Not All Thoughts Matter:
Selective Attention for Efficient Reasoning

Hao Tang[2]   **Guoqing Zheng**[1]   **Kanishk Gandhi**[3]   **Harkirat Behl**[1]

Vaishnavi Shrivastava[1]   **Mojan Javaheripi**[1]   **Kevin Ellis**[2]   **Shivam Garg**[1]

Dimitris Papailiopoulos[1]

[1]Microsoft Research   [2]Cornell University   [3]Stanford University

Abstract

Reasoning-optimized language models increasingly rely on test-time compute–long chains of thought before final answers–to boost accuracy, but this raises cost because causal self-attention scales quadratically in time and linearly in memory with sequence length. We observe that many intermediate thoughts are redundant: the model rarely needs to attend to all past tokens to generate effective next tokens and reach correct solutions. We propose RollingWindowReasoner, a simple yet effective inference-time technique that maintains only the first window (preserving critical problem context) and the last window (recent reasoning steps) of the key-value cache. Experiments across two model families and three reasoning domains–math reasoning, code generation, and academic QA–demonstrate that RollingWindowReasoner achieves similar accuracy with only 50% of the KV-cache budget, corresponding to $2\times$ memory savings and $4\times$ compute reduction.

1   Introduction

The emergence of reasoning-optimized language models has demonstrated remarkable improvements in complex problem-solving tasks through test-time compute scaling laws [12, 11]. This paradigm allows models to "think longer" by generating extended chains of reasoning before producing final answers, enabling LLMs to improve their outputs by using more test-time computation. Models such as OpenAI's o1 and o3 [6], and DeepSeek's R1 series [5] generate extensive chains of thought–intermediate reasoning steps–before producing final solutions, leading to significant performance gains on challenging benchmarks.

Generating long sequences, however, poses significant computational challenges due to the attention mechanism's quadratic scaling [13]. For a sequence of length n, the attention computation requires $O(n^2)$ time complexity and $O(n)$ space complexity for storing key-value (KV) caches. As reasoning chains can extend to tens of thousands of tokens – DeepSeek R1 averages 23K reasoning tokens for AIME problems [5] – these costs become prohibitive for practical deployment.

We introduce RollingWindowReasoner, an inference-time technique that exploits the inherent redundancy in reasoning chains (see Appendix B for examples). Our key insight is that models do not require access to all historical tokens to generate effective next tokens or reach correct solutions. By maintaining only strategically selected portions of the KV cache–specifically the first window containing critical problem context and the last window containing recent reasoning steps – we achieve significant computational savings while preserving reasoning quality. Surprisingly, we find
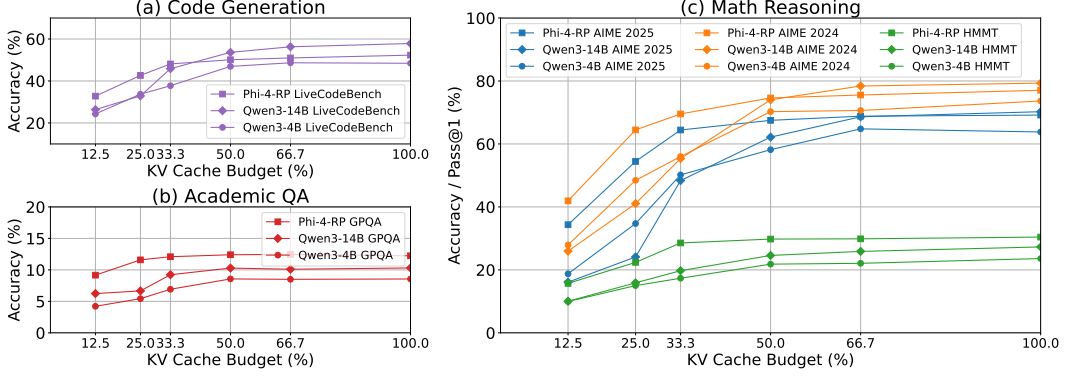
Figure 1: RollingWindowReasoner performance across reasoning domains and model families. Results show consistent efficiency gains across (a) code generation (LiveCodeBench), (b) academic question answering (GPQA), and (c) math reasoning tasks (AIME25, AIME24, HMMT25). Different markers represent different models (Phi-4-reasoning-plus, Qwen3-4B, and Qwen3-14B). Performance plateaus at 50% cache budget across all domains, demonstrating the generalizability of our approach.

this inference time technique works out-of-the box *even for models solely trained with quadratic full-sequence attention*: No fine-tuning is necessary.

