Midterm 1 Solutions

15-122 Principles of Imperative Computation

Thursday 6th October, 2016

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 20 pages (including 0 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
Short answer	25	
Predictive Search	25	
Shuttle Sort	25	
Max-Stacks	25	
Min-Stacks *(makeup)*	25	
Total:	125	

1 Short answer (25 points)

10pts **Task 1** In the C0 expressions below, assume that `x` and `y` are **both strictly positive** integers and that `POW(a, b)` calculates a^b as described in class, with the precondition that `b >= 0`.

For each expression, either…

1. write "*always true*" if the statement will evaluate to `true` whenever `x` and `y` are **both strictly positive**

   … or …

2. give *specific, concrete, and **strictly positive*** values of the relevant variables *in either **hex** or **decimal*** such that the statement will either evaluate to `false`, raise an arithmetic error, or fail a precondition.

`x > 0 && y > 0`	*always true*
`x >> 4 == x / 8`	$x = 17$
`POW(3, POW(2, y)) >= 0`	$y = 31$
`y * x + 25 * y == y * (25 + x)`	*always true*
`(x / y) * y + x % y == x`	*always true*
`0 <= x / y && x / y < x`	$y = 1$

6pts **Task 2** These questions are about the definition of big-O. Recall that $f(n) \in O(g(n))$ if there exists a $c > 0$ and $n_0 > 0$ such that forall $n \geq n_0$, $f(n) \leq cg(n)$.

In each box below indicate a value of c and n_0 that makes the statement true. If none exist, write **NONE**.

$2n^2 + 3n + 1 \in O(n^2)$	c = $c = 4$	n_0 = $n_0 = 3$
$n^2 + 2 \in O(15122n)$	c = NONE	n_0 = NONE

9pts **Task 3** In the following boxes, give an assertion that allows you to prove that the lines following it are safe. You should assume that `y`, `k`, and `p` are initialized and have the correct types, but that you don't know their values.

The assertion should itself be safe (it shouldn't be possible for the assertion to do something besides return `true` or `false`). The assertion should also be as *weak* as possible: it should only fail when the lines following it would definitely cause an error.

```
//@assert ____________p != NULL && *p != NULL____________;

return **p;
```

```
//@assert ____________y != 0 && y != -1____________;

return int_min() / y;
```

```
//@assert ____________k >= 2____________;

int[] A = alloc_array(int, k);
return A[k-2];
```

2 Predictive Search (25 points)

Binary search works by looking at the middle element of the portion of a sorted array still to be examined. But that's not the way we, people, look up words in a dictionary! Say we are searching for the word "ARRAY". Rather than opening the dictionary in the middle (like binary search would), we will open it towards the beginning because that's where words starting with "A" are found. Say we land on the word "ASSERTION". We would then repeat the process by flipping just a few pages towards the beginning of the dictionary rather than to the middle of what is remaining, because "AR" is close to "AS". We proceed this way until we reach the word "ARRAY". This promises to find words very fast.

In this exercise, we will tweak binary search as seen in class to implement this strategy. Say we are searching for an integer x in array segment $A[lo, hi)$: at each iteration, the next index to examine will be an educated guess where x should be based on the smallest and largest values seen so far and on x itself.

Here's some code — *most of it is identical to binary search seen in class*:

```
int search(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); @*/
{
  int max = int_max();
  int min = int_min();
  int lo = 0;
  int hi = n;

  while(lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  //@loop_invariant gt_seg(x, A, 0, lo);
  //@loop_invariant lt_seg(x, A, hi, n);
  //@loop_invariant min <= x && x <= max && min < max;
  {
    int guess = predict(x, lo, min, hi, max);
    //@assert lo <= guess && guess < hi;

    if (A[guess] == x) return guess;
    else if (A[guess] < x) {
      min = A[guess];
      lo = guess+1;
    } else {
      //@assert A[guess] > x;
      max = A[guess];
      hi = guess;
    }
  }
  return -1;
}
```

10pts **Task 1** Show that the loop invariant on line 21 is a valid. (You may not need all the lines.)

