CSE 21 Fall 2018
Homework 1
Due: Sunday, October 7, 2018 at 10:00pm

Instructions

Homework should be done in groups of **one to three** people. You are free to change group members at any time throughout the quarter. Problems should be solved together, not divided up between partners. Homework must be submitted through **Gradescope** by **a single representative**. Submissions must be received by **10:00pm** on the due date, and there are no exceptions to this rule.

You will be able to look at your scanned work before submitting it. Please ensure that your submission is **legible** (neatly written and not too faint) or your homework may not be graded.

Students should consult their textbook, class notes, lecture slides, instructors, TAs, and tutors when they need help with homework. Students should not look for answers to homework problems in other texts or sources, including the Internet. You may ask questions about the homework in office hours, but **not on Piazza**.

Your assignments in this class will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should **always explain** how you arrived at your conclusions and **justify your answers** with mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to **convince the reader** that your results and methods are sound.

For questions that require pseudocode, you can follow the same format as the textbook, or you can write pseudocode in your own style, as long as you specify what your notation means. For example, are you using "=" to mean assignment or to check equality? You are welcome to use any algorithm from class as a subroutine in your pseudocode. For example, if you want to sort list A using InsertionSort, you can call InsertionSort(A) instead of writing out the pseudocode for InsertionSort.

Required Reading Rosen Sections 3.1 and 5.5.

Key Concepts Sorting algorithms, including selection (min) sort, insertion sort, and bubble sort; loop invariants and correctness proofs; searching algorithms; counting comparisons; best and worst case.

Note: For this assignment, the word "comparison" refers only to comparisons involving list elements. For example, if a_i and a_j are list elements in a list of length n, the code **if** $a_i < a_j$ performs one comparison. Similarly, the code **if** $a_i < 5$ performs one comparison. However, we would say the code **if** $i < n$ performs no comparisons because it is not making a comparison involving a list element.

---

1. Consider the following algorithms for Bubble Sort and its revision.

   procedure *BubbleSort* ($A = a_1, \ldots, a_n$: list of integers)

   ```
   1.  for i := 1 to n − 1
   2.      for i := 1 to n − 1
   3.          if a_j > a_{j+1} then
   4.              Interchange a_j and a_{j+1}
   ```

   procedure *RevisedBubbleSort* ($A = a_1, \ldots, a_n$: list of integers)

   ```
   1.  for i := 1 to n − 1
   2.      done := true
   3.      for j := 1 to n − i
   4.          if a_j > a_{j+1} then
   5.              Interchange a_j and a_{j+1}
   6.              done := false
   7.      if (done = true) then break
   ```

   Here, the **break** command on line 7 of *RevisedBubbleSort* will terminate the execution of the loop in which it occurs.

   (a) (2 points) How many comparisons does BubbleSort make on an array of size n that is already sorted from smallest to largest?

   Solution: On the ith iteration, BubbleSort makes i comparisons. In total, the algorithm does $1 + 2 + \cdots + (n-1) = \dfrac{(n-1)n}{2}$ comparisons.

   (b) (2 points) How many comparisons does BubbleSort make on an array of size n that is sorted from largest to smallest?

   Solution: The answer is the same, $1 + 2 + \cdots + (n-1) = \dfrac{(n-1)n}{2}$ comparisons.

   For this particular algorithm (BubbleSort) the number of comparisons is independent of the input.

   (c) (2 points) Explain the difference between BubbleSort and RevisedBubbleSort.

   Solution: RevisedBubbleSort breaks out of the outer loop if no swaps were performed in the inner loop. That is, it recognizes when the array is sorted and terminates early if it makes a pass through the array without needing to swap anything.

(d) (2 points) How many comparisons does *RevisedBubbleSort* make on an array of size n that is already sorted from smallest to largest?

Solution: *RevisedBubbleSort* makes $n-1$ comparisons on an array of size n that is sorted from smallest to largest. This is the number of comparisons made for the first iteration of the outer loop. No other iterations are done because no swaps were made.

