

Midterm I Exam

15-122 Principles of Imperative Computation
André Platzer Ananda Gunawardena

February 23, 2012

Name: **Sample Solution** Andrew ID: **aplatzer** Section:

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 4 problems.
- 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.
- And most importantly,

DON'T PANIC!

	Mod.arith.	Safari	Contracts	Big O	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score	20	35	25	20	100
Max	20	35	25	20	100
Grader					

1 Modular Arithmetic (20 pts)

In C0, values of type `int` are defined to have 32 bits. In this problem we work with a version of C0 where values of type `int` are defined to have only 7 bits. In other respects it is the same as C0. All integer operations are still in two's complement arithmetic, but now modulo 2^7 . All bitwise operations are still bitwise, except on only 7 bit words instead of 32 bit words.

Task 1 (10 pts). Fill in the missing quantities, in the specified notation.

- The minimal negative integer, in decimal: -64
- The maximal positive integer, in decimal: 63
- 4, in hexadecimal: 0x 7C
- 44, in hexadecimal: 0x 2C
- 0x48, in decimal: -56

Task 2 (10 pts). Assume `int x` and `int y` have been declared and initialized to unknown values. For each of the following, indicate if the expression always evaluates to `true`, or if it could sometimes be `false`. In the latter case, indicate a counterexample in the C0 dialect described here by giving a value for `x` and `y` that falsifies the claim. You may use decimal or hexadecimal notation.

- `x >= x - 1` false, x = MIN_INT=-64
- `~(x ^ (~x)) == -1` false for all x
- `x+(y+1)-2*(x-1)-3 == -x+y` true
- `(x!=-x || y!==-y) || x==y` false, x=0, y=MIN_INT=-64 or x=-64, y=0
- `x <= (1<<(7-1))-1` true

2 Wild Elephant Safari (35pts)

Your computer is on safari to find an elephant in Africa. It assumes that Africa corresponds to a rectangular array represented as a one-dimensional array A . In the array A , the presence of an elephant is marked by number 7. The program on the next page looks at every element of the array from north to south and, on each row, from west to east. Unfortunately, it fails to find the elephant, even though *there is at least one elephant in Africa*.

The reason is that *elephants may move while your computer is searching for them*. An elephant may move to a position that your program already looked at in an earlier iteration. A computational zoologist confirms that elephants may move at every step of your function implementation that executes an assignment or increment/decrement (like $i++$ or $i--$). Elephants move either 1 cell in the north-south direction or 1 cell in the east-west direction, which concurrently changes the memory contents of array A by swapping an elephant to a neighbour cell. The computational zoologist also knows that *elephants never travel farther than 1 move away from their home cell!*

Hint: You decide to change your program so that when it checks for elephants at a cell, it also looks for the elephant in the immediate neighbor cells, just in case the elephant has moved. The computational zoologist warns you that you need to look around to neighboring cells without using assignments or increment/decrements or else the elephant may move away again.

Task 1 (5 pts). Fill in the requires and ensures clauses of the `safari` function (see next page).

Task 2 (5 pts). Fix the loop invariants in the `safari` function if they are broken.

Task 3 (20 pts). The `safari` function is broken. Fix the implementation of the `safari` function (see next page), so that it always finds an elephant. Make sure you clearly add a new statement, and clearly modify or delete a given statement, or clearly rewrite the implementation as needed.

Task 4 (5 pts). What is the worst-case asymptotic complexity of your `safari` implementation as a function of width and height, in big-O notation? Please briefly justify your answer.

$O(\text{width} * \text{height})$, because the function performs a constant number of operations for each i and j , which iterate from $0, \dots, \text{width}$ and from $0, \dots, \text{height}$, respectively, in two nested loops.

```

typedef int elephants;
int safari(elephants[] A, int width, int height)
//@requires \length(A) >= width*height && width >= 0 && height >= 0;
//@requires is_in(7, A, width*height);
//@ensures A[\result] == 7;
{
    for (int i = 0; i < height; i++)
        //@loop_invariant 0 <= i && i <= /**/ height;
        {
            for (int j = 0; j < width; j++)
                //@loop_invariant 0 <= j && j <= /**/ width;
                {
                    int r = i * width;
                    int b = r + j;
                    if (A[b] == 7) return b;
                    if (j > 0 && A[b - 1] == 7) return b - 1;
                    if (j < width - 1 && A[b + 1] == 7) return b + 1;
                    if (i > 0 && A[b - width] == 7) return b - width;
                    if (i < height - 1 && A[b + width] == 7) return b + width;
                }
            }
        // there is an elephant in Africa, so cannot get here
        // because we sweep 1 move around the position without assigns/increments/decrements
        //@assert false;
        return -1;
}

