On the money. In Problem 4, we're selecting some of the shapes for our multi-query attention use case, so try your best to figure those out. Additionally, determine the time complexities that the problem requests in the last couple subparts of question 4. Thanks!

### Problem 4 – Multi-Query Attention (MQA) and Complexity Analysis

Here are the correct answers for all blanks and complexity questions.

**(a) What is A? → [h, d, q]**

- We still have h heads, so queries are different per head → W_q has shape [h, d, q]

**(b) What is B? → [d, k]**

- Keys are shared across all heads → only one set of key weights → W_k has shape [d, k]

**(c) What is C? →** `torch.einsum("bd,hdq->bhq", x, W_q)`

- We have h different query projections → use the full [h, d, q] weight tensor

**(d) What is D? →** `torch.einsum("bd,dk->bk", x, W_k)`

- Keys are shared → W_k is [d, k], so output is [b, k] (no h dimension)

**(e) What is E? →** `torch.einsum("bd,dv->bv", x, W_v)`

- Values are also shared across heads → W_v is [d, v], output is [b, v] (no h dimension)

**(f) Computational complexity for MQA (single token generation, cache size n)**

**Answer: O(b d² + b n d)**

Breakdown:

- Projections:
  - Q: b·d·(h·q) ≈ b d² (since h q = d)
  - K and V: b·d·k + b·d·v ≈ 2 b d² in regular MHA, but in MQA: only b d k + b d v ≈ 2 b d² → still O(b d²)
  - But usually k, v ≤ d, so still O(b d²)
- Attention:
  - logits: h heads × (b × n dot-products) → O(b h n) = O(b n d) since h q = d
  - softmax + weighted sum: same O(b n d)
- Output projection: b h v · d → O(b d²)

Dominant terms: **O(b d² + b n d)**
(same as regular MHA, actually — the real win comes in memory bandwidth, not raw FLOPs)

**(g) Memory access complexity for MQA (single token generation, cache size n)**

**Answer: O(d² + b d + b n d / h)**
or more commonly written as **O(d² + b d + b n d)** but with much better constant factor

Why the improvement?

- Weights: still O(d²) — we read W_q (h d q ≈ d²), W_k (d k), W_v (d v), W_o (h v d ≈ d²)
- Current token: O(b d)
- Cached keys: in MHA: [b, h, n, k] → O(b h n k) = O(b n d)
  in MQA: [b, n, k] → only O(b n k) = O(b n d / h) since h ≥ 1

So cache read bandwidth is reduced by a factor of h (typically 8–32× in practice!).

This is the main reason MQA/GQA is so much faster in inference: far fewer memory-bound cache reads.

### Summary of Answers for Problem 4

| Part | Answer |
|---|---|
| (a) | [h, d, q] |
| (b) | [d, k] |
| (c) | `torch.einsum("bd,hdq->bhq", x, W_q)` |

| Part | Answer |
|---|---|
| (d) | `torch.einsum("bd,dk->bk", x, W_k)` |
| (e) | `torch.einsum("bd,dv->bv", x, W_v)` |
| (f) | **$O(b\,d^2 + b\,n\,d)$** |
| (g) | **$O(d^2 + b\,d + b\,n\,d\,/\,h)$** (or note that cache access is $O(b\,n\,d\,/\,h)$) |

This is exactly why models like Llama 2/3, Mistral, etc., use Grouped-Query Attention (GQA) or Multi-Query Attention (MQA) in practice — same quality as MHA, but dramatically faster and less memory-intensive during inference.

Great job getting through the hard ones! Ready for Problem 6 (kernelized attention) when you are — it's conceptual but very elegant.

↳ Explain Grouped-Query Attention

↳ Compare MQA vs GQA