CS3230 Tutorial 03: Correctness and Divide-and-Conquer

Solutions written for the questions in Tutorial 03.

Question 1. Consider InsertionSort(A[0..N-1]). Assume the inner loop on j is correct: assuming $A[0..i-1]$ is sorted, it inserts $A[i]$ into its correct position within $A[0..i]$ without changing $A[i+1..N-1]$.
(a) Suitable loop invariant for the outer loop i.

Solution. A suitable loop invariant for the outer for-loop (index i) is:

Invariant: *At the start of each outer-loop iteration with index* i *(for* $1 \leq i \leq N-1$*), the prefix* $A[0..i-1]$ *is sorted in nondecreasing order and is a permutation of the original elements that were initially in positions* $0..i-1$. □

(b) Show initialization, maintenance, termination.

Solution. We prove correctness of the outer loop using the above invariant.
Initialization. At the start of the first iteration, $i = 1$. The subarray $A[0..0]$ has a single element, hence it is sorted. It is trivially a permutation of the original elements in positions 0..0. Therefore, the invariant holds before the first iteration.
Maintenance. Assume the invariant holds at the start of some iteration i (where $1 \leq i \leq N-1$), i.e., $A[0..i-1]$ is sorted. During this iteration, we set $X = A[i]$ and run the inner loop on j. By the problem assumption, the inner loop correctly inserts $A[i]$ into its correct position among $A[0..i-1]$, producing a sorted subarray $A[0..i]$, and it does not modify any elements in $A[i+1..N-1]$. Hence, at the end of iteration i, the prefix $A[0..i]$ is sorted and contains exactly the original elements of $A[0..i]$ (i.e., still a permutation). Therefore, at the start of the next iteration $(i+1)$, the invariant holds with the prefix $A[0..(i+1)-1] = A[0..i]$.
Termination. The outer loop terminates after finishing iteration $i = N-1$. By maintenance, at that point $A[0..N-1]$ is sorted. Thus, the entire array is sorted when the algorithm ends, proving correctness. □

Question 2. Consider StoogeSort(A): if $n = 2$ and $A[0] > A[1]$ swap; if $n > 2$, recursively sort the first $\lceil 2n/3 \rceil$ elements, then the last $\lceil 2n/3 \rceil$ elements, then the first $\lceil 2n/3 \rceil$ elements again. Assume all numbers are distinct.
(a) Prove that StoogeSort(A) correctly sorts the input array A. second version.

Solution. We prove by *strong induction* on $n = |A|$ that StoogeSort outputs A in strictly increasing order (all elements are distinct).
Base cases. If $n = 1$, A is already sorted. If $n = 2$, the algorithm swaps $A[0], A[1]$ iff $A[0] > A[1]$, so the output is sorted.
Inductive hypothesis. Assume that for every array of length $< n$, StoogeSort correctly sorts it.
Inductive step ($n > 2$). Let

$$k = \left\lceil \frac{2n}{3} \right\rceil, \qquad m = n - k.$$

Note that $m = n - k \leq \lfloor n/3 \rfloor$ and, crucially, $m \leq k - 1$, so the two subarrays below overlap:

$$F := A[0..k-1] \quad \text{(first } k \text{ elements)}, \qquad L := A[m..n-1] \quad \text{(last } k \text{ elements)}.$$

Also define the (disjoint) right-only suffix

$$R := A[k..n-1],$$

whose length is $|R| = n - k = m$.
The algorithm performs:

(1) sort F recursively,
(2) sort L recursively,
(3) sort F recursively again.

Each recursive call is on a subarray of length $k < n$, so by the inductive hypothesis, each call sorts its subarray correctly.
After step 2, the subarray $L = A[m..n-1]$ is sorted in increasing order. Since $R = A[k..n-1]$ is a *suffix* of L, it follows that R is also sorted after step 2. Moreover, because L is sorted and the indices $m..k-1$ come *before* $k..n-1$ within L, we have

$$(\star) \qquad \max\bigl(A[m..k-1]\bigr) < \min\bigl(A[k..n-1]\bigr),$$

where the inequality is strict since all elements are distinct.
Step 3 sorts $F = A[0..k-1]$ again. This does not change any element of $R = A[k..n-1]$, hence R remains sorted in the final array.
Now consider the final state after step 3:

- $F = A[0..k-1]$ is sorted (by step 3),
- $R = A[k..n-1]$ is sorted (as argued above),
- the last element of F is $A[k-1] = \max(F)$, and since $k-1 \geq m$, this position lies in the overlap $[m..k-1]$.

Therefore,

$$\max(F) = A[k-1] \leq \max\bigl(A[m..k-1]\bigr).$$

Combining with $(\star)$ gives

$$A[k-1] = \max(F) < \min(R) = A[k].$$

Thus every element in the sorted prefix F is smaller than every element in the sorted suffix R; since F and R are contiguous and individually sorted, the entire array $A[0..n-1]$ is sorted in increasing order.
This completes the inductive step, hence StoogeSort correctly sorts A for all n. $\square$

(b) Analyze the time complexity of StoogeSort.

Solution. Let $T(n)$ be the worst-case running time on length n. For $n \leq 2$, $T(n) = \Theta(1)$. For $n > 2$, the algorithm makes three recursive calls on size $k = \lceil 2n/3 \rceil$ plus $O(1)$ overhead:

$$T(n) = 3T(\lceil 2n/3 \rceil) + O(1).$$

Ignoring ceilings (they do not change the asymptotic), write:

$$T(n) = 3T(2n/3) + O(1).$$

Using the Master/recursion-tree style result, this solves to:

$$T(n) = \Theta\bigl(n^{\log_{3/2} 3}\bigr).$$

Since $\log_{3/2} 3 = \frac{\ln 3}{\ln(3/2)} \approx 2.7095$, we get

$$T(n) = \Theta\bigl(n^{2.7095\ldots}\bigr).$$

$\square$

Question 3. Show that there is a peak in every $m \times n$ 2D array. A peak is a cell whose value is no smaller than all of its (up to) four neighbors (top/right/bottom/left).