```

```

//Alternative: Recursive linear search on A in assignment-free form also ok.
int recsearch(elephants[] A, int i, int n)
//@requires \length(A) >= n && n >= i && i >= 0;
//@requires is_in(7, A, width*height);
//@ensures A[\result] == 7;
{
    if (i >= n) return -1;
    if (A[i] == 7) return i;
    else return recsearch(A, i+1, n);
}

int safari(elephants[] A, int width, int height)
//@requires \length(A) >= width*height && width >= 0 && height >= 0;
//@requires is_in(7, A, width*height);
//@ensures A[\result] == 7;
{ return recsearch(A, 0, width*height); }

```

3 Contracts Office (25pts)

In this problem, you are given contracts and want to find “contract exploits”. For each of the contracts in the following tasks do:

- *Spot weaknesses* in the contract and briefly explain what the contract is missing to capture the informally stated purpose (one sentence explanation is sufficient).
- *Implement a simple contract exploit function*, i.e., a function that always satisfies the given contract without actually solving the informally stated problem.
Give simple code! If your code is nontrivial (more than 3 lines), you need to provide loop invariants and a correctness argument to show why it actually satisfies the contract.
- Briefly sketch your approach on *how the contract can be fixed* (you do not need to give a full bug-fixed contract, a brief one sentence description of your approach is sufficient).

Hint: The following contract for integer square root is too weak, because the trivial implementation `{ return 0; }` is a contract exploit that clearly does not compute the square root:

```
int isqrt(int n)
//@requires n >= 0;
//@ensures 0 <= \result*\result && \result*\result <= n;
```

Task 1 (5 pts). The following contract was intended to “describe a fancy operation on two arrays to give a new array”. The computation itself is so difficult and irrelevant, so only the array contract is important for this task.

Hint: Ignore what the fancy array operation might have been supposed to do. Try to implement the contract in $O(1)$.

```
int[] fancyarrayop(int[] A, int n, int[] B)
//@requires n > 0;
//@ensures \result[0] >= 0;
{
    return alloc_array(int, 1);
}
```

The contract does not look useful, because the `//@requires` contract leaves `\length(A)` and `\length(B)` unspecified, making it impossible for the function to access `A` or `B` safely. This effectively makes the arguments `A` and `B` useless and, thus, superfluous.

Suggested fix: add `\length(A)` and `\length(B)` constraints into `@requires`, preferably in relation to `n`, e.g.,

```
//@requires 0<=n && n <= \length(A) && n <= \length(B);
```

Task 2 (10 pts). The following contract was intended to specify a “super fast sort function”.

Hint: Ignore that the intention was to sort the input. Try to implement the contract super fast, i.e., faster than $O(n \log n)$.

```
int[] supersort(int[] A, int u)
//@requires 0 <= u && u <= \length(A);
//@ensures \length(\result) == u && is_sorted(\result, 0, u);
{
    return alloc_array(int, u);
}
// or
{
    for (int i = 0; i < u; i++)
        A[i] = 0;
    return A;
}
```

The contract is not exhaustive, because it does not rule out implementations that ignore and destroy the input data and result in an array that is sorted but contains different data. Suggested fix: Add contract ensuring that `\result` is a permutation of `A`.

Task 3 (10 pts). The following contract was intended to specify a “super fast find function”.

Hint: Ignore that the intention was to find x in the array. Try to implement the contract super fast, e.g., faster than $O(\log n)$.

```
int supersonicfind(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, n);
/*@ensures (-1 == \result && !is_in(x, A, n))
           || ((0 <= \result && \result < n) && A[\result] == x); @*/
{
    if (n == 0) return -1;
    A[0] = x;
    return 0;
}
```

The contract is not quite exhaustive, because it does not rule out implementations that ignore and overwrite the original input data. Suggested partial fix: at least `//@ensures A[\result]==\old(A[\result])`

4 Big-O (20pts)

Task 1 (10 pts). Define the big- O notation

$f(n) \in O(h(n))$ if and only if there is an n_0 and $c > 0$ such that $f(n) \leq c * h(n)$ for all $n \geq n_0$

and briefly state the two key ideas behind this definition in two sentences and briefly explain why and how it captures those key ideas:

In the mathematical analysis of function complexity, our main concern is the asymptotic behavior of functions on *larger and larger inputs* (the outermost quantifier for all big enough inputs n_0). Furthermore, we reason at a high level of abstraction, *ignoring constant factors* (the quantifier for a constant $c > 0$, which makes sure not to distinguish two functions that are within a constant factor of each other, since $c*$ can be on either side of the comparison, equivalently).

Task 2 (10 pts). For each of the following, indicate if the statement is true or false.

(a) $O(n^2 + 1024n \log n + 10^{10}) = O(5n^2 - 1)$ **true**

(b) $O(n * \log(n)) \subset O(n)$ **false**

(c) $O(n) \subset O(n * \log(n))$ **true**

(d) $O(3 * \log_2 n) = O(2 * \log_3 n)$ **true**

(e) $O((10 \log n + 3 * n) * n^2) \subset O(\log(n^3))$ **false**

Extra Space if Needed

If you decide to use this extra space, clearly mark to which problem your answers here belong!