SCS1308 Foundations of Algorithms

Tutorial Session - 04

Assignment Question Discussion

1.

Consider the following equations when considering masters theorem.

T(n) is a monotonically increasing function as follows:

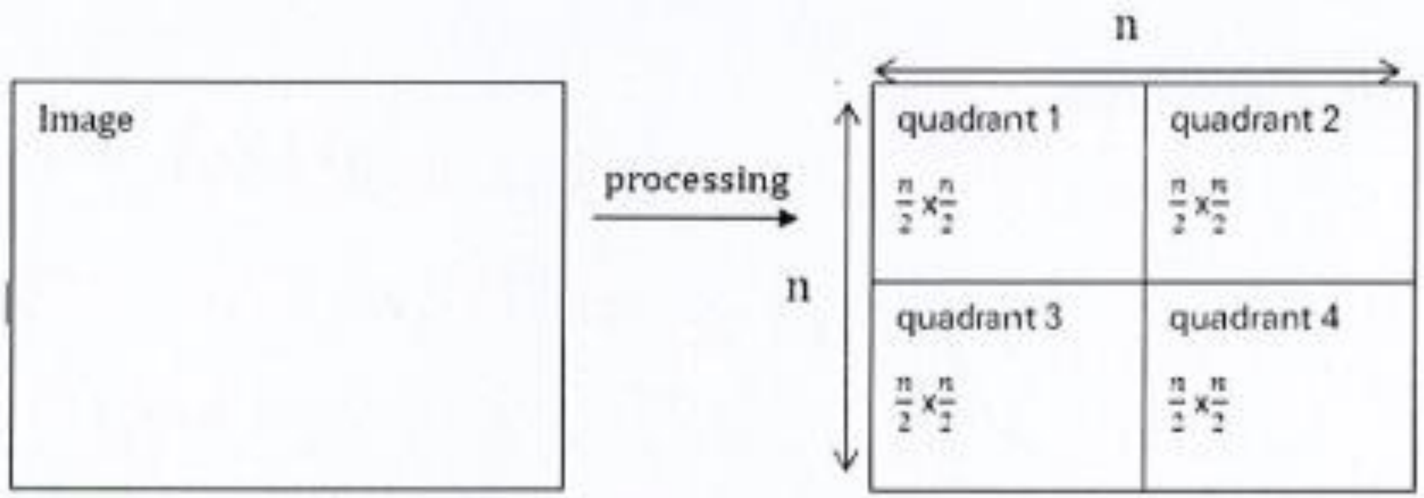
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = c$$

Where $a \geq 1, b \geq 2, c > 0, if\ f(n) is\ \Theta\left(n^{d}\right) where\ d \geq 0\ then,$

$$T(n) = \begin{cases} \Theta(n^{d}) \text{ if } a < b^{d} \\ \Theta(n^{d} \log n) \text{ if } a = b^{d} \\ \Theta(n^{\log_{b}^{a}}) \text{ if } a > b^{d} \end{cases}$$

