CS3230 Tutorial 3

Proof of Correctness, Divide & Conquer

(AY 25/26 Semester 2)

February 2, 2026

(Prepared by Benson & Hua Jun)

Contents

Tutorial Questions: Proof of Correctness
- Recap: Proof of Correctness
- Q1. Prove an Iterative Sorting Algorithm
- Q2. Prove a Recursive Sorting Algorithm
- Bonus. Prove 0-1 Breadth-First Search is correct

Tutorial Questions: Divide & Conquer
- Recap: Divide & Conquer
- FindPeakSp - Introduction
- Q3. There is always a peak
- Q4. Slower FindPeakSp(A)
- Q5. Faster FindPeakSp(A)
- Bonus. Linear FindPeakSp(A)

Recap: Proof of Correctness (Iterative Algorithm)

To show the correctness of an iterative algorithm, 4 things need to be shown:

1. **Loop Invariant**: A condition which is TRUE at the start of EVERY iteration.
2. **Initialization**: The invariant is true at the start of the first iteration of the loop.
3. **Maintenance**: If the invariant is true at the start of an iteration, it stays true at the end of the iteration (be careful of index values).
4. **Termination**: The invariant at the end of last iteration provides a useful property for showing correctness.

Example: Bubble Sort

Invariant: After k iterations, the largest k elements are placed correctly at `A[`$n-k$`]`, ..., `A[`$n-1$`]`.

Initial:

i	0	1	2	3	4
A[i]	7	2	9	6	4

Initialization: $k = 0$

After 1st Iteration:

i	0	1	2	3	4
A[i]	2	7	6	4	9

After 2nd Iteration:

i	0	1	2	3	4
A[i]	2	6	4	7	9

Maintenance

After 3rd Iteration:

i	0	1	2	3	4
A[i]	2	4	6	7	9

After 4th Iteration:

i	0	1	2	3	4
A[i]	2	4	6	7	9

Termination: Sorted Correctly

Recap: Proof of Correctness (Recursive Algorithm)

To show the correctness of a recursive algorithm, 2 things need to be shown:

1. **Base Case**: Show that the algorithm is correct for the base case.
   (without any recursive calls)
2. **Inductive Step**: If algorithm is correct for any input of size smaller than n, it stays correct for any input of size n.
   (with recursive calls)

Q1. Prove an Iterative Sorting Algorithm

Algorithm Insertion Sort on $A[0 \dots N-1]$

```
1: procedure InsertionSort(A)
2:     for i = 1 to N − 1 do
3:         X ← A[i]
4:         for j = i − 1 downto 0 do
5:             if A[j] > X then
6:                 A[j + 1] ← A[j]
7:             else
8:                 break
9:     A[j + 1] ← X
```

Assume the inner for loop for index j is correct.

(a) What is the suitable loop invariant for the outer for loop i?

(b) Show the invariant after initialization, maintenance, and termination.

Q1. Prove an Iterative Sorting Algorithm

Invariant: After i iterations, $A[0 \ldots i]$ is sorted in ascending order and $A[i+1 \ldots N-1]$ remains unchanged.

Intuition:

Initial:

i	0	1	2	3	4
A[i]	7	2	9	6	4

Initialization: $i = 0$

After 1st Iteration:

i	0	1	2	3	4
A[i]	2	7	9	6	4

Maintenance

After 2nd Iteration:

i	0	1	2	3	4
A[i]	2	7	9	6	4

After 3rd Iteration:

i	0	1	2	3	4
A[i]	2	6	7	9	4

After 4th Iteration:

i	0	1	2	3	4
A[i]	2	4	6	7	9

Termination: Sorted Correctly

Q1. Prove an Iterative Sorting Algorithm

Invariant: After i iterations, $A[0 \dots i]$ is sorted in ascending order and $A[i+1 \dots N-1]$ remains unchanged.

