integers:

```c
typedef ______* set_t;
```

// Returns true when the set has no elements
```c
bool empty(set_t S) /*@requires S != NULL; @*/;
```

// Checks whether n is present in the set S
```c
bool member(set_t S, int n) /*@requires S != NULL; @*/;
```

// Create a new, initially empty, set
```c
set_t new() /*@ensures \result != NULL; @*/;
```

// Adds the integer n to the set S.
// Does nothing if n already exists in set s
```c
void add(set_t S, int n) /*@requires S != NULL; @*/;
```

// Removes n from the set S.
// The integer MUST be in the set already.
```c
void delete(set_t S, int n) /*@requires S != NULL; @*/;
```

// Remove and return an arbitrary element
```c
int rem(set_t S) /*@requires S != NULL && !empty(S); @*/;
```

5 Task 1 Based on the descriptions above, give at least one meaningful additional postcondition that we could add to each of these three functions in our interface.
Don’t write postconditions that could not actually be violated in any implementation. Use the C0 //@ensures notation in your answer.

```c
set_t new()
//@ensures \result != NULL;
//@ensures empty(\result);

void add(set_t S, int n)
//@requires S != NULL;
//@ensures member(S, n);
//@ensures !empty(S);

int rem(set_t S)
//@requires S != NULL && !empty(S);
//@ensures !member(S, \result);
```
**Task 2** A fundamental operation on sets is intersection. Let’s say we have two sets $A$ and $B$. Then the intersection of sets $A$ and $B$, written as $A \cap B$, will be a set that includes exactly the elements are shared between the two sets.

\[
\begin{align*}
\{1, 2\} \cap \{1, 2\} &= \{1, 2\} \\
\{1, 2, 3\} \cap \{3, 5, 4\} &= \{3\} \\
\{1, 2\} \cap \{3, 4\} &= \{
\end{align*}
\]

Implement set intersection using the functions given in the set interface above. You don’t have to use every line we gave you in the while loop.

When the function returns, both sets should have the same contents they had originally.

```c
set_t intersection(set_t A, set_t B) {
//@requires A != NULL && B != NULL;
//@ensures \result != NULL;
{
    set_t result = new();
    set_t temp = new();

    while (!empty(A)) {
        int x = rem(A);

        add(temp, x);

        if (member(B, x)) {
            add(result, x);
        }
    }

    while(!empty(temp)) {
        add(A, rem(temp));
    }

    return result;
}
```
**Task 3** A set is a *singleton* if it contains exactly one element. The sets \{1\} and \{-42\} are singletons, the sets \{} and \{0,5,9\} are not. Use the set interface to implement a singleton check.

*For full credit, don’t add any control structures (if, while, for, etc.) other than the if statement we gave you at the beginning. You don’t have to use every line.*

When the function returns, the set should have the same contents it had originally.

```c
bool singleton(set_t S)
//@requires S != NULL;
{
    if (empty(S)) return false;
    int x = rem(S);
    bool ans = empty(S);
    add(S, x);
    return ans;
}
```

**Task 4** Two sets are *disjoint* in they have no common elements. Write a function `disjointSets` that returns true if the two sets passed as input parameters are disjoint.

In addition to the set interface, you can use the functions described in previous tasks, even if you didn’t complete those tasks.

*For full credit, your answer should contain a single return statement and should fit on one line.*

Obviously, you don’t have to use every line.

When the function returns, both sets should have the same contents they had originally.

```c
bool disjointSets(set_t A, set_t B)
//@requires A != NULL && B != NULL;
{
    return empty(intersection(A,B));
}
```
4 Coordinates (30 points)
In the homework, we used a 32-bit C integer to store four eight-bit unsigned quantities, with the alpha component stored in the eight high-order bits and the blue component stored in the eight low-order bits:

\[
\begin{align*}
a_0 & a_1 a_2 a_3 a_4 a_5 a_6 a_7 \\
r_0 & r_1 r_2 r_3 r_4 r_5 r_6 r_7 \\
g_0 & g_1 g_2 g_3 g_4 g_5 g_6 g_7 \\
b_0 & b_1 b_2 b_3 b_4 b_5 b_6 b_7 \\
\end{align*}
\]

(unsigned) (unsigned) (unsigned) (unsigned)

In this question, we will use a 32-bit C integer to store a four-dimensional coordinate; a series of coordinates can be used to tracking an object through 3-dimensional space over a period of time.