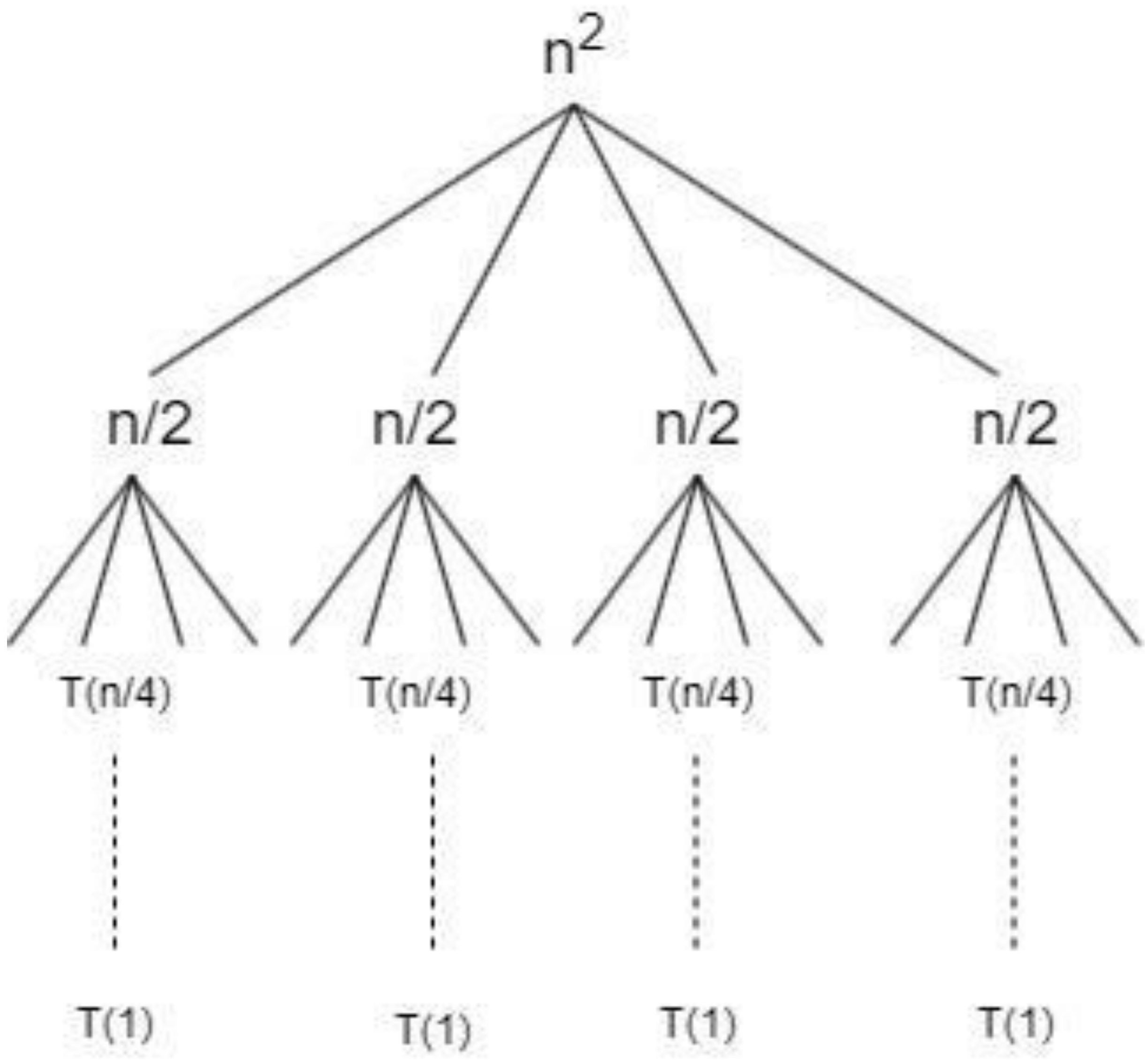
1. High resolution image is divided into 4 quadrants for processing. Each quadrant is processed recursively, and merging takes n^2 time due to pixel blending. Consider the dimensions of an image .



A. Write the recurrence equation for the above scenario considering recursive and non-recursive terms. Your final answer should be given in T(n) terms.

$$T(n) = 4T(n/2) + n^2$$

B. Solve the recurrence equation using iteration method. Note that tree structure including root, depth and how leaves in some levels formation required to obtain full marks.



C. Verify solution using substitution method.

Make educated guesses,

$T(u)$ grows faster than $O(n^2)$ due to $4T(n/2)$ but slower than $O(n^3)$.

prove by induction

$$T(n) \leq cn^2 \log n, \text{ for } c \geq 0.$$

base case : $n = 1$, $T(1) = d$, $d \leq c.\log(1) = 0$ holds.