Initialization:

`max = int_max()`	by	line 12
`min = int_min()`	by	line 13
`int_min() <= x <= int_max()`	by	definition
`int_min() < int_max()`	by	definition
`min <= x <= max && min < max`	by	substitution

Preservation:

Assumption: `min <= x <= max && min < max`

To show: `min' <= x <= max' && min' < max'`

There are 3 cases to examine based on the value of `A[guess]`

Case `A[guess] == x`: Nothing to do as the function returns.

Case `A[guess] < x`:

`min' = A[guess]` by line 28

`max' = max` (unchanged in this branch)

`min' < x` by case assumption

`min' < x <= max'` by the previous line and the assumption

`min' <= x <= max' && min' < max'` by the properties of the ordering relations

Case `A[guess] > x`: Similar.

The function `predict(x, lo, min, hi, max)` returns an index between `lo` and `hi` that is as close to `lo` as `x` is close to `min`. Here is an implementation:

```
int predict(int x, int lo, int min, int hi, int max) {
  return (x - min) / (max - min) * (hi - lo) + lo;
}
```

4pts **Task 2** You should be suspicious! For one, `predict` does not have any contracts. Write pre- and post-conditions for this functions as to match its use in the function `search`.

```
//@requires 0 <= lo && lo < hi;
//@requires min <= x && x <= max && min < max;
//@ensures lo <= \result && \result < hi;
```

2pts **Task 3** Although the formula used in `predict` is mathematically correct, the return value may violate the `@assert` statement on line 24 of `search`. In fact it does precisely this on the very first iteration! Explain why this is the case.

The subexpression `(max - min)` may overflow, which will produce a negative result. This will always happen on the first iteration of the loop in `search`, because of the initialization of `min` and `max` on lines 12 and 13: `int_max() - int_min() = -1`, which violates the assertion on line 24 that `0 <= guess`.

5pts **Task 4** Modify the above code for `predict` so that it implements the same mathematical idea and yet the `@assert` statement on line 24 will never be violated for valid inputs. **IGNORE:** BROKEN TASK

```
int predict(int x, int lo, int min, int hi, int max) {
// ... contracts omitted ...

    return (x/2 - min/2) / (max/2 - min/2) * (hi - lo) + lo;

}
```

4pts **Task 5** What is the worst-case asymptotic complexity of `search` for an array of length n? Give a concrete 5-element array A and value x that witness this worst case.

Worst-case complexity: $O($ n $)$

Worst-case scenario:

A: [0, 0, 0, 0, 1]

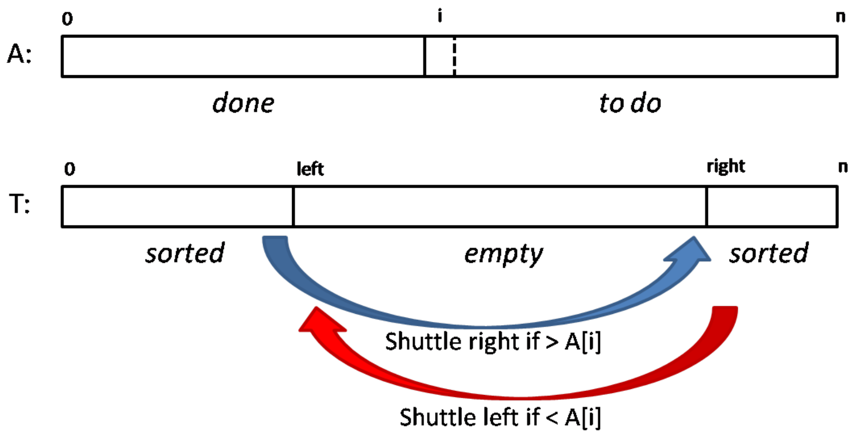
x: 1

3 Shuttle Sort (25 points)

The other day, Rob had an idea for a new sorting algorithm. He explained it to me as follows:

> To sort an array A, we allocate an auxiliary array T and maintain two indices *left* and *right* so that $T[0, left) \le T[right, n)$ and are sorted.
>
> We compare each element $A[i]$ with what's at $T[left - 1]$ and keep moving that element to the right side of T as long as it is larger than $A[i]$. We do the same thing with $T[right]$, moving smaller elements to the left side of T. When we're done with that, then we know that $T[0, left) \le A[i] \le T[right, n)$ and we put $A[i]$ in $T[left]$ and update *left*.
>
> Once all $A[i]$ have been placed in this way, we copy the contents of T back onto A.

