

CS3230 Semester 1 2025/2026
Design and Analysis of Algorithms

Tutorial 03
Proof of Correctness and D&C (1)
For Week 04

Document is last modified on: August 29, 2025

1 Lecture Review: Proof of Correctness

We prove the correctness of an algorithm depending on its type:

- For iterative algorithm, we usually use loop invariant.
Invariant is a condition which is TRUE at the start of EVERY iteration
We can then use invariant to show the correctness:
 1. Initialization: It is true before iteration 1
 2. Maintenance: If it is true for iteration x , it remains true for iteration $x+1$
 3. Termination: When the algorithm ends, it helps the proof of correctness
- For recursive algorithm, we usually use proof by induction.
 1. Show the recursive algorithm is (trivially) correct on its base case(s).
 2. Inductive step: show that the recursive algorithm is correct, assuming that the smaller cases are all correct.

2 Lecture Review: D&C

Here are the usual steps for using Divide and Conquer (D&C) problem solving paradigm for problems that are amenable to it:

1. **Divide:** Divide/break the original problem into ≥ 1 smaller sub-problems.
2. **Conquer:** Conquer/solve the sub-problems recursively.
3. **Combine** (optional): Optionally, combine the sub-problem solutions to get the solution of the original problem.

The most classic D&C example is **Merge Sort**.

1. **Divide:** Divide/break the original problem of sorting n elements into 2 smaller sub-problems of sorting $\frac{n}{2}$ elements.
2. **Conquer:** Conquer/solve the sorting of $\frac{n}{2}$ elements recursively.
3. **Combine** (optional): Merge 2 already sorted $\frac{n}{2}$ elements.

3 Tutorial 03 Questions

Q1). Consider the following iterative sorting algorithm:

Algorithm 1: InsertionSort($A[0..N - 1]$)

```
1 for  $i = 1$  to  $N - 1$  do // outer For loop  $i$ 
2   Let  $X = A[i]$  //  $X$  is the next item to insert into  $A[0..i - 1]$ 
3   for  $j = i - 1$  down to 0 do // inner For loop  $j$ 
4     if  $A[j] > X$  then
5       |  $A[j + 1] = A[j]$  // Make space for  $X$ 
6     else
7       | break
8    $A[j + 1] = X$  // Insert  $X$  at index  $j + 1$ 
```

Assuming the inner For loop for index j is correct (that is, assuming, $A[0..i - 1]$ is sorted it places $A[i]$ in its correct position, without making any other changes to $A[i + 1..N - 1]$) answer these questions:

- (a) What is the suitable loop invariant for the outer For loop i ?
- (b) Show the invariant after initialization, maintenance, and termination.

Not part of the tutorial,

but you may want to think about a suitable loop invariant for the inner For loop j .

Q2). Consider the following recursive sorting algorithm:

Algorithm 2: StoogeSort(A)

```
1 Let  $n$  be the length of array  $A$ 
2 if  $n = 2$  and  $A[0] > A[1]$  then
3   | Swap  $A[0]$  and  $A[1]$ 
4 if  $n > 2$  then
5   | Apply StoogeSort to sort the first  $\lceil 2n/3 \rceil$  elements recursively
6   | Apply StoogeSort to sort the last  $\lceil 2n/3 \rceil$  elements recursively
7   | Apply StoogeSort to sort the first  $\lceil 2n/3 \rceil$  elements recursively
```

Answer the following two questions:

- (a) Prove that $\text{StoogeSort}(A)$ correctly sorts the input array A .
For the sake of simplicity, you may assume that all numbers in A are distinct.
- (b) Analyze the time complexity of **StoogeSort**.

Q3, Q4, Q5. involves Finding a Peak Problem (see this week chosen LeetCode problems)

Given a 2D array with m rows and n columns, where each cell contains a number, a **peak** is a cell whose value is no smaller than all of its (up to) four neighbors: top, right, bottom, and left.

For example, given $m \times n = 3 \times 5$ grid below, there are 5 peaks (denoted with a '*'):

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

Q3). Show that there is a peak in every 2D array!

We want to come up with a recursive algorithm to find any peak:

Algorithm 3: FindPeakSp(A)

```
1 if  $A$  has  $n = 1$  column then
2   | return a maximal element in the column
3 if  $A$  has  $n \geq 2$  columns then
4   | Let  $C_m$  be the middle column of  $A$ 
5   | Find a maximal element in  $C_m$ 
6   | if the above maximal element in  $C_m$  is a peak then
7   |   | return that element
8   | else
9   |   |  $X \leftarrow \text{FindPeakSp}(\text{Left\_Half\_of\_A\_without\_}C_m)$ 
10  |   |  $Y \leftarrow \text{FindPeakSp}(\text{Right\_Half\_of\_A\_without\_}C_m)$ 
11  |   | if  $X$  or  $Y$  is a peak then
12  |   |   | return the peak ( $X$  or  $Y$ )
13  |   | else
14  |   |   | return None                                     // See Question Q3
```

Note: FindPeakSp finds a **S**pecial kind of peak element. The element that is a peak as well a maximal element in the column in which it is located. Call this kind of peak element special-peak.

Q4). What is the runtime complexity of FindPeakSp(A) algorithm?

Q5). Argue why FindPeakSp(A) will never return None (i.e., always returns a peak). Additionally, discuss whether any steps within the ‘else’ condition in Step 8 can be optimized (faster asymptotically). Note that we assume that the entirety of 2D array A with m rows and n columns have been loaded into memory, i.e., we ignore the $\Theta(mn)$ upfront cost of reading this 2D array.

Optional Q6) - time permitting. Discuss the following LeetCode task during tutorial:

TA can choose to discuss in high-level only or show the partial/full code (in C++/Python/Java)

- Wednesday class: find-the-peaks
(simple complete search; solve this trivial task, and argue that you cannot do better than $\Omega(n)$ and $\Theta(n)$ algorithm is an optimal algorithm for this task),
- Thursday class: find-peak-element
(DnC, 1D; ignore the $\Theta(n)$ I/O cost to load the 1D array $nums$),
- Friday class: find-a-peak-element-ii
(DnC, 2D, tut03; ignore the $\Theta(mn)$ I/O cost to load the 2D array mat).