NUS | Computing
National University of Singapore

CS3230
`eric_han@nus.edu.sg`
`https://eric-han.com`
Computer Science

T04 – Week 5

Correctness and Divide-and-conquer

CS3230 – Design and Analysis of Algorithms

Admin

Assignment 2 scores with comments are published, can be found on Canvas.

- Comments have been given
- Any queries please approach me (after class or on telegram)

Assignment 3 due this weekend.

General comments

- A2 Q2: on proving $f(n) = 3f(n/3) + n$ – closed form $f(n) = n \log_3 n$.
  - Cannot use MT here. We asked for exact, not a bound.
  - Must use any techniques that give exact closed form.

Lecture Review

Proof of Correctness

- **Iterative Algorithm**: Prove with a loop invariant:
  1. **Initialization**: True before iteration 1.
  2. **Maintenance**: True for iteration $x \implies$ true for $x + 1$.
  3. **Termination**: Ensures correctness at the end.
- **Recursive Algorithm**: Prove by induction:
  1. **Base Case**: Correct for trivial cases.
  2. **Inductive Step**: Assume smaller cases are correct, prove for current case.

Divide-and-conquer (D&C)

1. **Divide**: Break the problem into smaller sub-problems.
2. **Conquer**: Solve sub-problems recursively.
3. **Combine** (optional): Merge sub-problem solutions.

E.g., **Merge Sort**: Split into halves, sort recursively, merge results.

Question 1 [G]/[P1]

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 $j+1$
```

Recap

- What is the intuition behind insertion sort?
- What is a good/bad invariant?

Assuming the inner loop for index j is correct (i.e., assuming $A[0..i-1]$ is sorted and places $A[i]$ in its correct position without affecting $A[i+1..N-1]$):

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

Question 1 Optional

What is a suitable invariant for the inner for loop?

Answer 1a

Let B represent the original (unsorted) array A (or imagine copying A into B at the beginning). This allows us to reference the original values more easily.

Outer Loop Invariant

1. $A[0..i-1]$ is the sorted version of $B[0..i-1]$.
2. $A[i..N-1] = B[i..N-1]$ (the portion of the array from i to $N-1$ remains unchanged and matches the original values in B).

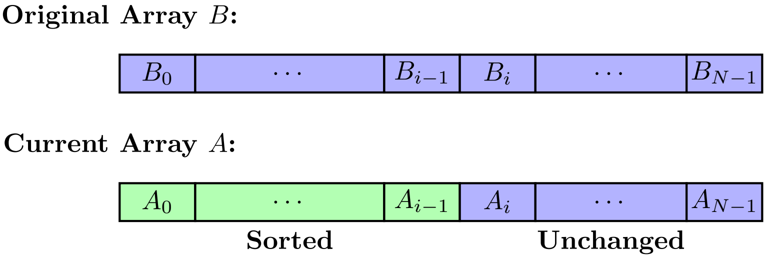

Figure 1: Illustration of outer loop invariant.

Answer 1b

1. **Initialization**:
    - When $i = 1$, $A[0] = B[0]$ is a single integer and sorted by default.
    - The rest of the array remains unchanged: $A[1..N-1] = B[1..N-1]$.
2. **Maintenance**:
    - Assuming the invariant holds at the start of iteration i, we have:
        - $A[0..i-1]$ is sorted $B[0..i-1]$.
        - $A[i..N-1] = B[i..N-1]$ is not sorted.
    - After the inner loop places X at its correct position, without affecting $A[i+1..N-1]$.
    - This ensures $A[0..i]$ is the sorted version of $B[0..i]$.
3. **Termination**:
    - At $i = N-1$, the invariant guarantees $A[0..N-1]$ is the sorted version of $B[0..N-1]$, proving the algorithm's correctness.

Question 2

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
```

a. Prove that $\mathrm{StoogeSort}(A)$ 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 $\mathrm{StoogeSort}(A)$.

Answer 2a

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 since the array is already sorted.
- If $n = 2$, the algorithm is correct due to Step 2.

Inductive Step

Assume the algorithm is correct for any array of size smaller than n.

Observation: Let $r = n - \lceil 2n/3 \rceil = \lfloor n/3 \rfloor$. After Step 5: - The r largest numbers of A are in the final $\lceil 2n/3 \rceil$ entries of A; Then:

1. After Step 6, the r largest numbers of A are correctly sorted.
2. Before Step 7, the initial $n - r$ numbers are the $\lceil 2n/3 \rceil$ first entries of A.
3. After Step 7, these $\lceil 2n/3 \rceil$ numbers are also correctly sorted.

Proof of 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:

- **Case 1:** Suppose x is not one of the initial $\lceil 2n/3 \rceil$ numbers of A before 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.
- **Case 2:** Suppose x is one of the initial $\lceil 2n/3 \rceil$ numbers of A before 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.

Hence, by induction, the algorithm is correct.

Answer 2b

The runtime $T(n)$ of the algorithm for 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 $a = 3$, $b = 3/2$, 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:

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

Answer 2b

The runtime $T(n)$ of the algorithm for 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 $a = 3$, $b = 3/2$, 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:

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

Question 2 Optional

Why does choosing $\lceil 2n/3 \rceil$ in the algorithm make sense?

The Peak Finding Problem

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.

Example

In the $m \times n = 3 \times 5$ grid below, there are 5 peaks (marked with `*`):

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

Question 3 [G]/[P2]

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

Question 3 [G]/[P2]

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

Answer

- 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.

