15-122 : Principles of Imperative Computation, Spring 2014

Written Homework 4

Due before class: Thursday, February 13, 2014

Name: _____

Andrew ID: _____

Recitation: _____

In this homework assignment, we will examine asymptotic complexity, searching and sorting.

You will use some of the functions from the `arrayutil.c0` library that are discussed in this assignment. You'll find this file in `theory4.tgz`.

Question	Points	Score
1	6	
2	2	
3	7	
Total:	15	

You must do this assignment in one of two ways and bring the stapled printout to the handin box on Thursday:

1) Write your answers *neatly* on a printout of this PDF.

2) Use the TeX template at `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/theory4.tgz`

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

```
void sort(int[] A, int n)
  //@requires 0 <= n && n <= \length(A);
  //@ensures is_sorted(A, 0, n);
  {
    for (int i = 0; i < n; i++)
      //@loop_invariant 0 <= i && i <= n;
      //@loop_invariant le_segs(A, 0, n-i, A, n-i, n);
      //@loop_invariant is_sorted(A, ____, _____);
      {
        int s = 0;
        for (int j = 0; j < n-i-1; j++)
          //@loop_invariant 0 <= j && j <= n-i-1;
          //@loop_invariant ge_seg(A[j], A, 0, j);
          //@loop_invariant s > 0 || (s == 0 && is_sorted(A, 0, j));
          {
            if (A[j] > A[j+1]) {
                swap(A, j, j+1);  // function that swaps A[j] and A[j+1]
                s = s + 1;
            }
          }
        if (s == 0) return;
      }
  }
```

(1) (a) Complete the missing loop invariant for the first (`i`) loop.

Solution:

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

(1) (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 $T(n)$.

Solution:

(1) (c) Using big-O notation, what is asymptotic complexity of $T(n)$? This is the worst-case runtime complexity of `sort`.

Solution:

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

(2) (d) Using your answer from the previous part, 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.

Solution:

(1) (e) Using big-O notation, what is the best case asymptotic complexity of this sort as a function of n.

Solution:

$O(\quad\quad)$

2. **Timing Code**

The following run times were obtained when using two different algorithms on a data set of size n. You are asked to determine asymptotic complexity of the algorithms based on this time data. Determine the asymptotic complexity of each algorithm as a function of n. Use big-O notation in its tightest form and briefly explain how you reached the conclusion.

(1) (a)

```
n              Execution Time
1000           0.564 milliseconds
2000           2.271 milliseconds
4000           8.992 milliseconds
8000           36.150 milliseconds
```

Solution: $O(\quad)$

(1) (b)

```
n              Execution Time
1000           0.043 milliseconds
1000000        43.68 milliseconds
1000000000     43.9 seconds
```

Solution: $O(\quad)$

3. **Computing Overlaps**

In this problem, we will study the Overlap Problem, which is the task of computing the number of shared elements between two arrays. The problem requires an array `A[]` of m integers and a second array `B[]` of n integers. We require the integers of `A[]` and `B[]` to be *distinct*, meaning no integer will occur more than once in `A[]` (or in `B[]`), though some integers may occur once in each of `A[]` and `B[]`.

Assume `linsearch(x, A, i, j)` returns the index of the first occurrence of integer `x` in integer array `A[i,j)` or -1 if `x` is not found. This function does not require that the elements be in sorted order.

Consider the following function which counts the number of integers which are in both of `A[]` and `B[]`.

```
/*  1 */   int overlap(int[] A, int m, int[] B, int n)
/*  2 */   //@requires 0 <= m && m <= \length(A);
/*  3 */   //@requires 0 <= n && n <= \length(B);
/*  4 */   // placeholder for future requirement (part b)
/*  5 */   // placeholder for future requirement (part d)
/*  6 */   {
/*  7 */       int count = 0;
/*  8 */       int i = 0;
/*  9 */       while (i < m)
/* 10 */       //@loop_invariant  0 <= i;
/* 11 */       {
/* 12 */           if (linsearch(A[i], B, 0, n) != -1) {
/* 13 */               count = count + 1;
/* 14 */           }
/* 15 */           i = i + 1;
/* 16 */       }
/* 17 */       return count;
/* 18 */   }
```

(1) (a) Uisng big-O notation, what is the worst-case runtime of this algorithm? Your answer should be in terms of m and n.

Solution:

$O($ $)$

(2) (b) Suppose we add a third precondition to the function on line 4 as follows:

```
/*  4 */   //@requires is_sorted(B, 0, n);
```

Using this additional requirement, explain how to modify the function to solve the Overlap Problem asymptotically faster than it currently does. (State which line(s) change and what the change(s) should be.)

Solution:

(1) (c) Using big-O notation, what is the worst-case runtime complexity of your revised algorithm? Your answer should be in terms of m and n.

Solution: $O($ $)$

(3) (d) Suppose we add a fourth precondition to the function on line 5 as follows:

```
/*  5 */   //@requires is_sorted(A, 0, m);
```

So now we require that both arrays are sorted prior to execution of this function. Describe an $O(m + n) = O(max(m, n))$ algorithm to solve the Overlap Problem. You should NOT write code; simply explain how the new algorithm works in a clear, concise manner. Justify in one sentence that the runtime complexity is $O(m + n)$. (HINT: The algorithm should be very similar to something you've seen in class already.)

Solution: