CS3230 Semester 2 2024/2025
Design and Analysis of Algorithms

Tutorial 04
Correctness and Divide-and-conquer
For Week 05

Document is last modified on: January 29, 2025

MODEL ANSWER IS FOR OUR CLASS ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

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 04 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?

Let B refer to the original (unsorted) array A (alternatively, you can imagine having copied original array A to B at the beginning). This makes it easier to refer to the original values.

Invariant: $A[0..i-1]$ is the sorted version of $B[0..i-1]$. Furthermore, $A[i..N-1] = B[i..N-1]$

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

The invariant is true at the beginning when $i = 1$,
i.e., $A[0] = B[0]$ is a single Integer and by default is sorted.
The rest of the array is as $A[1..N-1] = B[1..N-1]$

As we have been given the assumption that the inner for loop j is correct, after it terminates (break) and we reach Step 8, we will correctly slot X at $A[j+1]$, maintaining $A[0..i]$ is now the sorted values of $B[0..i]$ (one index more than before).

At termination, $i = N-1$, then the invariant says that $A[0..N-1]$ is the sorted values of the original array $B[0..N-1]$, which shows the correctness of InsertionSort(A).

Not part of the tutorial, but you may want to think about a suitable invariant for inner For loop.

Invariant: The following hold
(i) $A[0..j]A[j+2..i]$ is the sorted version of $B[0..i-1]$.
(ii) $A[i+1..N-1] = B[i+1..N-1]$.
(iii) $X = B[i]$.
(iv) If $j+2 \leq i$, then $A[j+2] > X$

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 ⌈2n/3⌉ elements recursively
6     Apply StoogeSort to sort the last ⌈2n/3⌉ elements recursively
7     Apply StoogeSort to sort the first ⌈2n/3⌉ 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.

We prove the correctness of the algorithm by an induction on the array size n.

Base case: If $n = 1$, the algorithm is trivially correct, as the array is already sorted.
If $n = 2$, the algorithm is correct due to Step 2.

Inductive step: Now consider the case of $n > 2$. By induction hypothesis, assume that the algorithm is correct on any array of size smaller than n. Let $r = n - \lceil 2n/3 \rceil = \lfloor n/3 \rfloor$. We make the following observation:

- After Step 5, the r largest numbers of A must be in the final $\lceil 2n/3 \rceil$ entries of A.

This observation implies that the r largest numbers of A are correctly sorted after Step 6. Therefore, at the beginning of Step 7, the initial $n-r = \lceil 2n/3 \rceil$ numbers of the array are precisely the $\lceil 2n/3 \rceil$ smallest numbers of A. After Step 7, these $\lceil 2n/3 \rceil$ numbers are also correctly sorted.

In the subsequent discussion, we prove the above observation. Let x be any number in the set of r largest numbers of A. We show that x must be in the final $\lceil 2n/3 \rceil$ entries of A after Step 5.

- Suppose x is not one of the initial $\lceil 2n/3 \rceil$ numbers of A at the beginning of Step 5. The algorithm of Step 5 does not change the position of x, so x is still in the final $n - \lceil 2n/3 \rceil \leq \lceil 2n/3 \rceil$ entries of A after Step 5.
- Suppose x is one of the initial $\lceil 2n/3 \rceil$ numbers of A at the beginning of Step 5. Among these $\lceil 2n/3 \rceil$ numbers, at least $\lceil 2n/3 \rceil - r \geq r$ of them are smaller than x. Therefore, after Step 5, x is not in the initial r entries of A. In other words, x is in the final $n - r = \lceil 2n/3 \rceil$ entries of A after Step 5.

(b) Analyze the time complexity of `StoogeSort`.

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

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

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

Optional: Can ask students why the choice of $\lceil 2n/3 \rceil$ makes sense in the algorithm.

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!

Since any 2D array must contain at least one maximal element, and a maximal element is no smaller than any other cell (including its four neighbors), all maximal elements are peaks.