Intuition:

Initial:

i	0	1	2	3	4
A[i]	7	2	9	6	4

Initialization: $i = 0$

After 1st Iteration:

i	0	1	2	3	4
A[i]	2	7	9	6	4

Maintenance

After 2nd Iteration:

i	0	1	2	3	4
A[i]	2	7	9	6	4

After 3rd Iteration:

i	0	1	2	3	4
A[i]	2	6	7	9	4

After 4th Iteration:

i	0	1	2	3	4
A[i]	2	4	6	7	9

Termination: Sorted Correctly

Q1. Prove an Iterative Sorting Algorithm

Invariant: After i iterations, $A[0 \dots i]$ is sorted in ascending order and $A[i+1 \dots N-1]$ remains unchanged.

Step 1: Initialization

When $i = 0$, $A[0 \dots 0]$ only contains a single element and is therefore trivially sorted. $A[1 \dots N-1]$ remains unchanged.

Initial:

i	0	1	2	3	4
A[i]	7	2	9	6	4

Q1. Prove an Iterative Sorting Algorithm

Invariant: After i iterations, $A[0 \ldots i]$ is sorted in ascending order and $A[i+1 \ldots N-1]$ remains unchanged.

Step 2: Maintenance

Consider the $(i+1)$-th iteration.

To show $A[0 \ldots i+1]$ is sorted in ascending order:

- From the invariant, $A[0] \leq A[1] \leq \ldots \leq A[i]$.
- j is chosen such that $A[j] \leq X < A[j+1]$.
  $\therefore A[0] \leq \ldots \leq A[j] \leq X < A[j+1] \leq \ldots \leq A[i]$.
  $A'[0]$ $A'[j]$ $A'[j+1]$ $A'[j+2]$ $A'[i+1]$
- Therefore, $A'[0 \ldots i+1]$ is sorted.

To show $A[i+2 \ldots N-1]$ remains unchanged:

- From the invariant, $A[i+1 \ldots N-1]$ remains unchanged in the first i iterations.
- The $(i+1)$-th iteration does not change $A[i+2 \ldots N-1]$.

Algorithm Insertion Sort on $A[0 \ldots N-1]$

```
1: procedure InsertionSort(A)
2:     for i = 1 to N − 1 do
3:         X ← A[i]
4:         for j = i − 1 downto 0 do
5:             if A[j] > X then
6:                 A[j + 1] ← A[j]
7:             else
8:                 break
9:     A[j + 1] ← X
```

Q1. Prove an Iterative Sorting Algorithm

Invariant: After i iterations, $A[0 \dots i]$ is sorted in ascending order and $A[i+1 \dots N-1]$ remains unchanged.

Step 3: Termination

The invariant is true when $i = N - 1 \Rightarrow A[0 \dots N-1]$ is sorted in ascending order.

After 4th Iteration:

i	0	1	2	3	4
A[i]	2	4	6	7	9

Q2. Prove a Recursive Sorting Algorithm

Algorithm Stooge Sort on $A[0 \ldots n-1]$

```
1: procedure STOOGESORT(A)
2:     if n = 2 and A[0] > A[1] then
3:         swap A[0] and A[1]
4:     if n > 2 then
5:         ▷ Sort the first ⌈2n/3⌉ elements ◁
6:         STOOGESORT(A[0 ... ⌈2n/3⌉ − 1])
7:         ▷ Sort the last ⌈2n/3⌉ elements ◁
8:         STOOGESORT(A[n−⌈2n/3⌉, n−1])
9:         ▷ Sort the first ⌈2n/3⌉ elements ◁
10:        STOOGESORT(A[0 ... ⌈2n/3⌉ − 1])
```

(a) Prove that StoogeSort(A) correctly sorts the input array A.

(b) Analyze the time complexity of StoogeSort.

Q2a. Prove a Recursive Sorting Algorithm

Example:

i	0	1	2	3	4	5
A[i]	7	2	9	6	4	3

i	0	1	2	3	4	5
A[i]	2	6	7	9	4	3

i	0	1	2	3	4	5
A[i]	2	6	3	4	7	9

i	0	1	2	3	4	5
A[i]	2	3	4	6	7	9

Algorithm Stooge Sort on $A[0 \dots n-1]$

```
1: procedure StoogeSort(A)
2:     if n = 2 and A[0] > A[1] then
3:         swap A[0] and A[1]
4:     if n > 2 then
5:         ▷ Sort the first ⌈2n/3⌉ elements ◁
6:         StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
7:         ▷ Sort the last ⌈2n/3⌉ elements ◁
8:         StoogeSort(A[n − ⌈2n/3⌉, n − 1])
9:         ▷ Sort the first ⌈2n/3⌉ elements ◁
10:        StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
```

Q2a. Prove a Recursive Sorting Algorithm

We prove by **inducting** on the array size n.

