INSTRUCTIONS

KEY CONCEPTS Sorting algorithms, including selection (min) sort, insertion sort, and bubble sort; loop invariants and correctness proofs; searching algorithms; counting comparisons.
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. This problem refers to the following two algorithms.

**procedure** SortA($a_1, a_2, \ldots, a_n$: a list of real numbers with $n \geq 1$)

1. for $i := 1$ to $n - 1$
2. item := $a_i$
3. location := $i$
4. for $j := i + 1$ to $n$
5. if $a_j < \text{item}$ then
6. item := $a_j$
7. location := $j$
8. $a_{\text{location}} := a_i$
9. $a_i := \text{item}$

**procedure** SortB($a_1, a_2, \ldots, a_n$: a list of real numbers with $n \geq 1$)

1. for $k := 1$ to $n - 1$
2. $i := n - k + 1$
3. item := $a_i$
4. location := $i$
5. for $j := 1$ to $i - 1$
6. if $a_j > \text{item}$ then
7. item := $a_j$
8. location := $j$
9. $a_{\text{location}} := a_i$
10. $a_i := \text{item}$

(a) (4 points) State, but do not prove, a loop invariant that could be used to show SortA correctly solves the sorting problem.

**Solution**: After the $t$th iteration of the outer loop, the first $t$ list elements are the $t$ smallest elements of the input, in sorted order.

(b) (5 points) State, but do not prove, a loop invariant that could be used to show SortB correctly solves the sorting problem.

**Solution**: After the $t$th iteration of the outer loop, the last $t$ list elements are the $t$ largest elements of the input, in sorted order.
2. To find the largest element in an unsorted array takes \( n - 1 \) comparisons, as does finding the smallest element in an unsorted array. Present, prove correct, and analyse comparisons for an algorithm that uses \([3/2n] - 2\) comparisons to find both the largest and smallest elements of an unsorted array. (2 points algorithm description, 4 points correctness proof, 2 points correct number of comparisons, 2 points correctly analysing number of comparisons)

The algorithm when \( n = 2k \) is even is as follows. Break up the \( n \) elements in the array into \( k = n/2 \) groups of 2, i.e., \( A[2i - 1] \) and \( A[2i] \) form a group, and compare each pair. If the first element in the group is larger than the second, swap them. Thus, the total number of comparisons is \( k \) + 1\( \sum_{i=1}^{k} 2 = 2 \left( [k/2] + 1 \right) \). The last iteration is when \( k = n - 2 \), so it will take \( 3(k - 1) - 2 \) comparisons and leave \( n - 4 \) elements left. In general, the \( I \)th iteration, we have \( n - 2(I - 1) = 2(k - I + 1) \) elements left, so finding the max and min takes \( 3(k - I + 1) - 2 \) comparisons. The last iteration is when \( I = n/2 = k \).

Thus, the total number of comparisons is \( 3 \sum_{I=1}^{k} (k - I + 1) - 2 = 3 \sum_{I=1}^{k} (k - I + 1) - \sum_{I=1}^{k} 2 = 3 \left( \sum_{I=1}^{k} J \right) - 2k \). The first equation is using the commutivity and distributivity rules, the second is a change of variables \( J = k - I + 1 \), and reversing the order of summation, as well as realising \( k \) 2's sum up to 2\( k \); the third is applying

3. Say we use the above algorithm to improve selection sort as follows: procedure ImpSelectSort(a_1, a_2, \ldots, a_n: a list of distinct real numbers with \( n \geq 2 \) even)

1. for \( k := 1 \) to \( n/2 \) do:
2. begin{do}
3. Use the algorithm from the previous problem to find the locations \( i \) of the maximum element and \( j \) of the minimum element in the sub-array \( a[k] \ldots a[n - k + 1] \).
4. Swap \( a[k] \) and \( a[i] \): swap \( a[n - k + 1] \) and \( a[j] \)
5. end{do}
6. Return the array \( a_1, \ldots, a_n \)

How many comparisons does this algorithm make? How does this compare to the number of comparisons selection sort makes? (10 points: give an explanation for your answers, but you do not need to give complete proofs.)
our well-known “little Gauss” formula for the sum of the first $k$ integers; and the rest is algebraic simplification, since $k = n/2$.

Since we already saw that the unimproved version of Select Sort does $n(n - 1)/2$ comparisons, meaning the co-efficient of $n^2$ is $1/2$ vs. $3/8$ in the improved version, the improved version does about 25% fewer comparisons for large $n$.

4. The predecessor problem is, given a sorted array $a_1, a_2, \ldots a_n$ and a target $x$, find the position $j$ of the largest $a_j \leq x$, or return 0 if $x$ is smaller than all elements in the array. Give a version of the binary search algorithm that solves this problem, and prove that it is correct. (4 points algorithm description, 6 points correctness proof).

We need to make a few changes to the standard binary search. First, we should check some edge conditions. Does $x$ have a predecessor in the array, or is it smaller than all of the elements in the array? Secondly, if we learn that $a_m < x$, we cannot rule $m$ out as a possible predecessor, since a predecessor might be smaller. (However, we can rule out $1 \ldots m - 1$ as possible predecessors, since although they are smaller, $a_m$ is a larger smaller value, so they are not the largest smaller value (unless they are equal to $a_m$, but removing duplicates doesn’t hurt us.)