(e) (2 points) How many comparisons does *RevisedBubbleSort* make on an array of size n that is sorted from largest to smallest?

Solution: *RevisedBubbleSort* makes $\frac{(n-1)n}{2}$ comparisons on an array of size n that is sorted from largest to smallest. On the i-th pass, the largest element must get swapped all the way from the front of the array to the back. Thus, every comparison results in a swap, and there is no early termination.

(f) (10 points) Prove the following loop invariant for *BubbleSort*.

Loop invariant: After the i^{th} pass, the last i elements $a_{n-i+1}, \ldots, a_n$ are the i largest elements of the input, in sorted order.

Solution: We do a proof by induction on i, the number of outer loop iterations. When $i = 0$, before we run the outer loop, the last 0 positions contain the 0 largest elements in sorted order. This is trivially true because the last 0 positions is the empty set and anything is true of the empty set.

For the induction hypothesis, suppose for some value of i, that after the i^{th} pass, the last i elements $a_{n-i+1}, \ldots, a_n$ are the i largest elements of the input, in sorted order. We must show that after the $(i+1)^{st}$ pass, the elements $a_{n-i}, \ldots, a_n$ are the $i+1$ largest elements of the input, in sorted order.

During the $(i+1)^{st}$ iteration of the outer loop, the algorithm finds the maximum value among the first $n-i$ elements of the array and places it in position a_{n-i}. We can see this because the inner loop compares consecutive elements and swaps elements as necessary so that the larger element of the pair ends up further to the right. Whenever the maximum value among the first $n-i$ elements is encountered, it will be swapped all the way to the right as far as possible, which is a_{n-i}.

Thus after the $(i+1)^{st}$ iteration of the outer loop, a_{n-i} is larger than all of a_1 through a_{n-i-1}. At the same time, by the inductive hypothesis, the last i elements of A contain the i largest elements. Since a_{n-i} is not among them, it is not one of the i largest elements. But it is larger than everything in a_1 through a_{n-i-1}, which means it is the $i+1$-st largest element. This says that the elements $a_{n-i}, \ldots, a_n$ are the $i+1$ largest elements of the input, in sorted order, which is what we wanted to show.

(g) (5 points) List all possible inputs containing the **distinct** numbers 1, 2, 3, 4, and 5 for which *RevisedBubbleSort* will terminate with a value of $i = 2$.

Solution: There are 15 such imputs:

15234,	14235,	13245,	13254,	12354,
12435,	12534,	51234,	41235,	31245,
31254,	21345,	21354,	21435,	21534.

(h) (5 points) For the input $(1, 2, 3, 4, 5, 6)$, the algorithm *RevisedBubbleSort* does **5** comparisons. Give a different input with the same entries for which *RevisedBubbleSort* also does 5 comparisons, or explain why no such input exists.

Solution: There is no other case. After one iteration of the **for** loop, we have used exactly 5 comparisons between list elements. Thus, in order or the algorithm to do exactly 5 comparison, it must terminate after the first iteration which means *done* must be *true* at the end of the iteration. Observe that once we set *done* to $false$, we cannot turn it back to *true* within the same iteration. Hence, it must be the case that *done* is never set to $false$ within the first iteration. Therefore, it must be the case that the input is an increasing sequence.

2. Consider to the following two algorithms.

procedure SortA($a_1, a_2, \ldots, a_n$: a list of real numbers with $n \geq 2$)

```
1.  for j := 2 to n
2.      i := 1
3.      while a_j > a_i
4.          i := i + 1
5.      m := a_j
6.      for k := 0 to j − i − 1
7.          a_{j−k} := a_{j−k−1}
8.      a_i := m
```

procedure SortB($a_1, a_2, \ldots, a_n$: a list of real numbers with $n \geq 2$)

```
1.  for j := 2 to n
2.      i := j
3.      while (i > 1 AND a_j < a_{i−1})
4.          i := i − 1
5.      m := a_j
6.      for k := 0 to j − i − 1
7.          a_{j−k} := a_{j−k−1}
8.      a_i := m
```

(a) (3 points) Trace the behavior of SortA on the input list $3, 2, 5, 6, 4, 1$ by filling in the following table. Each row corresponds to the completion of one iteration of the outermost loop. When listing the comparisons done in each iteration, say which two elements are being compared in each comparison (example: 3 vs. 5).

