AAD Quiz-1 Answer Key

Fall 2025

Problem 1

No, It's not true for ANY of the cases. Multiplication can be reduced to squaring and vice versa with only a constant number of calls plus lower-order work. Hence squaring cannot be asymptotically faster than multiplication. [**No Marks for Just the claim**]

Let $M(n)$ denote the time to multiply two size-n operands and $S(n)$ the time to square one size-n operand (numbers with n bits, $n \times n$ matrices, or degree $< n$ polynomials).

Case 1: Trivial Reduction (all three domains)

Squaring is a special case of multiplication given as $a^2 = a \cdot a$. Thus

$$S(n) \leq M(n) \quad \text{for all } n.$$

Notice that it works even when a is a matrix.

Case 2 Multiplication reduces to a constant number of squarings

(2.A) Integers

For n-bit integers a, b,

$$4ab = (a+b)^2 - (a-b)^2.$$

Algorithm: compute $u = (a+b)^2$ and $v = (a-b)^2$, then output $(u-v)/4$. This requires two squarings of $(n+1)$-bit integers (since $|a \pm b|$ has at most $n+1$ bits). The additions/subtractions and the exact division by 4 (right-shift by 2 bits) cost $O(n)$. Therefore,

$$M(n) \leq 2S(n+1) + O(n) = O(S(n)).$$

(Over $\mathbb{Z}$ the division by 4 is exact because the numerator equals $4ab$ and bit shifting causes no more than $O(n)$ addition)

[1.5 Marks for M $\rightarrow$ S and 1 Mark for S $\rightarrow$ M = 2.5]

(2.B) Matrices

Let A, B be $n \times n$. Consider the $2n \times 2n$ block matrix

$$M = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix}.$$

Then

$$M^2 = \begin{bmatrix} AA & AB \\ 0 & 0 \end{bmatrix}.$$

So one squaring of a $2n \times 2n$ matrix yields AB (read off the top-right block). Building M and extracting AB cost $O(n^2)$, negligible compared to the squaring cost. Hence

$$M(n) \leq S(2n) + O(n^2) = O(S(n)),$$

(because for all known matrix-squaring algorithms $S(kn) = \Theta(k^\alpha S(n))$ for some constant $\alpha > 2$; in particular with Strassen, $\alpha = \log_2 7$, so $S(2n) = \Theta(2^\alpha S(n))$, a constant-factor blow-up). **[4 Marks for M $\rightarrow$ S and 1 Mark for S $\rightarrow$ M = 5]**

Hence, We have $S(n) \leq M(n)$ and $M(n) = O(S(n))$ in each of the three settings, hence

$$S(n) = \Theta(M(n)).$$

Therefore, squaring cannot be asymptotically faster than multiplication (and conversely). Any algorithm that squares faster would immediately give an equally fast multiplication algorithm by the reductions above.

(2.C) Polynomials

Over coefficient rings/fields used in FFT-based multiplication (e.g., $\mathbb{C}$ or $\mathbb{F}_p$ with odd p), the same identity holds coefficient-wise:

$$4A(x)B(x) = (A+B)^2 - (A-B)^2.$$

Again we need two squarings of degree-$\leq n$ polynomials plus $O(n)$ coefficient work and an exact division by 4 (a constant-time scalar inverse). Thus

$$M(n) = O(S(n)).$$

[1.5 Marks for M $\rightarrow$ S and 1 Mark for S $\rightarrow$ M = 2.5]

Following proof strategy is incorrect

When someone claims "squaring a matrix is easier than multiplying two different matrices", a common mistake is to expand blockwise:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad M^2 = \begin{bmatrix} A^2 + BC & AB + BD \\ CA + DC & CB + D^2 \end{bmatrix}.$$

But this only shows **one naive way** to square. There may be any way. Why any such methodology is wrong becuase complexity is about the *best possible algorithm*, not how you personally expanded the product or applied the stated algorithm. You cannot reverse-engineer a "black-box squaring routine" and assume it behaves like your expansion. By definition the burden of proof is on you for your algorithm's complexity, not on the me for my black box. The right way to resolve the claim is through **reductions**:

- Multiplication $\rightarrow$ squaring (using one squaring on a block matrix).
- Squaring $\rightarrow$ multiplication (trivial since $a^2 = a \cdot a$).

This shows

$$\text{Squaring}(n) = \Theta(\text{Multiplication}(n)).$$

Hence any claim that squaring is *asymptotically faster* immediately collapses, because a faster squaring would imply a faster multiplication too. Contradiction complete.

Problem 2: Interval Coloring

Question: Let X be a set of n intervals on the real line. A coloring of X assigns a color to each interval so that any two overlapping intervals are differently colored. Design, analyze, prove the correctness/optimality of, as well as compare with a naive approach, an algorithm to compute the minimum number of colors needed to color X. (Marks: $4.5 + 1.5 + 3 + 1$)

Part 1: Algorithm Design (4.5 Marks)

This section assesses the student's ability to formulate a correct and efficient greedy algorithm.

