CS3230 Tutorial 3

Deng Tianle (T15)

5 September 2025

Agenda

Correctness proofs:

- iterative algorithm: insertion sort (Q1)
- recursive algorithm: 'stooge' sort (Q2)

Divide and conquer:

- peak-finding problem (Q3-5; also LeetCode problem)

Q1

Insertion sort for array $A[0 \ldots N-1]$:

```
for (i=1; i<N; i++) {
  x = A[i];
  for (j=i-1; j>=0 && A[j]>x; j--) {
    A[j+1]=A[j];
  }
  A[j+1]=x;
}
```

Idea: for i from 1 to $n-1$, at each step make $A[0 \ldots i]$ sorted by inserting $A[i]$ at the correct place (refresher: why is this sort stable?).

Q1: what does each iteration of the outer loop accomplish? Can you prove that it is so?

Q1

Let B denote (a copy of) the original array. Loop invariant: $A[0 \ldots i]$ is the sorted version of $B[0 \ldots i]$ and $A[i+1 \ldots N-1] = B[i+1 \ldots N-1]$. [1]

- Initialisation (base case): $A[0]$ is sorted (only one element),
- Maintenance (step): assume the inner loop is correct[2], since $A[0 \ldots i-1]$ is the sorted version of $B[0 \ldots i-1]$ and $A[i] = B[i]$, the invariant holds for i.
- Termination (conclusion): $A[0 \ldots n-1]$ sorted, as desired.

Remark

In most cases we can think of loop invariant as inducting on the iteration number.

---

[1] Actually, $A[i+1] = B[i+1]$ is enough as we will see below.

[2] We follow the assumption printed in the tutorial sheet; we can also use the weaker assumption that it inserts $A[i]$ correctly, but we then need to explain a little bit more here.

Q1

Remark

To be fully rigorous, the termination is that $A[0 \ldots n-1]$ *is sorted AND it is a permutation of* B*. Therefore, it will be good if you include information about what are the elements in* $A[0 \ldots i]$ *in your loop invariant, even though it is obvious we are just permuting elements of* A *throughout. This is why the given loop invariant is more than just '*$A[0 \ldots i]$ *is sorted'.*

Q2

For simplicity, assume $A[0 \ldots n-1]$ consists of distinct elements. Motivation: how to sort $[a, b, c]$? Compare first two elements, then last two elements (now largest element is last), finally first two elements again.

We induct on n. Base case: trivially true for $n = 1$ and $n = 2$. Now let $n > 2$ and suppose the algorithm is correct for any array size $< n$. The proof follows from the following observations:

1. After sorting fist $\lceil 2n/3 \rceil$ elements, the largest $\lfloor n/3 \rfloor$ elements will be among the last $\lceil 2n/3 \rceil$ elements (either remain in place or moved to middle $\lfloor n/3 \rfloor$).
2. Then, after sorting the last $\lceil 2n/3 \rceil$ elements, the largest $\lfloor n/3 \rfloor$ elements will be correctly in place.
3. Finally, the whole array is sorted after again sorting fist $\lceil 2n/3 \rceil$ elements.

Remark

To be rigorous, you need to be careful/generous and use $\lceil\ \rceil$, $\lfloor\ \rfloor$.

Q2

Remark

Of course, instead of first, last, first, we can also do last, first, last; in which case it will be more conveninent to consider smallest $\lfloor n/3 \rfloor$ *elements in the proof.*

Time complexity:

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

Use master theorem, $a = 3, b = 3/2$, we are in case 1 since $d = \log_{3/2} 3 > 0$ and $f(n) \in O(1)$, so $T(n) \in \Theta(n^{\log_{3/2} 3})$.

Note that $d \approx 2.7$, so this algorithm is more like a meme rather than actually being useful.

Q3-5

Consider a $m \times n$ matrix. We say that an entry is a **peak** if its value is $\geqslant$ its (up to) four neighbours (top, right, bottom, left).

Small observation: a peak always exists (because there is a maximal entry).

First we study the given algorithm. There are two main/time-consuming components:

- Find a maximal element in C_m: $\Theta(m)$
- Recursive step: let the number of columns processed be $T(n)$. Then $T(n) = 2T(n/2) + c$. By master theorem, $a = b = 2$, so $d = 1$, we are in case 1 and $T(n) \in \Theta(n)$.

Overall, time complexity is $\Theta(mn)$.

Q3-5

The correctness is probably more tricky than you think (see the last slide).

Let M be the maximal element of C_m.

Suppose M is not a peak (otherwise we are done). Then there is some $x > M$ in the column immediately to the left or right of C_m. WLOG suppose x is on the right. Then we claim that a special peak with respect to the submatrix strictly to the right of C_m is a special peak (in A). The only case to check is when the special peak is adjacent to the boundary C_m, where the claim follows from $x > M$.

In the general case, observe that a special peak with respect to the submatrix strictly within boundaries (up to two boundaries in general) is a special peak (in A). This finishes the proof, because a special peak must be encountered when (or before) we are reduced to the case of only one column within boundaries.

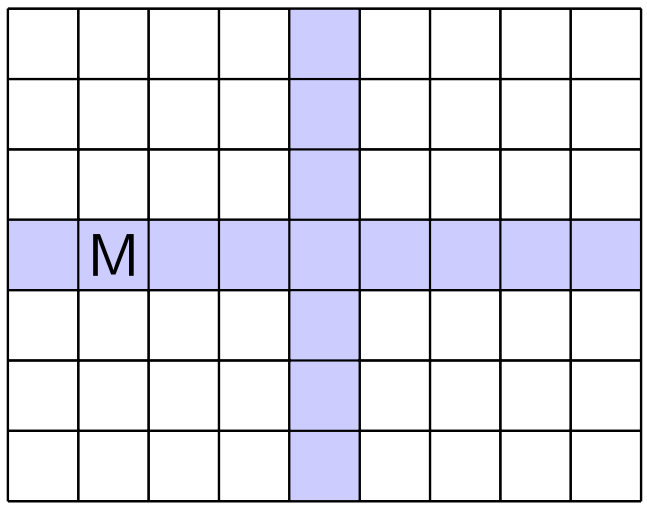
In fact, the last slide shows that we can focus our attention on just one of L_m or R_m (similar to binary search).

Hence we can actually have $T(n) = T(n/2) + c$ for number of columns. This gives $T(n) \in \Theta(\log n)$ and hence new overall complexity of $\Theta(m \log n)$.

Q3-5

If you search for this problem online, you may come across this proposed algorithm. [3]

The idea is that we add analogously a middle row R_m. We then compute the maximum M over $R_m \cup C_m$ and recurse into one of the (up to) four smaller regions where $x > M$ lies.



But is this algorithm correct?

[3] e.g. these slides from MIT back in 2011; it seems that they patched it the next semester.

Q3-5

There is a counterexample (see the picture in the same repository).