This gives us the following modified binary search algorithm:

**Pred($a[1..n], x$).**

(a) If $x < a_1$ return 0.
(b) If $x \geq a_n$ return $n$.
(c) $i \leftarrow 1, j \leftarrow n$.
(d) While $j > i + 1$ do:
   (e) $m \leftarrow \lfloor (i + j)/2 \rfloor$
   (f) IF $a_m \leq x$ THEN $i \leftarrow m$ ELSE $j \leftarrow m$.
(g) Return $i$

If $x < a_1$, there is no smaller element in the array, so the predecessor is 0, which is returned in line 1. If $x \geq a_n$, then the predecessor is $n$, because all elements in the array are $\leq x$, and $a_n$ is the largest such element.

Otherwise, we’ll show that the algorithm maintains the invariant: $a_i \leq x < a_j$, which we prove by the number of previous iterations of the while loop, $t$. If we get to line 3 at all, we know $a_1 \leq x < a_n$. Then before the loop starts, i.e., $t = 0$, we set $i = 1, j = n$, so $a_i = a_1 \leq x < a_n = a_j$, so the invariant is true for the base case, $t = 0$.

Assume the invariant holds after $t$ iterations. Then during the $t + 1$st iteration, we either have $a_m \leq x$ or $a_m > x$. In the first case, $i$ becomes $m$ and $j$ doesn’t change, so $a_m = a_i \leq x < a_j$ (with the second condition holding due to the induction hypothesis). In the second case, $j$ becomes $m$, and we have $a_i \leq x < a_m = a_j$, with the first inequality from the induction hypothesis. In either case, the loop invariant still holds after $t + 1$ iterations, so by induction, it holds throughout the loop.

Note that when we exit the loop, currently $j \leq i + 1$, but previously, $j > i + 2$. Then in the previous loop, since the gap between $i$ and $j$ is at least 2, the midpoint $m$ differs from both by at least 1, so $i < m < j$, so in the next iteration $i < j$. So the only way $i < j$ and $j \leq i + 1$ is if $j = i + 1$. Thus, at the end of the loop, by the loop invariant, $a_i \leq x < a_{i+1}$, so $a_i$ is less than or equal to $x$, but any larger value than $a_i$ in the array is also greater than $x$. So the predecessor of $x$ must be $i$, which is what we return.

The problem doesn’t ask for it, but like the binary search algorithm in class, we halve the range of $j - i$ every iteration. However, we have an additional 2 comparisons before the loop starts, so the total number of comparisons will be at most $\log n + 2$. 

4
5. Say we have an object of some unknown weight $w > 1$ in grams, an unlimited supply of 1 gram weights, and a balance with two arms, which will tip down the arm with the heavier total amount in it. Show how we can determine $\lfloor w \rfloor$, i.e., $w$ to within a gram, with at most $2\lceil \log w \rceil$ uses of the balance.

The first challenge is that we don’t have even a range of possibilities for $w$. So we’ll need to compare $w$ to larger and larger weights, until we find one bigger. We need to balance two things: First, the number of weighings until we get larger than $w$, and secondly, the number of weighings we’ll need to perform a binary search to get the exact value, once we find the range. To minimize both, we’ll increase the size of the weights we compare $w$ to exponentially, which won’t be too bad for the second, because we’ll overshoot $w$ by at most a factor of 2.

I’ll present the method we use as pseudo-pseudocode.

\textit{Weigh(object).}

(a) $J \leftarrow 2, I \leftarrow 1$

(b) While we haven’t exited do:
(c) Weigh $J$ 1 g weights vs. the object
(d) If the $J$ weights are the same as the object, return $J$
(e) If the weights are heavier, exit the loop, ELSE $I \leftarrow J, J \leftarrow 2J$.
(f) While $J > I + 1$ do:
(g) $M \leftarrow \lceil (I + J)/2 \rceil$
(h) Compare the object to $M$ weights.
(i) IF equal, return $M$
(j) If the weights are heavier, $J \leftarrow M$ ELSE $I \leftarrow M$.
(k) Return $I$

Since the object weighs $w > 1$ grams, it always weighs more than $I$ grams during the first loop. At the end of the first loop, we exit because it weighs less than $J = 2I$ grams. Thus, going into the second loop, $I < w < J$. We’ll show that this is true as a loop invariant: If we have done $t$ weighings in the second loop, then $I < w < J$. Assume this is true. If $M = w$, we exit the loop in the next iteration. If $M > w$, we set $J$ to $M$, and leave $I$ unchanged, so $I < w < M = J$. If $M < w$, we set $I$ to $M$ and leave $J$ unchanged, so $I = M < w < J$. So the invariant holds after the next iteration, and by induction, holds throughout the loop.

When we exit the second loop, $I < w < J = I + 1$, so $I = \lfloor w \rfloor$, which is what we return. If we return earlier, we return a value that is exactly $w$. So either way, we return the correct value.

In the first loop, in the $t$’th weighing, $J = 2^t$. We exit when $J = 2^t \geq w$, i.e., when $t = \lceil \log w \rceil$. In the second loop, we start with $J = 2I > w > I$, so $J - I = I < w$, and $J - I$ is halved every iteration. Thus, we have at most $\log w$ iterations before $J - I \leq 1$. Thus, the total number of weighings is at most $2\lceil \log w \rceil$ total.

(10 points, 3 points method description, 3 points correctness argument, 2 points correct number of weighings, 2 points correct analysis of the number of weighings.)