Solution. Consider a cell that contains a *global maximum* value in the entire array (i.e., a cell (r, c) such that $A[r, c] \geq A[i, j]$ for all (i, j)). Then in particular, $A[r, c]$ is at least as large as each of its neighbors (since each neighbor is also a cell in the array). Therefore $A[r, c]$ is a peak by definition. Hence every 2D array has at least one peak. $\square$

Question 4. What is the runtime complexity of FindPeakSp(A)?

Solution. Let the array have m rows and n columns. Let $T(m, n)$ be the worst-case runtime.

- If $n = 1$, the algorithm returns a maximal element in the only column, which takes $\Theta(m)$ time. So $T(m, 1) = \Theta(m)$.
- If $n \geq 2$, it:
  (1) selects the middle column ($O(1)$),
  (2) finds a maximal element in that column ($\Theta(m)$),
  (3) checks whether it is a peak ($O(1)$),
  (4) and in the worst case, recurses on *both* halves (left half and right half), each with about $n/2$ columns.

Thus, in the worst case:

$$T(m, n) = 2T(m, \lfloor n/2 \rfloor) + \Theta(m).$$

Treating m as a parameter and solving the recurrence in n:

$$T(m, n) = \Theta(mn).$$

(Indeed, $a = 2$, $b = 2$, so $n^{\log_b a} = n$, and the non-recursive work is $\Theta(m)$ per node; summing over the recursion tree gives $\Theta(mn)$.)
(more detailed reason) Recall the recurrence (for $n \geq 2$):

$$T(m, n) = 2T\Big(m, \Big\lfloor \frac{n}{2} \Big\rfloor\Big) + \Theta(m).$$

Define a new function

$$S(n) := \frac{T(m, n)}{m},$$

where we treat m as a fixed parameter. Dividing both sides of the recurrence by m, we obtain

$$\frac{T(m, n)}{m} = 2 \cdot \frac{T\big(m, \lfloor \frac{n}{2} \rfloor\big)}{m} + \frac{\Theta(m)}{m}.$$

That is,

$$S(n) = 2S\Big(\Big\lfloor \frac{n}{2} \Big\rfloor\Big) + \Theta(1).$$

Ignoring floors (which do not affect asymptotic growth), this has the standard form

$$S(n) = 2S\Big(\frac{n}{2}\Big) + \Theta(1).$$

By the Master Theorem with $a = 2$, $b = 2$, and $f(n) = \Theta(1)$, we have

$$S(n) = \Theta(n).$$

Finally, since $T(m, n) = mS(n)$, it follows that

$$T(m, n) = m \cdot \Theta(n) = \Theta(mn).$$

□

Question 5. Argue why FindPeakSp(A) will never return `None`. Additionally, discuss whether any steps within the `else` condition (Step 8) can be optimized asymptotically.

Solution. We prove that FindPeakSp always returns a peak (and hence never returns `None`) by induction on the number of columns n.
Claim. For any $m \times n$ array A, FindPeakSp returns a peak (in fact, a special-peak).
Base case ($n = 1$). The algorithm returns a maximal element in the only column. That element is $\geq$ its up/down neighbors (if any), and it has no left/right neighbors. Hence it is a peak. So it does not return `None`.
Inductive step ($n \geq 2$). Let C_m be the middle column, and let p be a maximal element in C_m (Step 5).

If p is a peak, the algorithm returns it (Step 6–7), done.
Otherwise, since p is maximal in its column, the only way it can fail to be a peak is that one of its *horizontal* neighbors is larger: either the left neighbor p_L (in column $m-1$) satisfies $p_L > p$, or the right neighbor p_R (in column $m+1$) satisfies $p_R > p$.
WLOG assume $p_L > p$ (the right case is symmetric). Then the left half of the array (all columns strictly left of C_m) contains an element larger than p. Let q be a *global maximum within the left half* (maximum over all cells in the left half). We show q is a peak in the *entire* array:

- Any neighbor of q that lies within the left half has value $\leq q$ by maximality of q in the left half.
- The only possible neighbor of q outside the left half is a right neighbor in the middle column C_m (this happens only if q is in the column adjacent to C_m). But every value in C_m is $\leq p < p_L \leq q$, since p is the maximum of C_m and q is at least as large as p_L. So the right neighbor (if it exists) is also $\leq q$.

Hence q is a peak. Therefore, the left half contains a peak.
Now, FindPeakSp recursively calls itself on the left half (Step 9) and the right half (Step 10). By the inductive hypothesis, the recursive call on the left half must return a peak (not `None`). Therefore Step 11 succeeds and the algorithm returns a peak; it never reaches Step 14.
Thus, by induction, FindPeakSp never returns `None`.

Asymptotic optimization of Step 8 (`else` branch). The current algorithm recurses on *both* halves, giving worst-case

$$T(m, n) = 2T(m, n/2) + \Theta(m) = \Theta(mn).$$

However, if the maximal element p in the middle column is not a peak, we can compare its left/right neighbor(s) and recurse into *only the side that contains a larger neighbor*. This is the classic 2D peak-finding idea: a larger neighbor indicates there exists a peak in that direction. With this optimization, the recurrence becomes:

$$T(m, n) = T(m, n/2) + \Theta(m) = \Theta(m \log n),$$

which is asymptotically faster than $\Theta(mn)$.
(We still need $\Theta(m)$ time per level to find a maximum in the chosen middle column, unless additional preprocessing/data structures are allowed.) $\square$