Inductive Hypothesis :

recurrence hold $k < n$;

$$T(k) \leq ck^2 \log k \text{ for } k < n$$

Inductive Step,

$$T(n) \le 4T(n/2) + n^2$$

$$T(n/2) \le c(n/2)^2 \log (n/2)$$

$$T(n) \le 4c(n/2)^2 \log (n/2) + n^2$$

$$\le cn^2(\log n - 1) + n^2$$

$$cn^2 \log n - (c - 1)n^2$$

$c \ge 1$; inequality holds $T(n) \le cn^2 \log n$. for $c = 2$

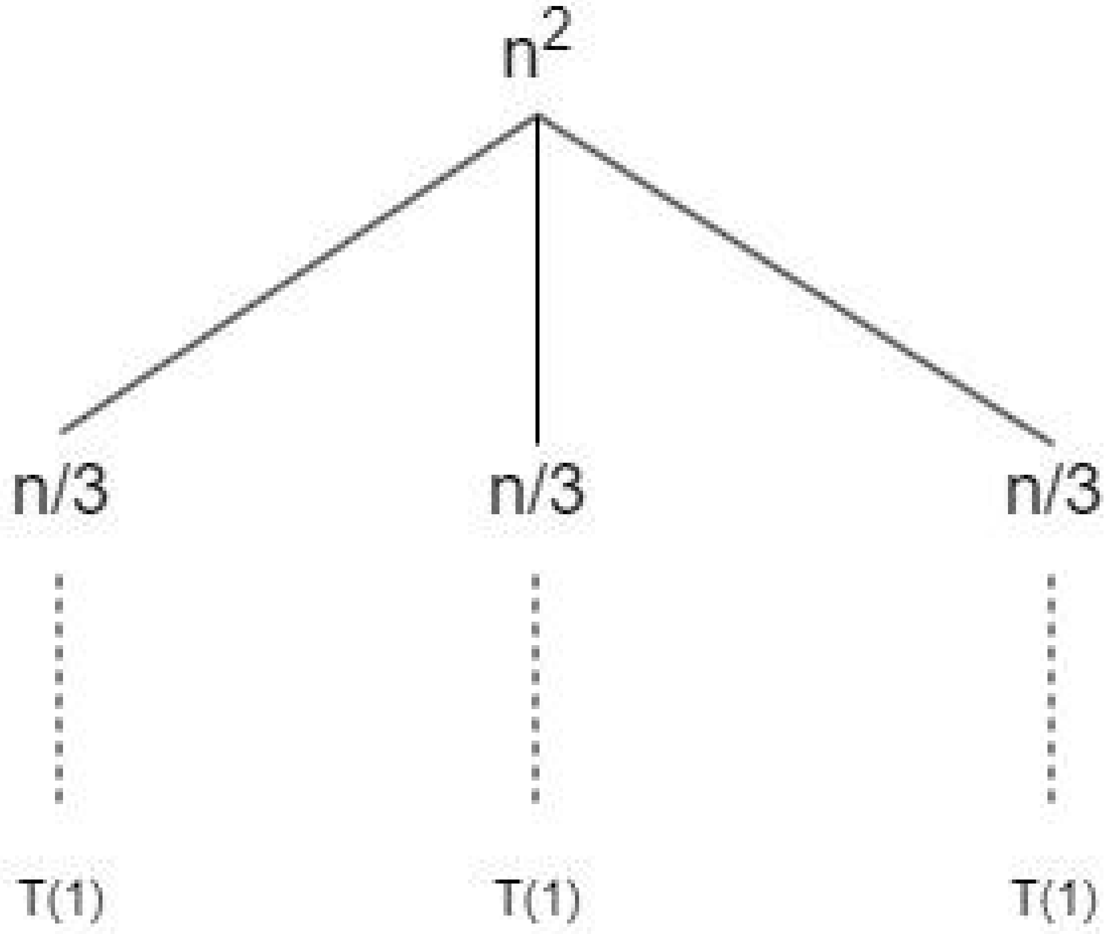
$T(n) = O(n^2 \log n)$.

2. A network splits data packet routing into 3 smaller subproblems. Processing each subproblem takes n time, and merging takes n^2 time.

   A. Write the recurrence equation for the above scenario considering recursive and non-recursive terms. Your final answer should be given in T(n) terms.

$$T(n) = 3T(n/3) + n^2$$

B. Solve the recurrence equation using iteration method. Note that tree structure includes root, depth and how leaves in some levels formation required to obtain full marks.



C. Verify solution using substitution method.

Assume $T(n)$ grows as $O(n^2)$.

$T(n) \le cn^2$, for c is constant.

base case : $n = 1$,

$T(k) \le c.n^2$; for all $k < n$.