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 Q3
```

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?

Time complexity of finding a maximal element in any column is $\Theta(m)$, as there are m rows.
So, we can consider how many columns are processed, then multiply the result by $\Theta(m)$.
Let $T(n)$ be the number of columns to be processed, then $T(n) = 2 \cdot T(\frac{n}{2}) + 1$.
Since $a = 2, b = 2, d = \log_2 2 = 1$, and $f(n) \in O(n^{d-\epsilon})$ for some $0.5 = \epsilon > 0$, by Case 1 of the Master Theorem, then $T(n) \in \Theta(n^d) = \Theta(n^{\log_2 2}) = \Theta(n)$.
Thus, FindPeakSp(A) runs in $T(n) \times \Theta(m) = \Theta(n) \times \Theta(m) \in \Theta(nm)$.

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).

The following argument demonstrates why the algorithm will always find a peak (special-peak) and thus never return None. It also explains why FindPeakSp(A) does not need to perform both Steps 9 and 10. Consequently, a faster divide-and-conquer (D&C) algorithm can be designed.

If we reach Step 8, the chosen maximal element W in the middle column (the k-th column) is not a peak. This implies one of the following scenarios for W:

- Only right neighbor of W is larger.
- Only left neighbor of W is larger. (Symmetric to above)
- Both the left and right neighbors of W are larger. (Covered by the two cases above)

Hence, we focus on the case where the right neighbor of W in column $k+1$ (denoted as X) is larger.

$$
\begin{array}{c c}
 & \begin{array}{ccc|c||c|cc} 1 & & \cdots & k & k+1 & \cdots & n \end{array} \\
\begin{array}{c} 1 \\ \\ \\ \vdots \\ \\ \\ m \end{array} &
\left[\begin{array}{ccc|c||c|cc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & a & b & W & X & c & \cdots \\
\cdots & d & e & f & g & h & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & l & o & p & q & r & \cdots \\
\cdots & s & t & Y & Z & u & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{array}\right]
\end{array}
$$

Figure 1: Illustration of the scenario where the right neighbor X in column $k+1$ is larger than W. The figure highlights the relevant elements W (max in C_m), X, Y, and Z (special-peak of A').

We argue below that this guarantees the existence of a special-peak in the columns $k+1, k+2, \ldots$ (i.e., columns $> k$). Refer to Figure 1 for an illustration of this scenario.

A special-peak in the right subarray $A' = A[1..m][k+1..n]$ must also be a special-peak of A if it is located in any column other than column $k+1$. Thus, the only case requiring further consideration is when a special-peak of A' is located in column $k+1$, as it directly borders column k.

Let Z be a special-peak of A' located in column $k+1$ of A. Observe the following:

- Z is a maximal element in column $k+1$ of A, so $Z \geq X$, where X is the right neighbor of W.
- Z is not smaller than any of its neighbors in A', i.e., it is not smaller than its top, bottom, or right neighbors. To confirm that Z is also a special-peak of A, we need to show that Z is not smaller than its left neighbor Y in column k.

Since the right neighbor of W is larger $(X > W)$ and $Z \geq X$, it follows that:

$$Z \geq X > W \geq Y.$$

This implies that Z is not smaller than its left neighbor Y. Therefore, Z is a special-peak of A.

With this, we can optimize the 'else' condition in Step 8, as shown below in the **Imp**roved algorithm:

Algorithm 4: FindPeakSp-Imp(A)

```
if A has n = 1 column then
    return a maximal element in the column
if A has n ≥ 2 columns then
    Let C_m be the middle column of A
    Find a maximal element in C_m
    if the above maximal element in C_m is a peak then
        return that element
    else
        if the right neighbor of the above maximal element in C_m is larger then
            return FindPeakSp-Imp(Right_Half_of_A_without_C_m)
        else
            return FindPeakSp-Imp(Left_Half_of_A_without_C_m)
```

Now we analyze its asymptotic behavior.

Let $T(n)$ represent the number of columns processed. In this case, the recurrence is: $T(n) = T(n/2)+1$. Since $a = 1$, $b = 2$, $d = 0$, and $f(n) \in \Theta(n^d)$, by Case 2 of the Master Theorem, yielding: $T(n) \in \Theta(\log n)$. Thus, the overall algorithm runs in $T(n) \times \Theta(m) = \Theta(\log n) \times \Theta(m) \in \Theta(m \log n)$, which is asymptotically faster.

Optional: Can ask the students whether the $\Theta(m \log n)$ algorithm is the best possible solution for this problem.