Our contributions are threefold: (1) We propose a simple yet effective rolling window mechanism that reduces attention complexity from quadratic to linear time and from linear to constant space; (2) We demonstrate consistent performance across multiple model families and reasoning tasks; (3) We provide ablation studies validating our design choices and comparing against more complex truncation strategies including one using attention weights as hints for KV-cache selection: Even though our method is much simpler, its performance is essentially the same as more complex decoding strategies.

2 Method

Problem Formulation. Consider a reasoning model generating a sequence of tokens $x_1, x_2, \cdots, x_T$ where early tokens typically contain the problem statement and instructions, while later tokens represent iterative reasoning steps. Standard causal self-attention requires computing attention weights over all previous tokens at each generation step t:

$$\text{Attention}(Q_t, K_{1:t}, V_{1:t}) = \text{softmax}\left(\frac{Q_t K_{1:t}^T}{\sqrt{d}}\right) V_{1:t}$$

This results in $O(t^2)$ time complexity for generating t tokens and $O(t)$ space complexity for storing KV caches.

RollingWindowReasoner Algorithm. Our method maintains KV caches for only two windows: a first window of size w_f containing initial tokens (problem statement, instructions) and a last window of size w_l containing recent reasoning steps. The total cache budget is $B = w_f + w_l$, typically much smaller than the full sequence length.

Algorithm 1: truncate_kv_cache

Input: Key cache K, Value cache V, Sequence length L,
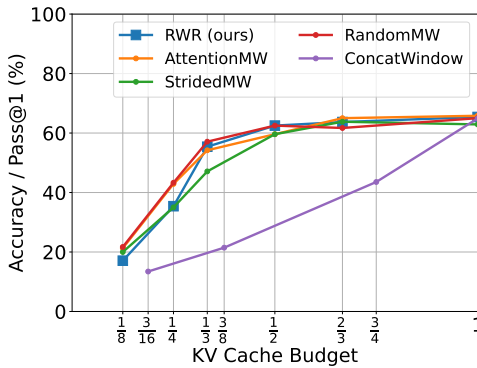First window size w_f, Last window size w_l
Output: Truncated key and value caches

1 **if** $L > w_f + w_l$ **then**
2 $\quad$ $N \leftarrow$ `number of layers in` K;
3 $\quad$ **foreach** *layer index* ℓ *in* $0 \dots N{-}1$ **do**
4 $\quad\quad$ $K[\ell] \leftarrow$ `concat`$(K[\ell][:,:,0{:}w_f,:],\ K[\ell][:,:,-w_l{:},:])$;
5 $\quad\quad$ $V[\ell] \leftarrow$ `concat`$(V[\ell][:,:,0{:}w_f,:],\ V[\ell][:,:,-w_l{:},:])$;

6 **return** K, V

When generating a new token, our algorithm follows a three-step process. First, we append new key-value pairs to the last window to incorporate the most recent reasoning step. Second, if the last window exceeds the predetermined size w_l, we remove the oldest key-value pair from the last window to maintain the fixed budget constraint. Third, we always preserve the first window unchanged to ensure the critical problem context remains accessible throughout the generation process. This approach fundamentally transforms the attention complexity from the standard $O(t^2)$ scaling to $O(\min(tB, t^2))$ time complexity and $O(\min(t, B))$ space complexity, where $B = w_f + w_l$ represents the fixed cache budget. For long sequences where $t \gg B$, this effectively becomes $O(Bt)$ time and $O(B)$ space.

The two-window design exploits the distinct functional roles of different token regions in reasoning tasks. The first window contains problem statements, instructions, and constraints that remain persistently relevant throughout the entire reasoning process, serving as the foundational context that guides solution development. The last window maintains recent reasoning steps and intermediate conclusions that directly inform subsequent token generation, capturing the model's current line of thinking and immediate logical dependencies. In contrast, middle tokens often contain redundant intermediate steps, exploratory reasoning paths, or detailed calculations that become less relevant as reasoning progresses toward the final solution. Further analysis of attention patterns supporting this design is provided in Appendix C, including a concrete example demonstrating the redundancy present in typical reasoning traces in Appendix B.