$T(n) \le 3T(n/3) + n^2$

Inductive Hypothesis : $T(n/3) \le c(n/3)$

Inductive Step,

$$T(n) \le 3c(n/3)^2 + n^2$$

$$3c(n^2/9) + n^2$$

$$cn^2 + n^2; \qquad \text{choose c to hold eq.}$$

$$c \le 3/2$$

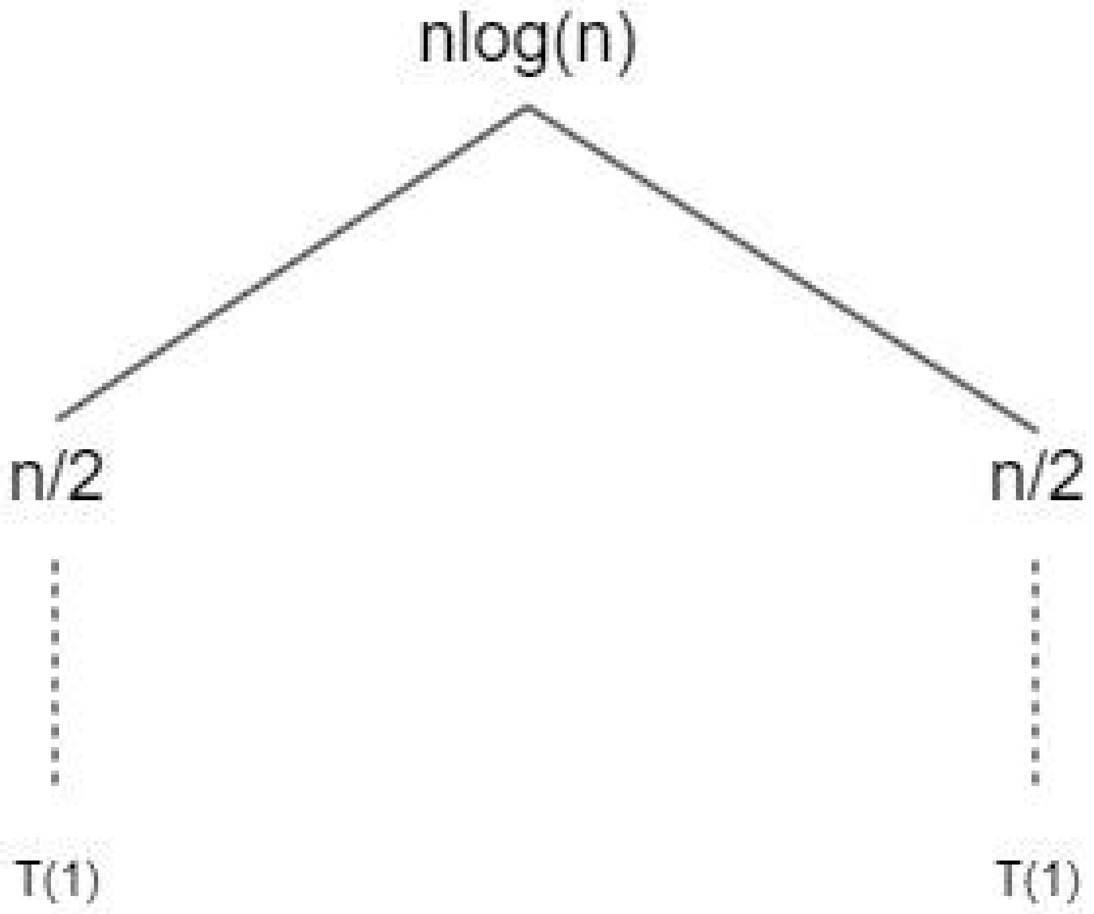
$$T(n) = O(n^2)$$

3. A deep learning model splits its dataset into two halves for training. Each half is trained recursively and combining results (using gradient merging) takes nlogn time.

A. Write the recurrence equation for the above scenario considering recursive and non-recursive terms. Your final answer should be given in T(n) terms.

$$T(n) = 2T(n/2) + n\log(n)$$

B. Solve the recurrence equation using an iteration method. Note that tree structure including root, depth and how leaves in some levels formation required to obtain full marks.