- **[1.5 marks] - Sorting:**
  - 1.0 mark: For explicitly stating that the intervals must be sorted based on their start times in increasing order.
  - 0.5 marks: For mentioning how to handle ties in start times.
- **[3.0 marks] - Greedy Logic & Data Structures:**
  - 2.0 marks: For clearly describing the core greedy logic: Iterate through the sorted intervals. For each interval i, assign it the smallest possible color (e.g., color 1, 2, 3, ...) that is not currently being used by any interval that overlaps with i and has already been processed.
  - 1.0 mark: For describing the data structures used to manage colors. A good answer might mention:
    - A way to track which colors are "in use" for intervals that are currently "active" (i.e., have started but not yet finished).
    - A min-priority queue (or a sorted list/set) to keep track of available colors is an excellent approach.
    - A simple array/list check is also acceptable.
- NOTE: any algorithm which is not greedy or $> O(nlogn)$ should be awarded a maximum of 2.5 marks as it is not eligible for the 2 marks of greedy logic.

Example of a High-Quality Answer:

1. Create a list of events (or points) from the interval endpoints. An event is a tuple (time, type, interval_index), where type is either 'start' or 'end'.

2. Sort all $2n$ events primarily by time. If times are equal, 'start' events come before 'end' events.

3. Initialize an empty list of `available_colors` (e.g., a stack or queue) and a variable `max_colors_used` $= 0$.

4. Iterate through the sorted event list:

   - If it's a 'start' event:
     - Fetch a color from `available_colors`. If none, create a new one (increment `max_colors_used`).
     - Assign this color to the corresponding interval.
   - If it's an 'end' event:
     - Take the color of the finished interval and add it back to `available_colors`.

5. Return `max_colors_used`.

Part 2: Algorithm Analysis (1.5 Marks)

This section assesses the analysis of the algorithm's time complexity.

- **[0.5 marks] - Sorting Complexity:** For correctly stating that sorting n intervals takes $O(n \log n)$ time.

- **[0.75 marks] - Iteration Complexity:** For analyzing the main loop. The student must explain that they iterate through n intervals once. The complexity of finding an available color in each iteration depends on the data structure used, but with a simple list of k colors, it could be $O(k)$, making the loop $O(nk)$. A more optimal approach (using a min-heap for colors) would be $O(n \log k)$. In the worst case, k can be n, so $O(n^2)$ or $O(n \log n)$ are possible answers depending on implementation. Full marks for either, provided the justification is sound.

- **[0.25 marks] - Overall Complexity:** For correctly concluding that the overall time complexity is dominated by the sorting step, resulting in $O(n \log n)$. (Depending on implementation, it can be a higher polynomial like $O(n^2)$. In this case, if they are able to identify/analyze the algorithm to reach their algorithm's time complexity accurately, award them full credit for algorithm analysis).

NOTE: correct answer (matching with their algorithm; not necessarily $O(n \log n)$), with no explanation gives 0.5 marks. $O(n \log n)$ with no/incomplete algorithm gives 0.25 marks.

Part 3: Proof of Correctness/Optimality (3 Marks)

This is the most theoretical part. The student must prove that the greedy algorithm uses the minimum number of colors.

- **[1.0 mark] - Defining the Lower Bound:**
  - The student must define k as the maximum depth (or maximum load) of the set of intervals - i.e., the maximum number of intervals that overlap at any single point in time.
  - They must argue that any valid coloring requires at least k colors. This is the lower bound, as those k mutually overlapping intervals must all receive a different color.
- **[1.5 marks] - Proving the Algorithm Achieves the Lower Bound (Upper Bound):**
  - This is the core of the proof. The argument should be a proof by contradiction or a direct proof.
  - Argument: Assume the algorithm uses m colors, and $m > k$. Let's focus on the first interval, say i, that is assigned color m.
  - When the algorithm assigns color m to interval i, it's because colors $1, 2, \ldots, m-1$ are all unavailable.
  - This means that at the start time of interval i, there are $m-1$ other intervals that have already started but have not yet finished, and are thus active and overlapping with i.

- These $m-1$ intervals, plus interval i itself, form a set of m intervals that are all mutually overlapping at the start point of i.
- This implies that the depth at this point is at least m.
- This contradicts our assumption that the maximum depth is k (since we assumed $m > k$). Therefore, the assumption that the algorithm uses more than k colors must be false. The algorithm uses at most k colors.

- **[0.5 marks] - Conclusion of Proof:**
  - For explicitly stating the conclusion: Since any algorithm needs at least k colors (lower bound) and our greedy algorithm uses at most k colors (upper bound), it must be optimal, using exactly k colors.

NOTE: a non-optimal algorithm ($> O(n \log n)$) should only be given 1 mark if it is proved correct and should not be credited for optimality.