Basis Step:

- When $n = 1$, the array is trivially sorted.
- When $n = 2$, lines 2-3 correctly sorts the array.

Induction Step:

- Suppose $n > 2$.
- Since $\lceil 2n/3 \rceil < n$, steps 6, 8 and 10 correctly sorts the sub-arrays by the induction hypothesis.
- ...
- Therefore, $A[0 \dots n-1]$ is sorted.

Algorithm Stooge Sort on $A[0 \dots n-1]$

```
1: procedure STOOGESORT(A)
2:     if n = 2 and A[0] > A[1] then
3:         swap A[0] and A[1]
4:     if n > 2 then
5:         ▷ Sort the first ⌈2n/3⌉ elements ◁
6:         STOOGESORT(A[0 ... ⌈2n/3⌉ − 1])
7:         ▷ Sort the last ⌈2n/3⌉ elements ◁
8:         STOOGESORT(A[n − ⌈2n/3⌉, n − 1])
9:         ▷ Sort the first ⌈2n/3⌉ elements ◁
10:        STOOGESORT(A[0 ... ⌈2n/3⌉ − 1])
```

Q2a. Prove a Recursive Sorting Algorithm

Intuition: Work backwards!

i	0	1	2	3	4	5
A[i]	7	2	9	6	4	3

i	0	1	2	3	4	5
A[i]	2	6	7	9	4	3

Largest $\lfloor n/3 \rfloor$ must be here

i	0	1	2	3	4	5
A[i]	2	6	3	4	7	9

Yes, Sorted.

i	0	1	2	3	4	5
A[i]	2	3	4	6	7	9

$\lceil 2n/3 \rceil$ Sorted

Largest $\lfloor n/3 \rfloor$? Sorted?

Algorithm Stooge Sort on $A[0 \dots n-1]$

1: **procedure** STOOGESORT(A)
2: **if** $n = 2$ **and** $A[0] > A[1]$ **then**
3: **swap** $A[0]$ and $A[1]$
4: **if** $n > 2$ **then**
5: ▷ *Sort the first* $\lceil 2n/3 \rceil$ *elements* ◁
6: STOOGESORT($A[0 \dots \lceil 2n/3 \rceil - 1]$)
7: ▷ *Sort the last* $\lceil 2n/3 \rceil$ *elements* ◁
8: STOOGESORT($A[n - \lceil 2n/3 \rceil, n-1]$)
9: ▷ *Sort the first* $\lceil 2n/3 \rceil$ *elements* ◁
10: STOOGESORT($A[0 \dots \lceil 2n/3 \rceil - 1]$)

Q2a. Prove a Recursive Sorting Algorithm

Claim. The largest $\lfloor n/3 \rfloor$ elements are correctly sorted.

- Take any $X = A[i]$ out of the $\lfloor n/3 \rfloor$ largest elements.
- Case 1: $i < \lceil 2n/3 \rceil$.
  - After step 6, X is within the last $\lfloor n/3 \rfloor$ elements of $A[0 \ldots \lceil 2n/3 \rceil - 1]$.
    $\Rightarrow X$ is within $A[n - \lceil 2n/3 \rceil \ldots n - 1]$.
  - After step 8, X is correctly sorted.
- Case 2: $i \geq \lceil 2n/3 \rceil$.
  - Step 6 does not change the position of X.
  - After step 8, X is correctly sorted.
- Step 10 does not change the position of X.