C. Verify solution using substitution method.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n\log n \\
&= 2[2T(n/4) + (n/2)\log (n/2)] + n\log n \\
&= 4T(n/4) + 2(n/2)(\log(n)-1)] + n\log n \\
&=4T(n/4) + 2n\log (n) - n \\
&=2^kT(n/2^k) + k.n\log (n)
\end{aligned}
$$

recursion stops when $n/2^k = 1$; $k = \log n$, $T(1) = O(1)$

$$
\begin{aligned}
T(n) &= 2^kT(1) + \log n.n\log n \\
&= n(O(1)) + n.\log^2 n \\
T(n) &= O(n\log^2 n)
\end{aligned}
$$

4. Consider the following program that reverse an array of integers in place. Prove the correctness of the loop invariant for initialization, maintenance and termination phases.

```
function reverseArray(arr,n):
left <- 0
right <- n-1

while left < right:
//Swap the elements at 'left' and 'right'
temp <- arr[left]
arr[left] <- arr[right]
arr[right] <- temp

//Move the pointers closer to the center
left <- left + 1
right <- right - 1

return arr
```

```
void reverseArray(int arr[], int n) {

 int left = 0;          // Initialize the left pointer
 int right = n - 1;     // Initialize the right pointer

while (left < right) {
            // Loop until the pointers meet or cross
            // Swap the elements at 'left' and 'right'
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
      // Move the pointers closer to the center
       left++;
       right--;
     }
}
```

Loop Invariant Definition

At the start of each iteration, the sub array

- **arr[0..left−1]**
- **arr[right+1..n−1]**

have been reversed, and **arr[left..right]** is yet to be reversed.

This invariant ensures that:
The parts of the array already processed **arr[0..left−1]** and **arr[right+1..n−1]** are correctly reversed.

Proving the Loop Invariant

Phase	Explanation
Initialization	Before the loop starts, no elements have been processed, and the invariant holds trivially.
Maintenance	Each iteration swaps the elements at $left$ and $right$, shrinking the unprocessed subarray while maintaining correctness.
Termination	When the loop ends ($left \geq right$), all elements have been reversed, ensuring the array is fully processed.

(a) Initialization (Before the First Iteration)

(a) At the Start: $left=0$ and $right=n-1$.

(b) The sub-array $arr[0..-1]$(before $left$) and $arr[n..n-1]$ (after $right$) are both empty, which satisfies the invariant since there's nothing to reverse initially.

(c) Conclusion: The loop invariant holds true before the first iteration.

Proving the Loop Invariant

(b) Maintenance (During Each Iteration)

- Action in Each Iteration:
  - Swap **arr[left]** and **arr[right]**.
  - Increment left and decrement right.
- Effect on the Array:
  - After the swap, **arr[left]** and **arr[right]** are correctly reversed.
  - The pointers left and right move inward, shrinking the unprocessed sub-array **arr[left..right]**.
- Invariant Holds:
  - After each iteration, the subarray **arr[0..left−1]** and **arr[right+1..n−1]** are reversed, and the middle part (arr[left..right]) remains to be processed.
  - Thus, the invariant is maintained throughout the loop.

Proving the Loop Invariant

(c) Termination (After the Loop Ends)

- Termination Condition: The loop ends when **left ≥ right**.
  - This means the entire array has been processed:
    - The pointers left and right meet or cross, leaving no unprocessed elements.
    - By the invariant, arr[0..left−1]and arr[right+1..n−1] have been reversed.
- Conclusion: At termination, the entire array arr[0..n−1] is reversed.

5. Consider the following program that shows whether a given number n is prime. Prove the correctness of the loop invariant for initialization, maintenance, and termination phases.

```
function isPrime(n):
        if n <= 1:
        return false
 // numbers less than or equal to 1 are
not prime
        for i from 2 to √n :
        if n % i == 0:
        return false
// n is divisible by i, so it's not prime
        return true
// n is prime if no divisors are found
```

```
bool isPrime(int n) {
  if (n <= 1) {
     return false;
// Numbers less than or equal to 1 are not prime
   }
   for (int i = 2; i <= sqrt(n); i++) {
            if (n % i == 0) {
                 return false;
// n is divisible by i, so it's not prime
            }
     } return true;
// n is prime if no divisors are found
 }
```

Loop Invariant Definition

The **loop invariant** for this function is:

"At the start of each iteration, no number from 2 to i−1 divides n."