Part 4: Comparison with a Naive Approach (1 Mark)

This section assesses the student's ability to conceptualize a less efficient solution and compare it.

- **[0.5 marks] - Describing a Naive Approach:**
  - The student must describe a valid naive (brute-force or backtracking) algorithm. For example:
    - "Try all possible assignments of c colors to n intervals and check for validity, starting with $c = 1$, then $c = 2$, etc."
    - "Model it as a graph coloring problem where intervals are vertices and an edge exists between overlapping intervals. Then, try to k-color the graph."

- **[0.5 marks] - Analyzing and Comparing the Naive Approach:**
  - The student must state that the complexity of such an approach is exponential.
  - For instance, trying to assign one of k colors to n intervals can be in the order of $O(k^n)$.

– They must explicitly compare this exponential complexity to the greedy algorithm's efficient $O(n \log n)$ complexity, highlighting the vast improvement.

1 Problem 3

Part 1 Solution:

Lemma 1: The array will follow the following property: A prefix of the array till index $k \in [0, n]$ (note that this means the prefix may be empty) will have the following property satisfied: $A[i] < i$ for all $i \leq k$. The other part of the array, after index k will have the following property satisfied: $A[j] \geq j$ for all $j > k$.

Proof: Assume the property does not hold. Then, there exists an index x such that $x \in [2, n]$ and $A[x] < x$ but $A[x-1] \geq x-1$.

Since $A[x-1] \geq x-1$, and we know that the array has distinct integers and is sorted, the smallest positive integer is 1. Thus,

$$A[x] \geq A[x-1] + 1 = x - 1 + 1 = x.$$

Hence $A[x] \geq x$ and, thus, such an x does not exist by contradiction.

□

Now that we have proved this property, we can binary search on this. For a range $[l, r]$ (starting from $[1, n]$), let

$$mid = \left\lfloor \frac{l+r}{2} \right\rfloor.$$

If $A[mid] = mid$, then we have found the answer. Else if $A[mid] < mid$, the answer lies to the right, and hence we can change $l = mid + 1$. Similarly, when $A[mid] > mid$, the answer lies to the left, and hence, we can change $r = mid - 1$. Finally, if l and r form a null range, we are ensured that no such answer exists.

Part 2 Solution:

Note: $A \cup B$ may have repeated elements. For example, $A = [0, 2]$ and $B = [2, 5]$. Thus, the exact same solution of part 1 will fail. However, there exist multiple solutions to solve this problem.

We follow a similar idea of part 1, with some modifications. Let's say we binary search on the value of k and find the number of elements in A and B that are less than or equal to k. Here, if the number of elements is equal to $k+1$, we have to do an additional check – whether there exists an element with the value equal to k in A and B both. If there exists an element with the value equal to k in both A and B, they will take up positions k and $k+1$

in the array $A \cup B$, and hence we can finish our algorithm here itself. The rest of the algorithm follows part 1.

Proof of correctness: We have proved that binary search will work in part 1 of the solution. Now we prove the modification, if a solution (lets say at index and value k) exists, then:

1. The number of elements lesser than or equal to it is either k or $k+1$,
2. And if it is $k+1$, then the element occurs both in A and B.

Proof of 1

If the number of elements lesser than or equal to it is $< k$, then the kth element in $(A \cup B)$ will be greater than k because of the statement. Furthermore, if the number of elements lesser than or equal to it is $> k+1$, then there have to exist atleast 3 elements in $A \cup B$ so that $(A \cup B)[k] = k$. Since A and B are distinct within themselves, $(A \cup B)$ can have atmost one such element from A and one such element from B. Which means, that $A \cup B$ contains atmost 2 elements of the same value.

Proof of 2

Assume that the element occurs either in A or B (since the element has to occur atleast once). WLOG, assume it occurs in A. Since all elements are less than or equal to k, then $(A \cup B)[k+1] = k$, but $(A \cup B)[k] < k$. Hence, this value cannot be the answer, which contradicts our claim.

Time complexity:

$$O(\log(n+m)) \quad \text{for iterating over all } k,$$

$$O(\log(\max(n,m))) \quad \text{for finding } (A \cup B)[k].$$

Total time complexity: $O(\log(n+m) \cdot \log(\max(n,m)))$.

Marking Scheme

1. **Part 1 Solution**
   (a) Stating Lemma 1 – (1 mark)
   (b) Proof of correctness of algorithm – (2 marks)
   (c) Pseudocode – (2 marks)

2. **Part 2 solution**

   (a) Pseudocode – (2 marks)

   (b) Proof of correctness – (2 marks)

   (c) Time complexity – (1 mark)

   (d) Other solutions:

      i. If the solution doesn't consider duplicate elements in $A \cup B$, then the marks are capped at 3. If an assumption is mentioned regarding the same, the marks will be capped at $3\frac{1}{2}$ instead.

      ii. If the solution has an asymptotically worse time complexity, then the marks are capped at $3\frac{1}{2}$.