After the...	What the list looks like	Comparisons done in this iteration	Total number of comparisons done so far
0th iteration	$3, 2, 5, 6, 4, 1$	none	0
1st iteration			
2nd iteration			
$\vdots$			

Solution: SortA compares each element with other elements in the list, starting from the beginning, until it finds an element of greater or equal value (possibly itself). This is what causes the **while** loop in line 3 to end.

After the...	What the list looks like	Comparisons done in this iteration	Total number of comparisons done so far
0th iteration	$3, 2, 5, 6, 4, 1$	none	0
1st iteration	$2, 3, 5, 6, 4, 1$	2 vs 3	1
2nd iteration	$2, 3, 5, 6, 4, 1$	5 vs 2, 5 vs 3, 5 vs 5	4
3rd iteration	$2, 3, 5, 6, 4, 1$	6 vs 2, 6 vs 3, 6 vs 5, 6 vs 6	8
4th iteration	$2, 3, 4, 5, 6, 1$	4 vs 2, 4 vs 3, 4 vs 5	11
5th iteration	$1, 2, 3, 4, 5, 6$	1 vs 2	12

(b) (3 points) Trace the behavior of SortB on the input list $3, 2, 5, 6, 4, 1$ by filling in the following table. Each row corresponds to the completion of one iteration of the outermost loop. When listing the comparisons done in each iteration, say which two elements are being compared in each comparison (example: 3 vs. 5).

After the...	What the list looks like	Comparisons done in this iteration	Total number of comparisons done so far
0th iteration	$3, 2, 5, 6, 4, 1$	none	0
1st iteration			
2nd iteration			
$\vdots$			

Solution: SortB compares each element with other elements in the list, starting from the element immediately before and working to the left, until it finds an element of lesser

or equal value, or until it reaches the first position in the list. This is what causes the **while** loop in line 3 to end.

After the...	What the list looks like	Comparisons done in this iteration	Total number of comparisons done so far
0th iteration	$3, 2, 5, 6, 4, 1$	none	0
1st iteration	$2, 3, 5, 6, 4, 1$	2 vs 3	1
2nd iteration	$2, 3, 5, 6, 4, 1$	5 vs 3	2
3rd iteration	$2, 3, 5, 6, 4, 1$	6 vs 5	3
4th iteration	$2, 3, 4, 5, 6, 1$	4 vs 6, 4 vs 5, 4 vs 3	6
5th iteration	$1, 2, 3, 4, 5, 6$	1 vs 6, 1 vs 5, 1 vs 4, 1 vs 3, 1 vs 2	11

Note: For parts (c)-(f), the input elements are distinct.

(c) (2 points) Find (with explanation) an input list containing the numbers $1, 2, 3, 4, 5$, and 6 for which SortA does the fewest possible number of comparisons (i.e. a best-case input).

Solution: Since SortA compares starting at the beginning of the list, the fewest comparisons will be made when each element is smaller than all previous elements in the input list, and can be inserted immediately after just one comparison. Therefore, a best-case input to SortA is the list $6, 5, 4, 3, 2, 1$.

(d) (2 points) Find (with explanation) an input list containing the numbers $1, 2, 3, 4, 5$, and 6 for which SortB does the fewest possible number of comparisons (i.e. a best-case input).

Solution: Since SortB compares starting at the end of the sorted part of the list, the fewest comparisons will be made when each element is larger than all previous elements in the input list, and can be inserted immediately after just one comparison. Therefore, a best-case input to SortB is the list $1, 2, 3, 4, 5, 6$.

