15-122: Principles of Imperative Computation, Spring 2023

Written Homework 4

Due on Gradescope: Sunday 5th February, 2023 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.
- ***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*.

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

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:	4	5.5	5.5	15
Score:				

1. **Another Sort**

   Consider the following function that sorts the integers in an array, using `swap` and `is_sorted` from `arrayutil.c0` which you can find on the course web page.

```
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 }
```

1pt **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 can be computed by counting the **number of comparisons between pairs of array elements** it makes. 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** 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 _____.

1pt **1.4** Is the sorting function on the previous page in-place? Is it stable? Justify your answers in one sentence.

In-place? ◯ Yes ◯ No, because _____

_____

Stable? ◯ Yes ◯ No, because _____

_____

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 $f(n) \leq c \cdot g(n)$.

2.5pts **2.1** To show that $f(n) \in O(g(n))$, you need to find a specific c and an n_0, assume that $n \geq n_0$, and then derive $f(n) \leq cg(n)$ for such values of n. Let's practice (follow the format for big-O proofs used in class):

a. Show that $f(n) = 3n + 7 \in O(n)$.

$c =$ _____, $n_0 =$ _____

To show: _____ (expand f, g, c and n_0)

A. $n \geq$ _____ by assumption

B. _____ by _____

C. _____ by _____

D. _____ by _____

E. _____ by _____

b. Show that $f(n) = 3n^2 + 7n + 4 \in O(n^2)$.

$c = 14.0$ (given to you), $n_0 =$ _____

To show: _____ (expand f, g, c and n_0)

A. _____ by _____

B. _____ by _____

C. _____ by _____

D. _____ by _____

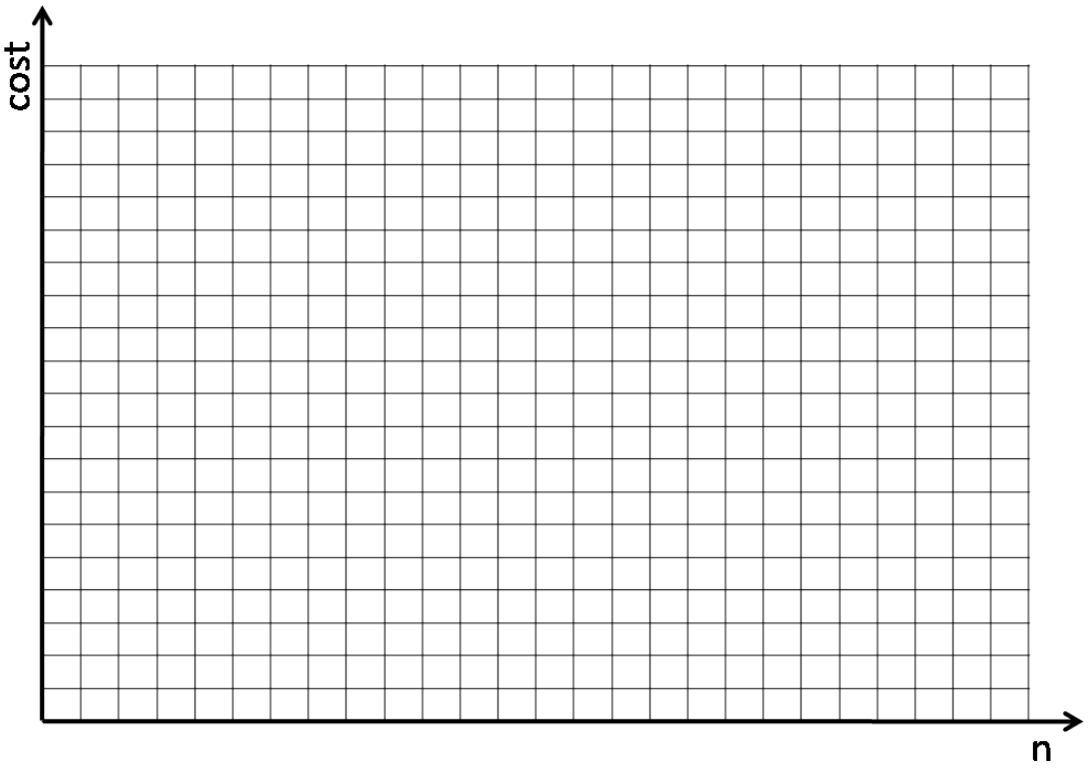
E. _____ by _____

F. _____ by _____

2pts **2.2** 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 $f(n) = 3n^2 + 7n + 4$ and $g(n) = n^2$. Draw a picture to illustrate that cn^2 acts as an upper bound to $3n^2 + 7n + 4$ for all $n \geq n_0$ using your values for c and n_0. Be sure to label the functions and make their intersection is prominently visible.

