Multithreaded Mergesort

Name(s): _____

Part 1: Tracing the Execution DAG

Algorithm 1 MERGE-SORT$'(A, p, r)$

```
1: if p < r then
2:     q = ⌊(p + r)/2⌋
3:     spawn MERGE-SORT'(A, p, q)
4:     MERGE-SORT'(A, q + 1, r)
5:     sync
6:     MERGE(A, p, q, r)
7: end if
```

Scenario: Sort the array $A = [8, 3, 5, 1]$. Assume the base case (size 1) takes **1 unit** of work. Assume the MERGE step takes k **units** of work (where k is the length of the merged subarray).
Task: Draw the Execution DAG. Represent **spawn** as a split and **sync** as a join. Write the time cost next to each node.

Part 2: Work vs. Span

Using your DAG from Part 1, calculate the following:

1. **Total Work (T_1):** _____

2. **Span (T_∞):** _____

3. **Parallelism (T_1/T_∞):** _____

Part 3: Correctness & Loop Invariants

Before scaling up, the sequential components must be logically sound. Consider the inner loop of the MERGE function:

```
for k = p to r do
    if i ≤ n1 and (j > n2 or L[i] ≤ R[j]) then
        A[k] ← L[i]; i ← i + 1
    else
        A[k] ← R[j]; j ← j + 1
    end if
end for
```

Task: State the **Loop Invariant** for this loop regarding the subarray $A[p \dots k-1]$.

Briefly explain the three stages of this invariant:

- **Initialization:** _____
- **Maintenance:** _____
- **Termination:** _____

Part 4: The Broken Barrier

A developer deletes Line 5 (**sync**) from Part 1. They argue that since the left half and right half of the array use different memory indices, they don't need to wait for each other.

Why is this a "Race Condition"? What is the specific risk to the Merge step?

Part 5: Theoretical Scaling & Amdahl's Law

In the pseudocode in Part 1, the MERGE step is **sequential**.

1. The Work of MERGE is $\Theta(n)$. Since it is sequential, its **Span** is also $\Theta(n)$.

2. Write the recurrence for the **Total Span** (T_∞) of the entire algorithm:

3. Using the Master Theorem, what is the Θ of the Span? _____

4. Calculate the ultimate Parallelism (T_1/T_∞). As $n \to 1,000,000$, what is the maximum theoretical speedup we can achieve, regardless of core count?

Part 6: The Memory Wall

Mergesort is an out-of-place sorting algorithm, requiring $\Theta(n)$ auxiliary space to perform the merge. In a multithreaded environment, hardware matters just as much as Θ notation.

Question 1: If 16 threads are all trying to allocate new arrays and copy data back and forth to RAM simultaneously, what hardware component becomes the bottleneck? (Hint: It connects the CPU to Main Memory).

Question 2 (Cache Locality): Why might a *sequential* in-place QuickSort sometimes outperform a *parallel* MergeSort on a multi-core machine for medium-sized arrays?

Part 7: True Parallel Merge

To fix the $\Theta(n)$ bottleneck, we must parallelize the Merge step itself. We have two sorted arrays, A and B, that need to be merged into a result array C.

The Strategy:

1. Find the median element x of the larger array (assume A).

2. Use **Binary Search** to find the index in B where x would be inserted.

3. This splits A into A_{left} and A_{right}, and B into B_{left} and B_{right}.

4. **Spawn** a thread to merge (A_{left}, B_{left}) and another for (A_{right}, B_{right}).

Trace the Algorithm:
Assume $A = [2, 4, 7, 9, 12]$ and $B = [1, 5, 8, 10, 15]$.

1. What is the median element x of A? _____
2. If we binary search for x in B, between which two numbers does it land? _____
3. Write out the contents of the four resulting subarrays:

- A_{left}: _____
- A_{right}: _____
- B_{left}: _____
- B_{right}: _____

By breaking the merge into independent pieces, the new span of the merge becomes $\Theta(\log^2 n)$.

Part 8: Real-World Implementation

The keywords **spawn** and **sync** are theoretical. How would you actually implement this in a modern programming language?

- **C++:** _____
- **Java:** _____