He drew this picture on a napkin:



Rob called his algorithm "shuttle sort". He started implementing it but he ran out of time. The code he wrote is in Figure 1 on page 9. The comments in ALL CAPS are where the parts he didn't have time to work on need to go. He is pretty confident about the rest of his code.

3pts **Task 1** Give the missing invariants that relate variables `left`, `right`, `i` and `n` on lines 12 and 19.

```
12  //@loop_invariant  i == left + n - right ;

19  //@loop_invariant  i == left + n - right ;
```

2pts **Task 2** Prove that the loop starting at line 17 terminates.

By line 23, the quantity `left` is strictly decreasing in any given iteration. By line 17, it is bounded by 0.

```
void sort(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures is_sorted(A, 0, n);
{
  int[] T = alloc_array(int, n);
  int left = 0;  // next free position on the left
  int right = n; // 1 + next free position on the right

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant 0 <= left && left <= right && right <= n;
  // SOME LOOP INVARIANT ABOUT HOW left AND right RELATE TO i AND n
  //@loop_invariant is_sorted(T, 0, left) && is_sorted(T, right, n);
  //@loop_invariant le_segs(T, 0, left, T, right, n);
  {
    // Move bigger elements, if any, from left side to right side
    while (left > 0 && T[left-1] > A[i])
    //@loop_invariant 0 <= left && left <= right && right <= n;
    // SOME LOOP INVARIANT ABOUT HOW left AND right RELATE TO i AND n
    //@loop_invariant is_sorted(T, 0, left) && is_sorted(T, right, n);
    {
      swap(T, left-1, right-1);
      left--;
      right--;
    }
    //@assert ge_seg(A[i], T, 0, left);

    // Move smaller elements, if any, from right side to left side
    // WILL FINISH LATER: GOT TO GO FOR COFFEE WITH TOM
    //@assert ge_seg(A[i], T, 0, left);
    //@assert le_seg(A[i], T, right, n);

    // Place A[i] -- putting it in T[right-1] would work also
    T[left] = A[i];
    left++;
  }
  // SOME ASSERTION ABOUT HOW left AND right ARE RELATED AT THE END OF THE FOR LOOP
  //@assert is_sorted(T, 0, n);

  // Copy sorted elements back into A
  // ARGH!! LEFT THE STOVE ON; GOT TO RUN
}
```

Figure 1: Rob's Partial Implementation of Shuttle Sort^{TM}

8pts **Task 3** Write the missing code snippet that shuttles elements smaller than `A[i]` from the right side of T to the left side of T. This code starts at line 29 and may extend multiple lines. Include loop invariants. *(Hint: take inspiration to the loop starting at line 17, but additional invariants may apply since you now have more information.)*

```
  while (right < n && T[right] < A[i])
  //@loop_invariant 0 <= left && left <= right && right <= n;
  //@loop_invariant i == left + n - right;
  //@loop_invariant is_sorted(T, 0, left);
  //@loop_invariant is_sorted(T, right, n);
  //@loop_invariant ge_seg(A[i], T, 0, left);
  {
    swap(T, right, left);
    right++;
    left++;
  }
```

6pts **Task 4** Fill in the missing assertion about how `left` and `right` are related at the end of the main loop on line 37, and use that to justify the assertion on line 38 (you may not need all the lines provided).

`//@assert` `left == right` `;`

Proof that line 38 holds:

a.	`is_sorted(T, 0, left)`	by	line 13
b.	`is_sorted(T, right, n)`	by	line 13
c.	`le_segs(T, 0, left, T, right, n)`	by	line 14
d.	`left = right`	by	line 37
e.	`is_sorted(T, 0, n)`	by	(a-d)

3pts **Task 5** Write the missing code fragment that copies the contents of `T` back into `A`. This code starts at line 41 and may extend multiple lines. Include loop invariants on `A` and `i` as needed to prove the correctness of `sort`.

```
for (int i=0; i<n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant is_sorted(A, 0, i);
  {
    A[i] = T[i];
  }
```

3pts **Task 6** Was Rob's (partial) effort worthwhile? What is the worst-case asymptotic complexity of `sort` for an array of length n? What does an array need to look like for this to happen?