This invariant ensures that if n is not divisible by any number less than i, it might still be a **prime**, and we need to continue the iterations.

Proving the Loop Invariant

Phase	Explanation
Initialization	Before the loop starts, no divisors are checked, and the invariant holds vacuously.
Maintenance	Each iteration ensures that n is not divisible by the current i, maintaining the invariant.
Termination	If no divisor is found by $\sqrt{n}$, n is prime because larger factors would require a smaller counterpart below $\sqrt{n}$.

(a) Initialization (Before the First Iteration)

Before the Loop Starts:

The loop runs from **I = 2 to √n**.

Before the first iteration, no numbers less than **2** exist, so the invariant is vacuously true.

Why It Holds:

No divisors have been checked yet, and there's no contradiction with the invariant.

Proving the Loop Invariant

(b) Maintenance (During Each Iteration)

- **Action in Each Iteration**:
- Check if **n** mod **i**=0 :

1. If true, **n** is divisible by **i**, so n is not prime, and the function returns false.
2. If false, **i** does not divide **n**, and the loop continues to the next iteration.

- **Why It Holds**:
- Before each iteration, no number from 2 to **i−1** divides **n**.
- The current iteration ensures **i** does not divide **n** before moving on to **i+1**.
- Thus, the invariant is maintained after each iteration.

Proving the Loop Invariant

c) Termination (After the Loop Ends)

Termination Condition:

The loop ends when $i > \sqrt{n}$.

By the invariant, no number from 2 to $\sqrt{n}$ divides n.

Why It Holds:

If no divisor has been found by $\sqrt{n}$, then n cannot have any divisors greater than $\sqrt{n}$ because any factor pair (a,b) of n satisfies $a \times b = n$. At least one of a or b must be $\leq \sqrt{n}$.

MCQ QUESTIONS DISCUSSION

Question 05: In a cryptographic hash function, a loop processes chunks of data to compute the hash. What invariants ensure correctness?

Correct Options:

- **A. Each chunk contributes uniquely to the hash**:
  - Each chunk must have a unique impact on the hash value. Without uniqueness, different inputs could produce the same hash (violating the hash function's integrity).
- **B. Chunks are processed in the same order for the same input**:
  - The order of processing must be consistent; otherwise, the same input could yield different hashes (violating determinism).
- **C. The final hash size is fixed regardless of input size**:
  - Cryptographic hash functions produce fixed-size outputs (e.g., 256 bits for SHA-256) regardless of the input length.

Incorrect Option:

- **D. All chunks must be equal in size**:
  - This is not required. Padding techniques are used to handle uneven chunks if needed.

Question 06 : A loop iterates through tasks to schedule them in a time slot. The invariant ensures no overlap between tasks. What additional conditions might ensure correctness?

Correct Options:

- **A. Tasks are scheduled in the order of their deadlines**:
  - Scheduling tasks by deadlines ensures that tasks with the earliest deadlines are prioritized, minimizing the risk of missed deadlines.
- **B. A task is only scheduled if it fits within the time slot**:
  - A task must fit into the available slot to prevent overlap.
- **D. The algorithm terminates when all tasks are considered**:
  - The loop must ensure all tasks are either scheduled or skipped to achieve correctness.

Incorrect Option:

- **C. Unscheduled tasks are moved to the next time slot**:
  - This is not necessarily required for correctness. Some algorithms may discard tasks that cannot be scheduled.

Question 07 : You are tasked with writing a loop to sort an array A[1…n] in ascending order. Which of the following could be valid loop invariants?

Correct Options:

- **A. The sub-array A[1…i] is sorted at the i-th iteration**:
  - A common invariant for insertion sort, where each iteration extends the sorted portion of the array.
- **C. No element in A[1…i] is greater than any element in A[i+1…n]:**
  - A valid invariant for selection sort, ensuring that the sorted portion has only smaller elements than the unsorted portion.