Algorithm Stooge Sort on $A[0 \ldots n-1]$

```
1: procedure StoogeSort(A)
2:   if n = 2 and A[0] > A[1] then
3:     swap A[0] and A[1]
4:   if n > 2 then
5:     ▷ Sort the first ⌈2n/3⌉ elements ◁
6:     StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
7:     ▷ Sort the last ⌈2n/3⌉ elements ◁
8:     StoogeSort(A[n − ⌈2n/3⌉, n − 1])
9:     ▷ Sort the first ⌈2n/3⌉ elements ◁
10:    StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
```

Q2a. Prove a Recursive Sorting Algorithm

Claim. The smallest $\lceil 2n/3 \rceil$ elements are correctly sorted.

- From the previous slide, before step 10 the largest $\lfloor n/3 \rfloor$ elements are correctly sorted. $\Rightarrow$ The smallest $\lceil 2n/3 \rceil$ elements are placed in $A[0 \dots \lceil 2n/3 \rceil - 1]$.
- After step 10, $A[0 \dots \lceil 2n/3 \rceil - 1]$ is correctly sorted.

Algorithm Stooge Sort on $A[0 \dots n-1]$

```
1:  procedure StoogeSort(A)
2:      if n = 2 and A[0] > A[1] then
3:          swap A[0] and A[1]
4:      if n > 2 then
5:          ▷ Sort the first ⌈2n/3⌉ elements ◁
6:          StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
7:          ▷ Sort the last ⌈2n/3⌉ elements ◁
8:          StoogeSort(A[n − ⌈2n/3⌉, n − 1])
9:          ▷ Sort the first ⌈2n/3⌉ elements ◁
10:         StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
```

Q2b. Time Complexity Analysis

The runtime $T(n)$ of the algorithm on an array of size n is given by the recurrence:

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 2, \\ 3T(\lceil 2n/3 \rceil) + O(1), & \text{if } n > 2. \end{cases}$$

Since $a = 3$, $b = \frac{3}{2}$, and $d = \log_{3/2} 3 \approx 2.7095\ldots$, and $f(n) \in O(n^{d-\varepsilon})$ for some $\varepsilon = 0.5 > 0$, by Case 1 of the Master Theorem, we get

$$T(n) \in O(n^d) = O(n^{2.7095\ldots}).$$

Algorithm Stooge Sort on $A[0\ldots n-1]$

```
1: procedure StoogeSort(A)
2:     if n = 2 and A[0] > A[1] then
3:         swap A[0] and A[1]
4:     if n > 2 then
5:         ▷ Sort the first ⌈2n/3⌉ elements ◁
6:         StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
7:         ▷ Sort the last ⌈2n/3⌉ elements ◁
8:         StoogeSort(A[n − ⌈2n/3⌉, n − 1])
9:         ▷ Sort the first ⌈2n/3⌉ elements ◁
10:        StoogeSort(A[0 ... ⌈2n/3⌉ − 1])
```

Bonus. Prove 0-1 Breadth-First Search is correct

Extra Slide

Consider a special case of the shortest path problem where the edge weights are either 0 or 1 only. This special case can be solved with 0-1 Breadth-First Search. Give a proof of correctness on the algorithm.

Hint (copy to view): What is the relation between 0-1 Breadth-First Search and Dijkstra's algorithm with a heap?

Algorithm 0-1 Breadth-First Search

```
 1: procedure 01BFS(G = (V, E), s)
 2:     for all x ∈ V do
 3:         dist[x] = ∞
 4:     dist[s] = 0
 5:
 6:     Q = [s]
 7:     while Q is not empty do
 8:         x = pop_front(Q)
 9:         for all neighbours of x, i.e. (x, y) ∈ E do
10:             if dist[x] + W(x, y) < dist[y] then
11:                 dist[y] = dist[x] + W(x, y)
12:                 if W(x, y) = 0 then
13:                     push_front(Q, y)
14:                 else if W(x, y) = 1 then
15:                     push_back(Q, y)
```

Bonus. Prove 0-1 Breadth-First Search is correct

Extra Slide

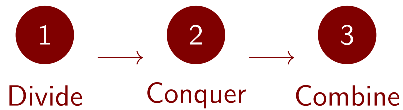
Idea: The double-ended queue Q in 0-1 Breadth-First Search works similarly to the heap in Dijkstra's algorithm. To prove the correctness of 0-1 Breadth-First Search, we only need additional invariants on the queue Q:

1. For all $v \in Q$, $\texttt{dist}[v]$ takes at most 2 (consecutive) distinct values.
2. For all $u, v \in Q$, if u is placed before v, then $\texttt{dist}[u] \leq \texttt{dist}[v]$.