Worst-case complexity: $O(\underline{\quad n^2 \quad})$

When alternating elements in array grow bigger and smaller.

For example [0, 1, -1, 2, -2, 3, -3].

4 Max-Stacks (25 points)

A *max-stack* is a data structure that, in addition to the standard stack operations, also supports the operation of getting the maximum value in the stack in constant time.

An implementation of a max-stack can achieve this by using *two* regular stacks, a stack D for the data elements and a stack M to keep track of the maxima. For each element e in M, e is the largest element in the data stack D up to e starting from the bottom. The maxima stack M may contain duplicate values. For example, suppose we push the values 42, 16, 29, 53, 9, 70 and 38 onto a max-stack in the order shown, then this max-stack implementation will consist of the following two stacks (the top is on the right):

(Data) D: $\boxed{42,\ 16,\ 29,\ 53,\ 9,\ 70,\ 38}$ $\rightleftarrows$

(Maxima) M: $\boxed{42,\ 53,\ 70}$ $\rightleftarrows$

Here is an interface for regular stacks (of **int**s), defined as the type `stack_t`:

```
// typedef ______* stack_t;

bool stack_empty(stack_t S)          /* O(1) */
  /*@requires S != NULL; @*/;

stack_t stack_new()                  /* O(1) */
  /*@ensures \result != NULL; @*/
  /*@ensures stack_empty(\result); @*/;

void push(stack_t S, int x)          /* O(1) */
  /*@requires S != NULL; @*/;

int pop(stack_t S)                   /* O(1) */
  /*@requires S != NULL; @*/
  /*@requires !stack_empty(S); @*/;

int peek(stack_t S)                  /* O(1) */
 /*@requires S != NULL; @*/
 /*@requires !stack_empty(S); @*/;
```

Note the inclusion of a `peek` function in this stack interface.

You will manipulate the stack data structure using the type `stack_t` and its interface to build a new data structure for a max-stack as described above. You do not know how the stack data structure is implemented. You are only given the interface above.

You may assume that the specification function `is_maxstack` is already written and tests if the data structure invariant for a max-stack holds.

2pts **Task 1** Complete the data type definition for max-stacks below:

```
typedef struct maxstack_header maxstack;

struct maxstack_header {

    __________stack_t__________ data;

    __________stack_t__________ maxima;
};
```

2pts **Task 2** Write a *constant-time* function `maxstack_empty` that returns `true` if it is empty or `false` otherwise.

```
bool maxstack_empty(maxstack* MS)              /* O(1) */
//@requires is_maxstack(MS);
{
    return stack_empty(MS->data);
    // OR
    // return stack_empty(MS->maxima);
}
```

4pts **Task 3** Write a *constant-time* function `maxstack_new` that returns a new `maxstack_t` that is empty.

```
maxstack* maxstack_new()                       /* O(1) */
//@ensures is_maxstack(\result);
//@ensures maxstack_empty(\result);
{
    maxstack* MS = alloc(maxstack);
    MS->data = stack_new();
    MS->maxima = stack_new();
    return MS;
}
```

2pts **Task 4** Write a *constant-time* function maxstack_getmax that returns, but does not remove, the maximum in a non-empty max-stack.

```
int maxstack_getmax(maxstack* MS)            /* O(1) */
//@requires is_maxstack(MS);
//@requires !maxstack_empty(MS);
{

    return peek(MS->maxima);
}
```

6pts **Task 5** Write a *constant-time* function maxstack_push for max-stacks as described above.

```
void maxstack_push(maxstack* MS, int n)    /* O(1) */
//@requires is_maxstack(MS);
//@ensures is_maxstack(MS);
{
    push(MS->data, n)                                            ;

    if (      stack_empty(MS->maxima) || n >= peek(MS->maxima)       )

        push(MS->maxima, n)                                      ;
}
```

5pts **Task 6** Write a *constant-time* function maxstack_pop for the max-stack described above. This returns the top data element of the max-stack.

```
int maxstack_pop(maxstack* MS)               /* O(1) */
//@requires is_maxstack(MS);
//@requires !maxstack_empty(MS);
//@ensures is_maxstack(MS);
{
    int n = pop(MS->data)                                         ;

    if (                    n == peek(MS->maxima)                     )

       n = pop(MS->maxima)                                         ;

    return n;
}
```