Find Peak Algorithm

We aim to design a recursive algorithm $\mathrm{FindPeakSp}$ to find *any* peak.

Special-Peak Definition

Note: $\mathrm{FindPeakSp}$ finds a **Sp**ecial kind of peak element. This element is both a peak and the maximal element in its column. We refer to this as a **special-peak**.

Algorithm 3: FindPeakSp(A)

```
if $A$ has $n = 1$ column then
    return a maximal element in the column
if $A$ has $n \geq 2$ 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
        $X \leftarrow \mathrm{FindPeakSp}($Left_Half_of_A_without_$C_m)$
        $Y \leftarrow \mathrm{FindPeakSp}($Right_Half_of_A_without_$C_m)$
        if $X$ or $Y$ is a peak then
            return the peak ($X$ or $Y$)
        else
            return None                                    // See Question Q3
```

Question 4 [P3]

What is the runtime complexity of the $\text{FindPeakSp}(A)$ algorithm?

Question 4 [P3]

What is the runtime complexity of the $\mathrm{FindPeakSp}(A)$ algorithm?

Answer

Finding the maximal element in a column takes $\Theta(m)$ (as there are m rows). The total complexity depends on *how many columns are processed*, scaled by $\Theta(m)$.

Column Processing Complexity

Let $T(n)$ represent the number of columns to be processed:

$$T(m,n) = 2 \cdot T(m, n/2) + k \cdot m \implies T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 1$$

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 $T(n) \in \Theta(n^d) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Overall Runtime

$$T(n) \times \Theta(m) = \Theta(n) \times \Theta(m) = \Theta(nm)$$

Question 5 [G]

Argue why $\mathrm{FindPeakSp}(A)$ 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).

Question 5 [G]

Argue why $\mathrm{FindPeakSp}(A)$ 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).

Answer

The argument shows that Steps 9 and 10 can be skipped, optimizing our algorithm.

Never Return None $\iff$ Special-Peak Exists

If Step 8 is reached, the maximal element W in column k is not a peak, then:

- Only the right neighbor of W is larger.
- Only the left neighbor of W is larger (symmetric).
- Both the left and right neighbors of W are larger (covered by cases above).

We focus on the case where W's right neighbor X (in column $k+1$) is larger. This ensures a special-peak exists in columns $> k$.

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

Figure 2: Illustration of the right neighbor scenario – Subarray $A' = A[1..m][k+1..n]$.

Right Neighbor of W is Larger

- Then, $X > W$.

Special-Peak in A'

If special-peak in column:

- $> k+1$: Its a special-peak of A.
- $k+1$: Adjacent to column k:
  - Z is the max in column $k+1$,
  - So $Z \geq X$: X is right of W.
  - Z is not smaller than its top, bottom, or right neighbors in A'.
  - We must check the left neighbor:
    - Show $Z \geq Y$ in column k:
    - Since $X > W$ and $Z \geq X$,
    - $Z \geq X > W \geq Y$.
  - Z is a special-peak of A.

Thus, Z is a special-peak of A.

Steps to optimize

How does this translate to optimizing (**imp**roving) the `else` condition in Step 8?

Steps to optimize

How does this translate to optimizing (**imp**roving) the `else` condition in Step 8?

Algorithm 5: FindPeakSp-Imp(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          if the right neighbor of the above maximal element in $C_m$ is larger then
10             return FindPeakSp-Imp(Right_Half_of_A_without_$C_m$)
11         else
12             return FindPeakSp-Imp(Left_Half_of_A_without_$C_m$)
```

Asymptotic Behavior

Let $T(n)$ be the number of columns processed, with the recurrence:

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

Since $a = 1$, $b = 2$, $d = 0$, and $f(n) \in \Theta(n^d)$, by Case 2 of the Master Theorem:

$$T(n) \in \Theta(\log n).$$

Thus, the algorithm runs in:

$$T(n) \times \Theta(m) = \Theta(\log n) \times \Theta(m) = \Theta(m \log n),$$

which is asymptotically faster.

Asymptotic Behavior

Let $T(n)$ be the number of columns processed, with the recurrence:

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

Since $a = 1$, $b = 2$, $d = 0$, and $f(n) \in \Theta(n^d)$, by Case 2 of the Master Theorem:

$$T(n) \in \Theta(\log n).$$

Thus, the algorithm runs in:

$$T(n) \times \Theta(m) = \Theta(\log n) \times \Theta(m) = \Theta(m \log n),$$

which is asymptotically faster.

Question 5 Optional [Snack]

Is this $\Theta(m \log n)$ algorithm the best possible solution?

Asymptotic Behavior

Let $T(n)$ be the number of columns processed, with the recurrence:

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

Since $a = 1$, $b = 2$, $d = 0$, and $f(n) \in \Theta(n^d)$, by Case 2 of the Master Theorem:

$$T(n) \in \Theta(\log n).$$

Thus, the algorithm runs in:

$$T(n) \times \Theta(m) = \Theta(\log n) \times \Theta(m) = \Theta(m \log n),$$

which is asymptotically faster.

Question 5 Optional [Snack]

Is this $\Theta(m \log n)$ algorithm the best possible solution? We can achieve $\Theta(n)$ - How?

Practical [Optional]

Practical repo: To help you further your understanding, not compulsory; Work for Snack!

1. Implement Algorithm 3 in code, `find_peak_sp` to return a special peak.
2. Check that you get this output:

```
...
Test 6: Matrix
6  8* 7  7* 1
9* 3  1  7* 3
8  4  5* 3  2
 Peak found at (0, 1) with value 8
```