Recap: Divide & Conquer

1. **Divide**: Divide the problem into smaller subproblems.
2. **Conquer**: Solve the subproblems recursively.
3. **Combine**: Use the subproblem solutions to get the solution of the full problem.

Recap: Divide & Conquer



Example: Tiling L-pieces

FindPeakSp - Introduction

Problem. Suppose we are given a 2D-array of size *m* rows by *n* columns. An element in the array $A[i][j]$ is called a "peak" if it is **greater than or equal to** its adjacent neighbours. You want to find any single peak.

6	8	7	7	1
9	3	1	7	3
8	4	5	3	2

Q3. There is always a peak

Show that there is a peak in every array.

Q3. There is always a peak

Observations:

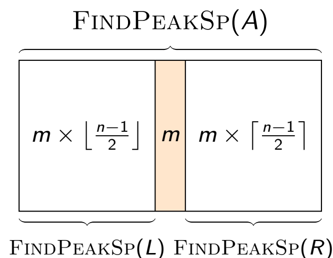
1. The **largest number** on the whole array A must be **a peak**.
   - Proof: That largest number must be greater or equal than all of its neighbours.

6	8	7	7	1
9	3	1	7	3
8	4	5	3	2

Q4. Slower FindPeakSp(A)

Algorithm Slower FindPeakSp(A)

```
1:  procedure FindPeakSp(A)
2:      if A is empty then
3:          return none
4:      else if A has n = 1 column then
5:          return the maximum element on that column
6:      else if A has n ≥ 2 columns then
7:          find the maximum element on that middle column
8:          if that element is a peak then
9:              return that element
10:         else
11:             X = FindPeak(Left_Half_Of_A)
12:             Y = FindPeak(Right_Half_Of_A)
13:             if X or Y is a peak then
14:                 return that peak
15:             else
16:                 return none
```



Q4. Slower FINDPEAKSP(A)

Algorithm Slower FINDPEAKSP(A)

```
1:  procedure FINDPEAKSP(A)
2:      if A is empty then
3:          return none
4:      else if A has n = 1 column then
5:          return the maximum element on that column
6:      else if A has n ≥ 2 columns then
7:          find the maximum element on that middle column
8:          if that element is a peak then
9:              return that element
10:         else
11:             X = FINDPEAK(Left_Half_Of_A)
12:             Y = FINDPEAK(Right_Half_Of_A)
13:             if X or Y is a peak then
14:                 return that peak
15:             else
16:                 return none
```

What is the time complexity of this algorithm?

Q4. Slower FindPeakSp(A)

Algorithm Slower FindPeakSp(A)

```
1: procedure FindPeakSp(A)
2:     if A is empty then
3:         return none
4:     else if A has n = 1 column then
5:         return the maximum element on that column
6:     else if A has n ≥ 2 columns then
7:         find the maximum element on that middle column
8:         if that element is a peak then
9:             return that element
10:        else
11:            X = FindPeak(Left_Half_Of_A)
12:            Y = FindPeak(Right_Half_Of_A)
13:            if X or Y is a peak then
14:                return that peak
15:            else
16:                return none
```

What is the time complexity of this algorithm?

$$T(m, n) = 2T\left(m, \frac{n}{2}\right) + \Theta(m)$$

Let $S(n)$ be the number of columns processed, then

$$S(n) = 2S\left(\frac{n}{2}\right) + 1$$

By Master Theorem (Case 1), we get $S(n) = \Theta(n)$.
Thus, $T(m, n) = \Theta(mn)$.

Q5. Faster FINDPEAKSP(A)

Observations:

2. Pick any column, say column k. Pick the largest number from the column k. If this number is not a peak because **its right neighbour** (on column $k+1$) **is larger**, then **there must be a peak** located on the **columns** $[k+1, n]$ (at column $k+1, k+2, \ldots, n$).