4pts **Task 7** Here is the interface of the max-stack library you have just developed.

```
// typedef _________ *maxstack_t;

bool maxstack_empty(maxstack_t MS)             /* O(1) */
/*@requires MS != NULL;@*/ ;

maxstack_t maxstack_new()                      /* O(1) */
/*@ensures \result != NULL;@*/ ;
/*@ensures maxstack_empty(\result);@*/ ;

int maxstack_getmax(maxstack_t MS)             /* O(1) */
/*@requires MS != NULL;@*/
/*@requires !maxstack_empty(MS);@*/ ;

void maxstack_push(maxstack_t MS, int n)       /* O(1) */
/*@requires MS != NULL;@*/ ;

int maxstack_pop(maxstack_t MS)                /* O(1) */
/*@requires MS != NULL;@*/
/*@requires !maxstack_empty(MS);@*/ ;
```

Explain why the function `is_maxstack` is not used in the contracts above.

Because it refers to the internal details of the implementation,

which are not visible to the client.

5 Min-Stacks *(makeup)* (25 points)

A *min-stack* is a data structure that, in addition to the standard stack operations, also supports the operation of getting the minimum value in the stack in constant time.

An implementation of a min-stack can achieve this by using *two* regular stacks, a stack D for the data elements and a stack M to keep track of the minima. For each element e in M, e is the smallest element in the data stack D up to e starting from the bottom. The minima stack M may contain duplicate values. For example, suppose we push the values 42, 16, 29, 53, 9, 70 and 38 onto a min-stack in the order shown, then this min-stack implementation will consist of the following two stacks (the top is on the right):

$$\text{(Data)} \quad D: \boxed{42,\ 16,\ 29,\ 53,\ 9,\ 70,\ 38} \rightleftarrows$$

$$\text{(Minima)} \quad M: \boxed{42,\ 16,\ 9} \rightleftarrows$$

Here is an interface for regular stacks (of **int**s), defined as the type `stack_t`:

```
// typedef ______* stack_t;

bool stack_empty(stack_t S)           /* O(1) */
  /*@requires S != NULL; @*/;

stack_t stack_new()                   /* O(1) */
  /*@ensures \result != NULL; @*/
  /*@ensures stack_empty(\result); @*/;

void push(stack_t S, int x)           /* O(1) */
  /*@requires S != NULL; @*/;

int pop(stack_t S)                    /* O(1) */
  /*@requires S != NULL; @*/
  /*@requires !stack_empty(S); @*/;

int peek(stack_t S)                   /* O(1) */
 /*@requires S != NULL; @*/
 /*@requires !stack_empty(S); @*/;
```

Note the inclusion of a `peek` function in this stack interface.

You will manipulate the stack data structure using the type `stack_t` and its interface to build a new data structure for a min-stack as described above. You do not know how the stack data structure is implemented. You are only given the interface above.

You may assume that the specification function `is_minstack` is already written and tests if the data structure invariant for a min-stack holds.

2pts **Task 1** Complete the data type definition for min-stacks below:

```
typedef struct minstack_header minstack;

struct minstack_header {

    __________stack_t__________ data;

    __________stack_t__________ minima;
};
```

2pts **Task 2** Write a *constant-time* function `minstack_empty` that returns `true` if it is empty or `false` otherwise.

```
bool minstack_empty(minstack* MS)           /* O(1) */
//@requires is_minstack(MS);
{
    return stack_empty(MS->data);
    // OR
    // return stack_empty(MS->minima);
}
```

4pts **Task 3** Write a *constant-time* function `minstack_new` that returns a new `minstack_t` that is empty.

```
minstack* minstack_new()                    /* O(1) */
//@ensures is_minstack(\result);
//@ensures minstack_empty(\result);
{
    minstack* MS = alloc(minstack);
    MS->data = stack_new();
    MS->minima = stack_new();
    return MS;
}
```