Incorrect Options:

- **B. The largest element in A[i…n] is always at A[i]**:
  - This describes bubble sort but isn't true in all sorting algorithms.
- **D. All elements are sorted when the loop exits**:
  - This is true at the end of the algorithm, but it's not an invariant (a condition that holds at every step).

Question 08 : A loop checks if a string of parentheses is balanced. What invariants hold?

Correct Options:

- **A. The count of open parentheses is non-negative at each step**:
  - At no point should there be more closing parentheses than opening parentheses.
- **B. The total count of open and closed parentheses matches**:
  - For the string to be balanced, the counts of open and closed parentheses must be equal.

Incorrect Options:

- **C. The string is balanced at any intermediate step**:
  - This is not necessarily true for intermediate states, as balancing is only guaranteed at the end.
- **D. The algorithm terminates with a count of zero**:
  - This is a property of the final result, not an invariant during the loop.

Understanding Question 08: Balanced Parentheses

A balanced string has:

1. Equal numbers of opening (and closing) parentheses.
2. Closing parentheses) never outnumber opening parentheses (at any point.

The question revolves around loop invariants, conditions that must hold true during every iteration of the loop.

Input String: (())()

Execution for `(())()`**:**

Step	Index	Character	Balance	Explanation
Start	-	-	0	Initial balance is 0.
1	0	`(`	1	Open parenthesis increments balance.
2	1	`(`	2	Another `(` increments balance.
3	2	`)`	1	Closing parenthesis decrements balance.
4	3	`)`	0	Another `)` decrements balance to 0 (balanced so far).
5	4	`(`	1	Open parenthesis increments balance.
6	5	`)`	0	Closing parenthesis decrements balance to 0 (balanced).

Question 09 : A loop calculates the n-th Fibonacci number iteratively. What invariants ensure correctness?

Correct Options:

- **A. At step i, the variable ‘fib1‘ stores F(i–1)**:
  - The first variable represents the previous Fibonacci number.
  - at ***i*=3, fib1 = F(2).**
- **B. At step ii, the variable ‘fib2‘ stores F(i)**:
  - The second variable represents the current Fibonacci number.
  - **i=3, fib2 = F(3).**
- **C. The variables fib1 and fib2 always hold consecutive Fibonacci numbers**:
  - The loop updates both variables to maintain this relationship.

Incorrect Option:

- **D. The algorithm terminates after calculating F(n)**:
  - While true, this is not an invariant (it doesn’t hold during the loop).

$F(0)=0, F(1)=1 \rightarrow$

$F(n)=F(n-1)+F(n-2)$, for $n \geq 2$

For example:

F(0)=0,

F(1)=1,

F(2)=1,

F(3)=2,

F(4)=3,

F(5)=5,

F(6)=8,…

Output Explanation for $F(5)$

Step	fib1 (F(i-1))	fib2 (F(i))
2	0	1
3	1	2
4	2	3
5	3	5

Question 10: When iterating through an array to find the maximum element, which invariants ensure correctness?

Correct Options:

- **A. The variable `max_so_far` is greater than or equal to any element in A[1…i]**:
  - Ensures the variable holds the maximum value encountered so far.
- **D. No element before iii is greater than `max_so_far`**:
  - Ensures all elements processed so far are less than or equal to the current maximum.

Incorrect Options:

- **B. The variable `max_so_far` is updated whenever a larger element is found**:
  - This describes an action, not an invariant.
- **C. After the loop exits, `max_so_far` is the maximum element in A[1…n]**:
  - This is true post-loop but not during execution, so it's not an invariant.

Input Array: [3,7,2,9,5]

Step	Index (i)	Element ($A[i]$)	max_so_far	Explanation
Start	-	-	3	Initialize `max_so_far` with the first element ($A[1]$).
1	1	7	7	$A[2] = 7 > max\_so\_far = 3$, so update `max_so_far = 7`.
2	2	2	7	$A[3] = 2 < max\_so\_far = 7$, no update.
3	3	9	9	$A[4] = 9 > max\_so\_far = 7$, so update `max_so_far = 9`.
4	4	5	9	$A[5] = 5 < max\_so\_far = 9$, no update.