15-122 : Principles of Imperative Computation, Spring 2016

Written Homework 4

Due: Monday 8th February, 2016

Name: ________________________________

Andrew ID: ________________________________

Section: ________________________________

This written homework covers big-O notation and some reasoning about searching and sorting algorithms. You will use some of the functions from the arrayutil.c0 library that was discussed in lecture in this assignment.

Print out this PDF double-sided, staple pages in order,
and write your answers *neatly* by hand in the spaces provided.

The assignment is due by 5:30pm on Monday 8th February, 2016.

You can hand in the assignment during lab
or in the box outside of the lab room.

You must hand in your homework yourself;
do not give it to someone else to hand in.

1. Another Sort

Consider the following function that sorts the integers in an array, using `swap` and `is_sorted` from `arrayutil.c0`.

```
 1 void sort(int[] A, int n)
 2 //@requires 0 <= n && n <= \length(A);
 3 //@ensures is_sorted(A, 0, n);
 4 {
 5   for (int i = 0; i < n; i++)
 6   //@loop_invariant 0 <= i && i <= n;
 7   //@loop_invariant le_segs(A, 0, n-i, A, n-i, n);
 8   //@loop_invariant is_sorted(A, __________, __________);
 9   {
10     int s = 0;
11     for (int j = 0; j < n-i-1; j++)
12     //@loop_invariant 0 <= j && j <= n-i-1;
13     //@loop_invariant ge_seg(A[j], A, 0, j);
14     //@loop_invariant s > 0 || (s == 0 && is_sorted(A, 0, j));
15     {
16       if (A[j] > A[j+1]) {
17         swap(A, j, j+1); // function that swaps A[j] and A[j+1]
18         s = s + 1;
19       }
20     }
21     if (s == 0) return;
22   }
23 }
```

1pt (a) Complete the missing loop invariant on line 8.

```
8 //@loop_invariant is_sorted(A, _____________, _____________);
```

1pt (b) The asymptotic complexity of this sort depends on the number of comparisons made between pairs of array elements. Let $T(n)$ be the worst-case number of such comparisons made when `sort(A, n)` is called. Give a *closed form* expression for $T(n)$. Then express $T(n)$ in big O notation in its simplest, tightest form.

$T(n) = $ ______________________________

$T(n) \in O($______________________________$)$

1pt (c) Using big-O notation, what is the **best** case asymptotic complexity of this sort as a function of n? Under what condition does the best case occur?

The best case asymptotic complexity of this sort is $O($____________$)$. This occurs when

2. **Big O notation**

Recall the definition for big O:

$g \in O(f)$ if there is a constant $c > 0$ and some n_0 such that for every $n \geq n_0$ we have $g(n) \leq c \cdot f(n)$.

2pts (a) Demonstrate graphically that $3n^2 + 7n + 4 \in O(n^2)$ by finding values for c and n_0 that satisfy the definition above, where $g(n) = 3n^2 + 7n + 4$ and $f(n) = n^2$. Draw a picture to illustrate that cn^2 acts as an upper bound to $3n^2 + 7n + 4$ for all $n_0 \geq n$ using your values for c and n_0. Be sure to label the horizontal and vertical axes.

$c =$ ____________________

$n_0 =$ ____________________

1pt (b) In one sentence, explain why $3n^2 + 7n + 4 \notin O(n)$.

1pt (c) Determine the asymptotic complexity of the following function using big O notation as a function of a and b in its simplest, tightest form.

```
int mystery(int a, int b)
//@requires a > 0 && b > 0;
{
   int x = 0;
   for (int i = 1; i < 10; i++) {
      int j = a;

      while (j > 0) {
         int k = b;

         while (k > 0) {
            x = x + i * j * k;
            k = k / 2;
         }

         j--;
      }
   }

   return x;
}
```

$O($ ______________________________ $)$

Reminder: as discussed in recitation, in this class when we are dealing with logarithms, we consider the "simplest form" to be the one without the base. Therefore, we prefer $O(\log n)$ over big-O descriptions like $O(\log_4 n)$ or $O(\ln n)$.

3. **Binary Search**

   Consider the `search` function we were analyzing from the previous written assignment. This function returned the index of the first occurrence of x in the array A, or -1 if x is not found. Now let's fill in the loop body with code that implements the binary search algorithm (on a sorted array, of course).

```
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
         && (\result == 0 || A[\result-1] < x)); @*/
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo;
  //@loop_invariant lo <= hi;
  //@loop_invariant hi <= n;
  //@loop_invariant gt_seg(x, A, 0, lo);
  //@loop_invariant le_seg(x, A, hi, n);
  {
    if (A[lo] == x) return lo;
    int mid = lo + (hi-lo)/2;
    if (A[mid] < x) lo = mid+1;
    else { /*@assert(A[mid] >= x); @*/
      hi = mid;
    }
  }
  //@assert lo == hi;
  if(lo != n && A[lo] == x) return lo;
  return -1;
}
```

1pt

(a) Argue that the loop has to terminate. Follow the format for termination arguments described in the first written homework!

2pts (b) Prove that, in the case that the code returns on line 18, the postcondition on lines 4–7 always evaluates to true. We've given the starting facts you'll use in this proof.

When we start an iteration of the loop, we know the following:

- `0 <= n && n <= \length(A)` by the function's precondition (line 2, `A` and `n` are never modified by the function)
- `A[0,n)` SORTED by the function's precondition (line 3, `A` and `n` are never modified by the function and `A` is not written to anywhere)
- `lo < hi` by the loop guard (line 11)
- `0 <= lo && lo <= hi && hi <= n` by the first three loop invariants (lines 12–14)
- `lo == 0 || x > A[lo-1]` by the fourth loop invariant (line 15)
- `hi == n || x <= A[hi]` by the fifth loop invariant (line 16)

5pts

(c) Modify the search function above so that it uses the binary search algorithm to return the index of the *last* occurrence of x in array A (as opposed to the first occurrence) or -1 if x is not found. Think carefully about the contracts to make sure that your array accesses are safe and that the function is logically correct!

```
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 lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo;
  //@loop_invariant lo <= hi;
  //@loop_invariant hi <= n;

  //@loop_invariant ________________________________;

  //@loop_invariant ________________________________;
  {
    int mid = lo + (hi-lo)/2;

    if (________________________________) lo = mid+1;

    else { /*@assert(__________________________); @*/
      hi = mid;
    }
  }
  //@assert lo == hi;

  if (__________________________________________) {

     return ________________________________;
  }

  return -1;
}
```