2pts **Task 4** Write a *constant-time* function `minstack_getmin` that returns, but does not remove, the minimum in a non-empty min-stack.

```
int minstack_getmin(minstack* MS)          /* O(1) */
//@requires is_minstack(MS);
//@requires !minstack_empty(MS);
{
    return peek(MS->minima);
}
```

6pts **Task 5** Write a *constant-time* function `minstack_push` for min-stacks as described above.

```
void minstack_push(minstack* MS, int n)    /* O(1) */
//@requires is_minstack(MS);
//@ensures is_minstack(MS);
{
    push(MS->data, n)                                      ;

    if (      stack_empty(MS->minima) || n <= peek(MS->minima)       )

        push(MS->minima, n)                                ;
}
```

5pts **Task 6** Write a *constant-time* function `minstack_pop` for the min-stack described above. This returns the top data element of the min-stack.

```
int minstack_pop(minstack* MS)             /* O(1) */
//@requires is_minstack(MS);
//@requires !minstack_empty(MS);
//@ensures is_minstack(MS);
{
    int n = pop(MS->data)                                  ;

    if (                    n == peek(MS->minima)                    )

       n = pop(MS->minima)                                 ;

    return n;
}
```

4pts **Task 7** Here is the interface of the min-stack library you have just developed.

```
// typedef _________ *minstack_t;

bool minstack_empty(minstack_t MS)            /* O(1) */
/*@requires MS != NULL;@*/ ;

minstack_t minstack_new()                     /* O(1) */
/*@ensures \result != NULL;@*/ ;
/*@ensures minstack_empty(\result);@*/ ;

int minstack_getmin(minstack_t MS)            /* O(1) */
/*@requires MS != NULL;@*/
/*@requires !minstack_empty(MS);@*/ ;

void minstack_push(minstack_t MS, int n)      /* O(1) */
/*@requires MS != NULL;@*/ ;

int minstack_pop(minstack_t MS)               /* O(1) */
/*@requires MS != NULL;@*/
/*@requires !minstack_empty(MS);@*/ ;
```

Explain why the function `is_minstack` is not used in the contracts above.

Because it refers to the internal details of the implementation,

which are not visible to the client.

Selected library functions

```
int int_max();

int int_min();

/* POW: calculates a ** b */
int POW(int a, int b)
  /*@requires 0 <= b; @*/ ;

/* is_in: x in A[lo,hi) */
bool is_in(int x, int[] A, int lo, int hi)
  /*@requires 0 <= lo  && lo <= hi && hi <= \length(A); @*/ ;

/* is_sorted: A[lo..hi) SORTED */
bool is_sorted(int[] A, int lo, int hi)
  /*@requires 0 <= lo  && lo <= hi && hi <= \length(A); @*/ ;

/* lt_seg: x < A[lo..hi) */
bool lt_seg(int x, int[] A, int lo, int hi)
  /*@requires 0 <= lo && lo <= hi && hi <= \length(A); @*/ ;

/* le_seg: x <= A[lo..hi) */
bool le_seg(int x, int[] A, int lo, int hi)
  /*@requires 0 <= lo && lo <= hi && hi <= \length(A); @*/ ;

/* gt_seg: x > A[lo..hi) */
bool gt_seg(int x, int[] A, int lo, int hi)
  /*@requires 0 <= lo && lo <= hi && hi <= \length(A); @*/ ;

/* ge_seg: x >= A[lo..hi) */
bool ge_seg(int x, int[] A, int lo, int hi)
  /*@requires 0 <= lo && lo <= hi && hi <= \length(A); @*/ ;

/* ge_segs: A[lo1,hi1) >= B[lo2,hi2) */
bool ge_segs(int[] A, int lo1, int hi1, int[] B, int lo2, int hi2)
  /*@requires 0 <= lo1 && lo1 <= hi1 && hi1 <= \length(A); @*/
  /*@requires 0 <= lo2 && lo2 <= hi2 && hi2 <= \length(B); @*/ ;

/* le_segs: A[lo1,hi1) <= B[lo2,hi2) */
bool le_segs(int[] A, int lo1, int hi1, int[] B, int lo2, int hi2)
  /*@requires 0 <= lo1 && lo1 <= hi1 && hi1 <= \length(A); @*/
  /*@requires 0 <= lo2 && lo2 <= hi2 && hi2 <= \length(B); @*/ ;
```