We want the spatial coordinates, \(x\), \(y\), and \(z\), to be able to be either positive or negative. Time will always be nonnegative. Therefore, the three spatial \(x\), \(y\), and \(z\) coordinates will be treated as seven-bit, signed two’s complement quantities. The \(t\) (time) coordinate will be treated as an eleven-bit unsigned quantity.

\[
\begin{align*}
x_0 & x_1 x_2 x_3 x_4 x_5 x_6 \\
y_0 & y_1 y_2 y_3 y_4 y_5 y_6 \\
z_0 & z_1 z_2 z_3 z_4 z_5 z_6 \\
t_0 & t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} \\
\end{align*}
\]

(signed) (signed) (signed) (unsigned)

The interface to the coordinate type is much like the interface to the pixel type:

```c
// typedef _____ coord_t;
typedef int coord_t;
coord_t make_coord(int x, int y, int z, int time);
int get_x_coord(coord_t C);
int get_y_coord(coord_t C);
int get_z_coord(coord_t C);
int get_time_coord(coord_t C);
```

This interface is incomplete: in particular, these functions are missing preconditions and post-conditions.

### Task 1
Give reasonable postconditions for get_x_coord and get_time_coord:

```c
int get_x_coord(coord_t C)
//@ensures -64 <= \result && \result < 64; // 64 == 0x40
// 63 == 0x3F
```

```c
int get_time_coord(coord_t C)
//@ensures 0 <= \result && \result < 2048; // 2048 == 0x800
// 2047 == 0x7FF
```

Do we need to attach the postcondition

//@ensures 0 <= \result && \result <= int_max()

to the make_coord function in the client interface above? Briefly, why or why not?

**Solution:** No.
Reason 1: this violates the interface (we’re exposing that coord_t is an integer).
Reason 2: this is not always true - in particular, it’s false when \(x\) is an negative integer.
**Task 2** Complete the following partial implementation. *Except for the last three blanks, every blank line should be filled in with a single integer constant, either in **hexadecimal** or in **decimal**.*

```c
int get_x_coord(coord_t C) {
    return (C & 0xFE000000) >> 25;
    // Any mask s.t. mask & 0xFE000000 == 0xFE000000 is fine
    // In particular, -1 is fine as a mask
}

int get_y_coord(coord_t C) {
    C = C << 7;
    return C >> 25;
}

int get_z_coord(coord_t C) {
    C = (C >> 11) & 0x7F;
    if (C > 63) {
        return C - 128;
        // or C | 0xFFFFFFFF80; (-128)
        // or C | 0xFFFFFFFFC0; (-64)
    } else {
        return C;
    }
}

int get_time_coord(coord_t C) {
    return C & 0x7FF;
}
```
This implementation of `make_coord` has a bug:

```c
coord_t make_coord(int x, int y, int z, int t)
// Contracts omitted
{
    return (x << 25) | (y << 18) | (z << 11) | t;
}
```

Write a unit test that exposes this bug. It should return 0 on a proper implementation of the `coord` interface, but should fail an assertion when run with the `make_coord` implementation above.

*For full credit, respect the interface (do not rely on details of the way coord_t is implemented) and do not call any functions from the coord interface except for `make_coord()` and `get_y_coord()`. You can still use built-in functions like `assert()`. Make the test simple – you do not need to use all the space we’ve given you.*

```c
int main() {
    coord_t C = make_coord(5, 3, -1, 10);
    assert(get_y_coord(C) == 3);
    return 0;
}
```