15-122: Principles of Imperative Computation, Spring 2020

Written Homework 4

Due on Gradescope: Monday 10th February, 2020 by 9pm

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.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***pdfescape*** or ***dochub***, two web-based PDF editors that work from anywhere.
- ***Preview***, the Mac's PDF viewer.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though.*

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	3	Total
Points:	3.5	3.5	5	12
Score:				

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 c = 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 c > 0 || (c == 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         c = c + 1;
19       }
20     }
21     if (c == 0) return;
22   }
23 }
```

0.5pts **1.1** Complete the missing loop invariant on line 8.

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

1pt **1.2** The asymptotic complexity of this function 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 $T(n)$ — a simple, non-recursive mathematical expression that doesn't use $\sum$ or similar. Then express $T(n)$ in big-O notation in its simplest, tightest form.

$T(n) \;=$ ______________________________

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

1pt **1.3** You will now justify your last answer. Call $f(n)$ the function you wrote in the second blank of the previous question. Show that $T(n) \in O(f(n))$ using the formal definition of big-O. That is, find a $c > 0$ and $n_0 \geq 0$ such that for every $n \geq n_0$, $T(n) \leq cf(n)$. Show your work.

1pt **1.4** 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:

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

2pts **2.1** Demonstrate graphically that $2n^2 + 5n + 1 \in O(n^2)$ by finding values for c and n_0 that satisfy the definition above, where $f(n) = 2n^2 + 5n + 1$ and $g(n) = n^2$. Draw a picture to illustrate that cn^2 acts as an upper bound to $2n^2 + 5n + 1$ for all $n \geq n_0$ using your values for c and n_0. Be sure to label the horizontal and vertical axes.

If you choose to insert a plot (or a picture) generated using an external application, please **paste it in the space provided** — do not insert it as a new page.

$c =$ ________________________________

$n_0 =$ ________________________________

0.5pts

2.2 In one sentence, explain why $2n^2 + 5n + 1 \notin O(n)$.

1pt

2.3 Determine the asymptotic complexity of the following function using big-O notation as a function of its arguments in its simplest, tightest form.

```
int mystery(int u, int v)
//@requires u > 0 && v > 0;
{
    int y = 0;
    for (int i = 1; i < 10; i++) {
        int j = u;

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

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

            j--;
        }
    }

    return y;
}
```

$O(\underline{\hspace{6cm}})$

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 analyzed in the last 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).

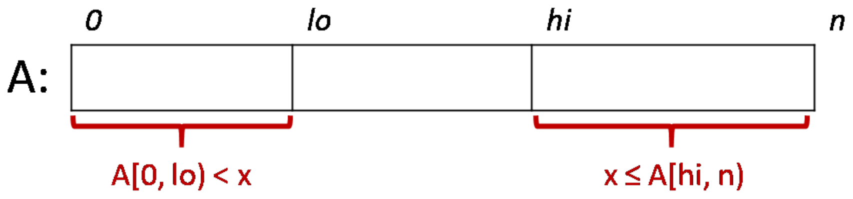
```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
        || (0 <= \result && \result < n
            && A[\result] == x
            && (\result == 0 ||
                A[\result-1] < x)); @*/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

   Here is a graphical representation of the loop invariants of this function:



1.5pts

3.1 Prove that, in the case that the code returns on line 22, the postcondition on lines 4–8 always evaluates to true. We've given the starting facts you'll use in this proof. Justify your answers using line numbers or previous steps.

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

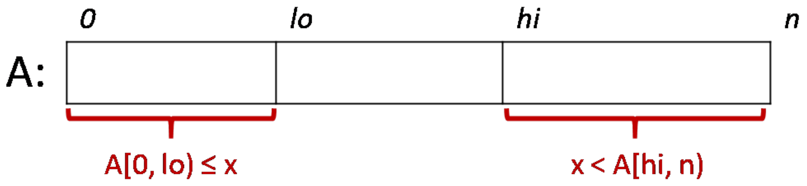
A.	`0 <= n && n <= \length(A)`	by line 2 (precondition 1)
B.	`A[0,n)` SORTED	by line 3 (precondition 2)
C.	`lo < hi`	by line 14 (loop guard)
D.	`0 <= lo && lo <= hi && hi <= n`	by lines 15–17 (loop invariants 1–3)
E.	`lo == 0 || x > A[lo-1]`	by line 18 (loop invariant 4)
F.	`hi == n || x <= A[hi]`	by line 19 (loop invariant 5)

1pt

3.2 Argue that the loop has to terminate. Follow the format for termination arguments described in class and in prior homework!

2.5pts

3.3 On the next page, 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. Here's a picture of the loop invariants you are aiming for, and that you should express in C0 on lines 18 and 20. Study them carefully.



There are operationally correct implementations for which it is not possible to prove that the postconditions are true. Such implementations will not get full credit.

```
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;
}
```