CS3230 Semester 2 2025/2026
Design and Analysis of Algorithms

Tutorial 03
Correctness and Divide-and-conquer
For Week 04

Document is last modified on: January 28, 2026

1 Lecture Review: Proof of Correctness

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

- For iterative algorithms, we usually use a loop invariant.
  An invariant is a condition that is TRUE at the start of EVERY iteration.
  We can then use the 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 algorithms, we usually use a 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 the Divide and Conquer (D&C) problem solving paradigm for problems that are amenable to it:

1. **Divide**: Divide/break the original problem into $\underline{\geq 1 \text{ smaller}}$ 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 lists of $\frac{n}{2}$ elements.

3 Tutorial 03 Questions

Q1). Consider the following iterative sorting algorithm:

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

```
for i = 1 to N − 1 do                                  // outer for-loop i
    Let X = A[i]                  // X is the next item to insert into A[0..i − 1]
    for j = i − 1 down to 0 do                         // inner for-loop j
        if A[j] > X then
            A[j + 1] = A[j]                            // Make space for X
        else
            break
    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 the following two 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 invariant for the inner for-loop.

Q2). Consider the following recursive sorting algorithm:

Algorithm 2: StoogeSort(A)

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

Answer the following two questions:

(a) Prove that 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`.

The Peak Finding Problem (Q3-5)

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 <u>any</u> 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 ≥ 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 ← FindPeakSp(Left_Half_of_A_without_C_m)
10         Y ← FindPeakSp(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 Q5
```

Note: FindPeakSp finds a **Sp**ecial 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).