Exam I

15-122 Principles of Imperative Computation, Summer 2011
William Lovas

May 27, 2011

Name: **Sample Solution** Andrew ID: **wlovas**

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.	Search	Quicksort	Big-O	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score	**30**	**40**	**40**	**40**	**150**
Max	30	40	40	40	150
Grader					

1 Modular Arithmetic (30 pts)

In C0, values of type `int` are defined to have 32 bits. In this problem we work with a version of C0 called C8 where values of type `int` are defined to have only 8 bits. In other respects it is the same as C0. All integer operations are still in two's complement arithmetic, but now modulo 2^8. All bitwise operations are still bitwise, except on only 8 bit words instead of 32 bit words.

Task 1 (15 pts). Fill in the missing quantities, in the specified notation.

(a) The minimal negative integer, in decimal: ___-128___

(b) The maximal positive integer, in decimal: ___127___

(c) -5, in hexadecimal: ___0x FB___

(d) 23, in hexadecimal: ___0x 17___

(e) `0x36`, in decimal: ___54___

Task 2 (15 pts). Assume `int x` has been declared and initialized to an unknown value. 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 C8 by giving a value for `x` that falsifies the claim. You may use decimal or hexadecimal notation.

(a) `x+1 > x` ___false, x = 127___

(b) `((x<<1)>>1) | (x & 0x80) == x` ___false, x = 0x40___

(c) `x ^ (~x) == -1` ___true___

(d) `x <= 1<<7` ___false, x = 0 (or any x other than 1<<7)___

(e) `x+x == 2*x` ___true___

2 Binary Search (40 pts)

Consider a recursive implementation of binary search. The main `binsearch` function calls a recursive helper function:

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

Task 1 (10 pts). Complete the recursive `bsearch` helper function.

```
int bsearch(int x, int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@requires is_sorted(A, lower, upper);
/*@ensures (\result == -1 && !is_in(x, A, lower, upper))
        || (lower <= \result && \result < upper && A[\result] == x);
  @*/
{
    if (upper == lower) return ______ -1 _____________________ ;  /* (a) */

    int mid = lower + (upper - lower) / 2;

    if (A[mid] == x)

        return ______ mid ___________________________________ ;  /* (b) */

    else if (A[mid] < x)

        return ______ bsearch(x, A, mid+1, upper) ____________ ;  /* (c) */

    else

        return ______ bsearch(x, A, lower, mid) ______________ ;
}
```

Task 2 (20 pts). Argue that your code satisfies the postcondition given for `bsearch`.

(a) Argue that the return statement marked `(a)` satisfies the postcondition.

At `(a)` we return `-1`, so we must show `!is_in(x, A, lower, upper)`.

When `upper == lower`, the subarray under consideration is empty, which implies `!is_in(x, A, lower, upper)`.

(b) Argue that the return statement marked `(b)` satisfies the postcondition.

At `(b)` we return `mid`, so we must show `lower <= mid && mid < upper && A[mid] == x`, where `mid == lower + (upper - lower) / 2`.

We have `A[mid] == x` by the given `if`-condition.

We have `lower < upper` by the precondition and the negation of the first `if`-statement. From this, we get `0 < upper - lower`, or `1 <= upper - lower`, which gives `0 <= (upper - lower) / 2`. Adding `lower` to both sides gives us `lower <= lower + (upper - lower) / 2)`, i.e., `lower <= mid`.

From `0 < upper - lower`, we also know that `(upper - lower) / 2 < upper - lower`. Adding `lower` to both sides gives us `lower + (upper - lower) / 2 < upper`, i.e., `mid < upper`.

(Continued)

(c) Argue that the return statement marked `(c)` satisfies the postcondition. (**Hint:** there will be two cases.)

In order to call `bsearch(x, A, mid+1, upper)`, we need to establish its precondition: `0 <= mid+1 && mid+1 <= upper && upper <= \length(A)`. We argued above that `lower <= mid && mid < upper`, which gives `lower <= mid+1 && mid+1 <= upper`. The "outer" precondition tells us `0 <= lower` and `upper <= \length(A)`, so the call is permitted.

After the call to `bsearch` returns, we obtain its postcondition:

```
   (\result == -1 && !is_in(x, A, mid+1, upper))
|| (mid+1 <= \result && \result < upper && A[\result] == x)
```

We proceed by cases.

In the first case, we return `-1` so we must establish `!is_in(x, A, lower, upper)`. This follows from `!is_in(x, A, mid+1, upper)` and the `if`-condition `A[mid] < x`.

In the second case, we return a `\result` for which we know `mid+1 <= \result && \result < upper && A[\result] == x`. It remains only to be shown that `lower <= \result`; this follows from `lower <= mid`, which we argued above.

Task 3 (10 pts). Argue that your code terminates by arguing that every recursive call has "smaller" inputs than the function itself, for some appropriate meaning of "smaller". (Your argument may appeal to intuition using pictures.)

We have `lower < upper` by the precondition and the negation of the first `if`-statement, meaning our interval is non-empty. The intervals on either side of the `mid` point, excluding the `mid` point itself, are both smaller than the entire interval, i.e., the recursive calls are on strictly smaller intervals.

3 Quicksort (40 pts)

The implementation of quicksort we saw in lecture chooses the middle element as the pivot.

Task 1 (5 pts). Give an array of 5 distinct elements that exhibits worst-case $O(n^2)$ behavior. That is, at each partitioning step, one of the segments is empty.