7	2	7	10	1
5	6	8	9	3
8	4	5	3	2
	k	$k+1$	$k+2$	

Question. Prove Observation 2.

Q5. Faster FindPeakSp(A)

Idea. The largest number among columns $[k+1, n]$ must be a peak.

- If the largest number among columns $[k+1, n]$ is located at columns $[k+2, n]$, then it must be a peak by **Observation 1**.

7	2	7	10	1
5	6	8	9	3
8	4	5	3	2

- What if the largest number among columns $[k+1, n]$ is located at column $k+1$?

Q5. Faster FINDPEAKSP(A)

Edge case: The largest number among columns $[k+1, n]$ is located at column $k+1$.

k	$k+1$
W	X
Y	Z

Consider columns k and $k+1$.

- Suppose that the largest number on column k is W.
- Then we know that $X > W$, from the original assumption of the algorithm.
- If X is the largest number on column $k+1$, then X must be a peak (because it is also the largest among columns $> k$).
- If X is not the largest, suppose Z is the largest.
- Consider Y. We have $Z \geq X > W \geq Y$.
- Therefore, Z must be a peak.

Q5. Faster FINDPEAKSP(A)

Observations:

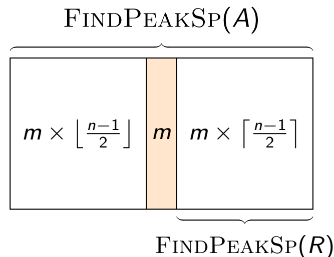
2. Pick any column, say column k. Pick the largest number from the column k. If this number is not a peak because **its right neighbour** (on column $k+1$) **is larger**, then **there must be a peak** located on the **columns** $[k+1, n]$ (at column $k+1, k+2, \ldots, n$).
3. Symmetrically, the case where the **left neighbour** is larger indicates that **there must be a peak** located on the **columns** $[1, k-1]$ (at column $1, 2, \ldots, k-1$)

Design a faster algorithm to find the peak using this observation!

Q5. Faster FindPeakSp(A)

Algorithm Faster FindPeakSp(A)

```
1: procedure FindPeakSp(A)
2:     if A is empty then
3:         return none
4:     else if A only has 1 column then
5:         return the maximum element on that column
6:     else if A has ≥ 2 columns then
7:         find the maximum element on that middle column
8:         if that element is a peak then
9:             return that element
10:        else
11:            if right neighbour > left neighbour then
12:                return FindPeak(Right_Half_Of_A)
13:            else
14:                return FindPeak(Left_Half_Of_A)
```



Q5. Faster FindPeakSp(A)

Algorithm Faster FindPeakSp(A)

```
 1: procedure FindPeakSp(A)
 2:     if A is empty then
 3:         return none
 4:     else if A only has 1 column then
 5:         return the maximum element on that column
 6:     else if A has ≥ 2 columns then
 7:         find the maximum element on that middle column
 8:         if that element is a peak then
 9:             return that element
10:         else
11:             if right neighbour > left neighbour then
12:                 return FindPeak(Right_Half_Of_A)
13:             else
14:                 return FindPeak(Left_Half_Of_A)
```

What is the time complexity of this algorithm?

$$T(m,n) = T\left(m, \frac{n}{2}\right) + \Theta(m)$$

Let $S(n)$ be the number of columns processed, then

$$S(n) = S\left(\frac{n}{2}\right) + 1$$

By Master Theorem (Case 2), we get $S(n) = \Theta(\log n)$. Thus, $T(m,n) = \Theta(m \log n)$.

Bonus. Linear FindPeakSp(A)

Extra Slide

In the tutorial we have seen an algorithm for FindPeakSp(A) with running time $O(m \log n)$. Can you modify that algorithm to achieve running time $O(m + n)$?

Hint (copy to view): Reduce the problem from size $m \times n$ to size $m/2 \times n/2$.

Bonus. Linear FINDPEAKSP(A)

Extra Slide

Idea: Define *crosshair* of the 2D array to be subset of the array that contains the middle row, middle column, and the border of the array. Building on our solution in tutorial, finding the maximum element in the crosshair would allow us to cut the search columns and rows by half.