Another possible best-case input is $2, 1, 3, 4, 5, 6$, which also does the same number of comparisons (five).

(e) (2 points) Find (with explanation) an input list containing the numbers $1, 2, 3, 4, 5$, and 6 for which SortA does the greatest possible number of comparisons (i.e. a worst-case input).

Solution: Since SortA compares starting at the beginning of the list, the greatest number of comparisons will be made when each element is larger than all previous elements in the input list, and will have to be compared to all previous list elements. Therefore, a worst-case input to SortA is the list $1, 2, 3, 4, 5, 6$.

(f) (2 points) Find (with explanation) an input list containing the numbers $1, 2, 3, 4, 5$, and 6 for which SortB does the greatest possible number of comparisons (i.e. a worst-case input).

Solution: Since SortB compares starting at the end of the sorted part of the list, the greatest number of comparisons will be made when each element is smaller than all previous elements in the input list, and will have to be compared to all previous list elements. Therefore, a worst-case input to SortB is the list $6, 5, 4, 3, 2, 1$.

Another possible worst-case input is $5, 6, 4, 3, 2, 1$, which also does the same number of comparisons (fifteen).

(g) (3 points) For the input list $1, 2, 3, \ldots, n-1, n$ (when the input happens to be already sorted), how many comparisons does SortA perform, in terms of n? Simplify your answer.

Solution: On this input, each element j from 2 to n will be compared to all elements before it, and compared to itself. So j comparisons are done for each element j from 2 to n. Therefore, the total number of comparisons is

$$\begin{aligned} 2 + 3 + \cdots + n &= (1 + 2 + 3 + \cdots + n) - 1 \\ &= \frac{n(n+1)}{2} - 1 \\ &= \frac{n^2 + n - 2}{2} \end{aligned}$$

(h) (3 points) For the input list $1, 2, 3, \ldots, n-1, n$ (when the input happens to be already sorted), how many comparisons does SortB perform, in terms of n? Simplify your answer.

Solution: On this input, each element j from 2 to n will be compared only to the element immediately before it. So 1 comparison is done for each element j from 2 to n. Since there are $n-1$ such j values, the total number of comparisons is $n-1$.

3. In this problem, we are given a sequence $a_1, a_2, \ldots, a_n$ of integers and we want to return a list of all terms in the sequence that are greater than the sum of all previous terms of the sequence. For example, on an input sequence of $1, 4, 6, 3, 2, 20$, the output should be the list $1, 4, 6, 20$. The following algorithm solves this problem.

procedure PartialSums($a_1, a_2, \ldots, a_n$: a sequence of integers with $n \geq 1$)

```
1.    total := 0
2.    Initialize an empty list L.
3.    for i := 1 to n
4.        if a_i > total
5.            Append a_i to list L.
6.        total := total + a_i
7.    return L
```

(a) (6 points) Prove the following loop invariant by induction on the number of loop iterations:

Loop Invariant: After the t^{th} iteration of the **for** loop, $total = a_1 + a_2 + \cdots + a_t$ and L contains all elements from $a_1, a_2, \ldots, a_t$ that are greater than the sum of all previous terms of the sequence.

Solution: The base case is when $t = 0$, before the loop ever iterates. In line 1, *total* is set to 0, which is the value of the sum $a_1 + a_2 + \cdots + a_0$ because this sum has no terms. In line 2, L is set to the empty list, which does contain all elements from the set $\{a_1, a_2, \ldots, a_0\}$ that are greater than the sum of all previous terms of the sequence, because this set is empty and anything is true of the empty set. Thus, the invariant is true when $t = 0$.