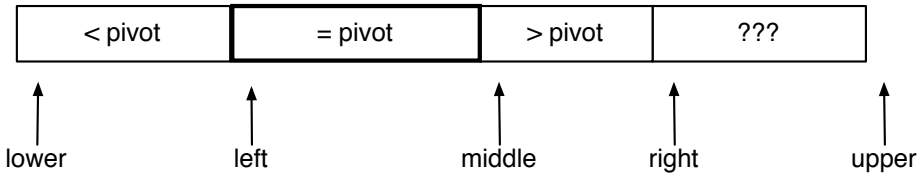
The array $[2, 4, 5, 3, 1]$ is one such array: at each partitioning step, the largest remaining element is selected as the pivot.

Even an implementation of quicksort that chooses a random pivot can have deterministic worst-case inputs.

Task 2 (5 pts). Explain why even a randomized version of our algorithm has $O(n^2)$ running time if all the array elements are the same.

If all the array elements are the same, then every element is an equally bad pivot: all other elements will wind up in the "$\geq pivot$" segment of the partition, and the "$< pivot$" segment will always be empty.

Consider an alternative implementation of quicksort that partitions the current subarray into three segments: those *less* than the pivot, those *equal* to the pivot, and those *greater* than the pivot. We maintain three indices, *left*, *middle*, and *right*, and the general case of the loop invariant can now be pictured as follows, where the bold segment must be non-empty.



The partition function will now need to return not just an index but an entire *range* representing the segment of the partition containing elements equal to the pivot. (Since a pivot must exist, this range must be non-empty.) We represent a range as a pointer to a struct containing two `int` fields, *start* and *end*:

```
struct range {
    int start;
    int end;
};
```

A variable `r` of type `struct range *` represents the half-open interval [`r->start` .. `r->end`).

Using this representation of a range, the three-way `tripartition` function may be specified as follows:

```
struct range * tripartition(int[] A, int lower, int upper)
//@requires 0 <= lower && lower < upper && upper <= \length(A);
//@ensures \result != NULL;
/*@ensures lower <= \result->start
       && \result->start < \result->end    // NB: result range non-empty
       && \result->end <= upper;
  @*/
//@ensures gt(A[\result->start], A, lower, \result->start);
//@ensures eq(A[\result->start], A, \result->start, \result->end);
//@ensures lt(A[\result->start], A, \result->end, upper);
;
```

Task 3 (20 pts). Fill in the loop invariants and code for the `tripartition` function below. You should ensure—but need not prove—that your code satisfies your loop invariants and that your loop invariants imply the postcondition. (**Hint:** draw pictures!)

(Continued)

```
struct range * tripartition(int[] A, int lower, int upper)
{
    int pivot_index = lower + (upper - lower) / 2;
    int pivot = A[pivot_index];
    swap(A, lower, pivot_index);

    int left = lower;
    int middle = lower + 1;
    int right = lower + 1;

    while (right < upper)
                              lower <= left && left < middle
    //@loop_invariant ____ && middle <= right && right <= upper _____________ ;

    //@loop_invariant ____ gt(pivot, A, lower, left) ___________________________ ;

    //@loop_invariant ____ eq(pivot, A, left, middle) ________________________ ;

    //@loop_invariant ____ lt(pivot, A, middle, right) _______________________ ;
    {
        if (pivot < A[right]) {

            ____ right++ ________________________________ ;

        } else if (pivot == A[right]) {

            ____ swap(A, middle, right) ______________ ;

            ____ middle++ ________________________________ ;

            ____ right++ ________________________________ ;

        } else /*@assert pivot > A[right]; @*/ {

            ____ swap(A, middle, right) ______________ ;

            ____ swap(A, left, middle) _______________ ;

            ____ left++ _________________________________ ;

            ____ middle++ ________________________________ ;

            ____ right++ ________________________________ ;
```

```
        }
    }

    //@assert right == upper;
    struct range * r = alloc(struct range);

    r->start = ____ left ________________ ;

    r->end   = ____ middle _____________ ;

    return r;
}
```

Task 4 (10 pts). Complete the following `triqsort` function that uses `tripartition` to sort an array.

```
void triqsort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{

    if ( ____ upper - lower <= 1 ________________________ ) return;

    struct range * r = ____ tripartition(A, lower, upper) _____________ ;

    triqsort(A, ____ lower _____________ , ____ r->start __________ );

    triqsort(A, ____ r->end _____________ , ____ upper _____________ );

}
```

4 Big-O (40 pts)

Task 1 (15 pts). Define the big-O notation

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

and briefly state the two key ideas behind this definition in two sentences:

> In the mathematical analysis of function complexity, our main concern is the asymptotic behavior of functions on *larger and larger inputs*. Furthermore, we reason at a high level of abstraction, *ignoring constant factors*.

Task 2 (15 pts). For each of the following, indicate if the statement is true or false.

(a) $O(n^2 + 1024n + 32) = O(31n^2 - 34)$ **true**

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

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

(d) $O(32) = O(2^{32})$ **true**

(e) $O(2^n) = O(2^{2^n})$ **false**

Task 3 (10 pts). You observe the following timings when executing an implementation of sorting on randomly chosen arrays of size n. Form a conjecture about the asymptotic running time of each implementation.

A	
n	time (s)
2^{15}	10.23
2^{16}	20.51
2^{17}	41.99
2^{18}	85.27

$O(\underline{\quad n \quad})$

B	
n	time (s)
2^{15}	22.36
2^{16}	90.55
2^{17}	368.97
2^{18}	1723.03

$O(\underline{\quad n^2 \quad})$

C	
n	time (s)
2^{15}	2.01
2^{16}	5.03
2^{17}	12.77
2^{18}	29.93

$O(\underline{\quad n * \log(n) \quad})$

Which would be preferable for inputs of about 1 million elements? Circle one:

A B C **C**