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

$c =$ _____, $n_0 =$ _____



1pt

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

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

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

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

            j--;
        }
    }

    return u;
}
```

$O($_____$)$

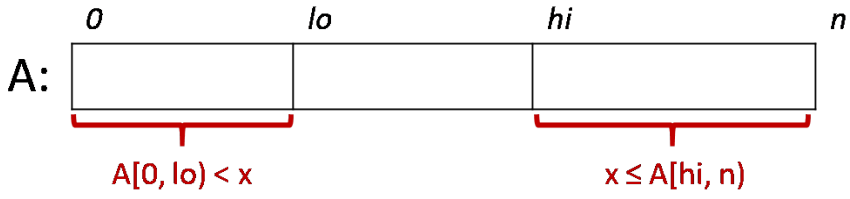
Reminder: as discussed in recitation, in this class when we are dealing with logarithms, we consider the "simplest form" to be the one *without the base*. Therefore, we write $O(\log n)$ over big-O descriptions like $O(\log_4 n)$ or $O(\ln n)$.

3. **Binary Search**

Consider a streamlined version of 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).

```
42 int search(int x, int[] A, int n)
43 //@requires 0 <= n && n <= \length(A);
44 //@requires is_sorted(A, 0, n);
45 /*@ensures (\result == -1 && !is_in(x, A, 0, n))
46         || (0 <= \result && \result < n
47             && A[\result] == x
48             && gt_seg(x, A, 0, \result)); @*/
49 {
50   int lo = 0;
51   int hi = n;
52
53   while (lo < hi)
54   //@loop_invariant 0 <= lo;
55   //@loop_invariant lo <= hi;
56   //@loop_invariant hi <= n;
57   //@loop_invariant gt_seg(x, A, 0, lo);
58   //@loop_invariant le_seg(x, A, hi, n);
59   {
60     if (A[lo] == x)
61       return lo;
62     int mid = lo + (hi-lo)/2;
63     if (A[mid] < x)
64       lo = mid+1;
65     else { /*@assert(A[mid] >= x); @*/
66       hi = mid;
67     }
68   }
69   //@assert lo == hi;
70
71   if (lo != n && A[lo] == x) return lo;
72   return -1;
73 }
```

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



2pts

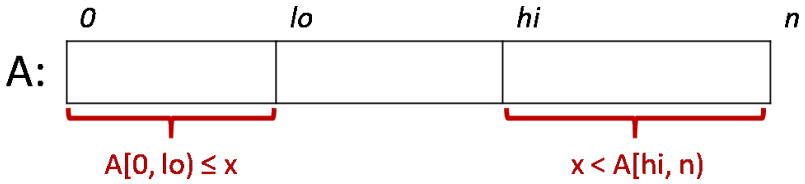
3.1 Prove that, in the case that the code returns on line 61, the postcondition on lines 45–48 always evaluates to true. We've given the starting facts you'll use in this proof (you may not need all of them). Justify your answers using line numbers or previous steps. *(You may not need all the lines provided.)*

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

A.	`0 <= n && n <= \length(A)`	by line 43 (precondition 1)
B.	`A[0,n)` SORTED	by line 44 (precondition 2)
C.	`lo < hi`	by line 53 (loop guard)
D.	`0 <= lo && lo <= hi && hi <= n`	by lines 54–56 (LIs 1–3)
E.	`gt_seg(x, A, 0, lo)`	by line 57 (LI 4)
F.	`le_seg(x, A, hi, n)`	by line 58 (LI 5)
G.		
H.		
I.		
J.		
K.		
L.		
M.		

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 56 and 58. Study it 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.

```
39 int search(int x, int[] A, int n)
40 //@requires 0 <= n && n <= \length(A);
41 //@requires is_sorted(A, 0, n);
42 /*@ensures (\result == -1 && !is_in(x, A, 0, n))
43         || (0 <= \result && \result < n
44             && A[\result] == x
46             && (_____)); @*/
47 {
48   int lo = 0;
49   int hi = n;
51   while (lo < hi)
52   //@loop_invariant 0 <= lo;
53   //@loop_invariant lo <= hi;
54   //@loop_invariant hi <= n;
56   //@loop_invariant _____;
58   //@loop_invariant _____;
59   {
60     int mid = lo + (hi-lo)/2;
62     if (_____) lo = mid+1;
64     else { /*@assert(_____); @*/
65       hi = mid;
66     }
67   }
68   //@assert lo == hi;
70   if (_____) {
72       return _____;
73   }
75   return -1;
76 }
```