For the induction step, let t be a positive integer. Suppose that we have gone through the loop $t-1$ times, and that $total = a_1 + a_2 + \cdots + a_{t-1}$ and L contains all elements from $a_1, a_2, \ldots, a_{t-1}$ that are greater than the sum of all previous terms of the sequence. On the tth iteration of the loop, the value of i is t. In lines 4 and 5, a_t is added to list L if and only if $a_t > total$, which equals $a_1 + a_2 + \cdots + a_{t-1}$ by the inductive hypothesis. So a_t is added to list L if and only if it is greater than the sum of all previous terms of the sequence. Since, by the inductive hypothesis, L already contained all elements from $a_1, a_2, \ldots, a_{t-1}$ that are greater than the sum of all previous terms of the sequence and we have added a_t exactly when it also satisfies this property, we know that now after the tth iteration, L contains all elements from $a_1, a_2, \ldots, a_t$ that are greater than the sum of all previous terms of the sequence. Similarly, on the tth iteration of the loop, we add a_t to the current $total$ in line 6. Since we know from the inductive hypothesis that $total = a_1 + a_2 + \cdots + a_{t-1}$ and we just added a_t, then after the tth iteration, we have $total = a_1 + a_2 + \cdots + a_t$. This proves the inductive step, and so we conclude that the loop invariant is true for all $t \geq 0$ as desired.

(b) (4 points) Use the loop invariant to prove that the algorithm is correct, i.e., prove that it returns a list of all terms in the sequence that are greater than the sum of all previous terms of the sequence.

Solution: The for loop in this algorithm runs n times, as seen in line 3. We have already proved that the loop invariant is true for all values of $t \geq 0$, in particular when $t = n$. This says that after the nth iteration of the **for** loop (that is, when the algorithm is finished), $total = a_1 + a_2 + \cdots + a_n$ and L contains all elements from $a_1, a_2, \ldots, a_n$ that are greater than the sum of all previous terms of the sequence. The algorithm returns the value of L, which is a list containing all elements from $a_1, a_2, \ldots, a_n$ that are greater than the sum of all previous terms of the sequence, which is exactly what this algorithm was supposed to return according to the problem specification.

4. (10 points) In this problem, your goal is to identify an individual that carries the deadly Z-virus among a group of n people. You collect a blood sample from each of the people in the group, and label them 1 through n. Suppose that you know in advance that **exactly one** person is infected with the Z-virus, and you must identify who that person is by performing blood tests.

In a single blood test, you can specify any subset of the samples, combine a drop of blood from each of these samples, and then get a result. If any sample in the subset is infected, the test will come up positive, otherwise it will come up negative. Your goal is to find the infected person with **as few blood tests as possible**. Give an algorithm that finds the infected sample in a set of n blood samples, using as few tests as you can. Write pseudocode and an English description of how your algorithm works.

Remark: This idea of testing multiple samples at a time has been used in history at times when it was impractical or expensive to perform blood tests, for example, to find out which soldiers in World War II were carrying diseases. In those situations, the problem was even harder because there could be any number of infected people in the group.

Solution: Pseudocode:

FindInfected (blood samples labeled 1 through n, exactly one of which is infected)

1. $i := 1, \quad j := n$
2. While $i < j$
3. $\quad m := \lfloor (i + j)/2 \rfloor$
4. $\quad$ Combine some blood from samples i through m and test.
5. $\quad$ If test is positive, set $j := m$.
6. $\quad$ Else, set $i := m + 1$
7. Return i.

English description: The algorithm works by repeatedly splitting the samples in two halves. It will combine drops of blood from each individual in one of the halves, and test that combination of blood. If it comes up positive, we know the infected person is among the half that we tested. If the test is negative, we know the other half contains the infected sample. This reduces our problem now to testing half as many samples as we originally had. We continue testing one half at a time until we reduce our search range down to one sample. At this point, we know the one sample must be the infected sample, so we return it.

This process is essentially binary search because we are searching for our infected person by splitting the search range in half each time. Therefore, it takes about $log_2 n$ tests to narrow the search range down to a single person. We saw in class that binary search was the most efficient searching algorithm, so this solution solves the problem using as few tests as possible.