Exam I

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

May 27, 2011

Name: Andrew ID:

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,

$\boxed{\textbf{DON'T PANIC!}}$

	Mod.arith.	Search	Quicksort	Big-O	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score					
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: __________

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

(c) -5, in hexadecimal: 0x__________

(d) 23, in hexadecimal: 0x__________

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

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` ____________________

(b) `((x<<1)>>1) | (x & 0x80) == x` ____________________

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

(d) `x <= 1<<7` ______________________________

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

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 _______________________________ ;  /* (a) */

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

    if (A[mid] == x)

        return _______________________________________________ ;  /* (b) */

    else if (A[mid] < x)

        return _______________________________________________ ;  /* (c) */

    else

        return _______________________________________________ ;
}
```

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.

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

(Continued)

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

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.)

3 Quicksort (40 pts)

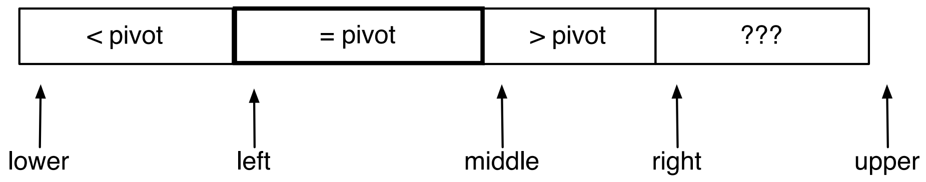
The implementation of quicksort we saw in lecture chooses the middle element as the pivot.

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.

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.

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)

    //@loop_invariant _______________________________________________________ ;

    //@loop_invariant _______________________________________________________ ;

    //@loop_invariant _______________________________________________________ ;

    //@loop_invariant _______________________________________________________ ;
    {
        if (pivot < A[right]) {

            _____________________________________________ ;

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

            _____________________________________________ ;

            _____________________________________________ ;

            _____________________________________________ ;

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

            _____________________________________________ ;

            _____________________________________________ ;

            _____________________________________________ ;

            _____________________________________________ ;

            _____________________________________________ ;
        }
    }                                    /* (code continues on following page) */
    //@assert right == upper;
    struct range * r = alloc(struct range);

    r->start = ____________________________ ;

    r->end   = ____________________________ ;

    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 ( ________________________________________________ ) return;

    struct range * r = ________________________________________________ ;

    triqsort(A, __________________________ , __________________________ );

    triqsort(A, __________________________ , __________________________ );

}
```

4 Big-O (40 pts)

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

$f(n) \in O(g(n))$ if and only if ____

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

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

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

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

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

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

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

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

B

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

C

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

$O($____________$)$ $O($____________$)$ $O($____________$)$

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

A B C