(a) Alternative methods comparison
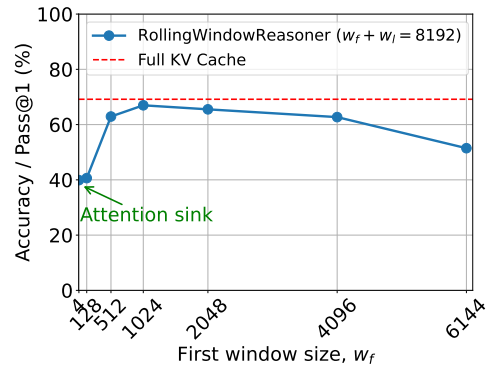
(b) First window size ablation

Figure 2: Method validation on AIME25 using Phi-4-reasoning-plus. (a) Comparison of Rolling-WindowReasoner against alternative KV-cache selection methods. Our simple approach performs competitively with more complex selection strategies including AttentionWeightedScoreMiddleWindow (AttentinoMW). (b) Ablation study on first window size with total budget of 8192 tokens. Performance increases significantly from 4 tokens (Attention Sink) to 1024 tokens, demonstrating the importance of preserving complete problem context for reasoning tasks.

3 Experiments

3.1 Experimental Setup

We evaluate RollingWindowReasoner on two state-of-the-art reasoning model families: Qwen3 (4B, 14B parameters) [15] and Phi-4-reasoning-plus (14B parameters) [1]. Our evaluation spans five challenging benchmarks requiring multi-step reasoning across three diverse domains: math reasoning (AIME24/25, HMMT25), code generation (LiveCodeBench [7]), and academic QA (GPQA [10]).

We compare our method against several baselines and ablations:

- Full KV Cache: Standard attention over complete sequence.
- AttentionWeightScoredMiddleWindow: Selects middle tokens based on exponential moving averages of attention weights.
- StridedMiddleWindow: Selects middle tokens with fixed stride sampling.
- RandomMiddleWindow: Randomly samples middle tokens.
- ConcatWindow: Generates window-by-window by concatenating the first and last window of tokens each time.

All MiddleWindow methods maintain the same first window structure as RollingWindowReasoner while allocating part of the last window budget to middle token selection, differing in their selection strategies (see Appendix D for implementation details).

3.2 Results

We first compare RollingWindowReasoner against alternative KV-cache selection methods on AIME25 using Phi-4-reasoning-plus to validate our design choices. As shown in Figure 2a, our method achieves competitive performance while being significantly simpler than alternative selection strategies. Specifically, RollingWindowReasoner performs similarly to AttentionWeightScored-MiddleWindow, demonstrating that simple temporal recency is as effective as more sophisticated selection criteria. The ConcatWindow method shows substantially worse performance while requiring additional computation due to repeated forward passes. Even though ConcatWindow uses position embeddings that indicate that middle tokens were discarded, it loses information about middle tokens by regenerating the KV-cache for each window.

Our comprehensive evaluation results across three reasoning domains show that across all domains and model sizes, RollingWindowReasoner maintains competitive performance with substantially reduced KV-cache budgets. As shown in Figure 1, for math reasoning tasks (AIME25, AIME24, HMMT25), our method achieves near-optimal performance with 50% cache reduction. Code generation on LiveCodeBench demonstrates similar efficiency gains, with performance plateauing at moderate cache budgets. Academic QA tasks on GPQA show consistent improvements, validating that our two-window design generalizes across diverse reasoning requirements. Additional experiments on simpler MATH dataset, memory efficiency, and more model families are presented in the Appendix A.

Our ablation study on first window size reveals a critical difference from prior work. While Attention Sink [14] uses only 4 initial tokens, we find that reasoning tasks require substantially larger first windows to preserve complete problem context. Figure 2b shows that performance increases significantly from 4 to 1024 tokens in the first window, with diminishing returns beyond this point. This finding highlights the importance of selective attention based on redundancy analysis of reasoning traces. It is important to preserve problem statements and instructions in their entirety, rather than simply maintaining a few irrelevant tokens to allocate useless attention weights for generating sequences that are longer than training fluently.

4 Related works

Attention Sink. The Attention Sink method [14] bears similarity to our approach but differs in motivation and design. Attention Sink maintains the first few irrelevant tokens (typically 4) to allocate useless attention weights, enabling generation longer than training sequences. In contrast, we target faster reasoning and exploit the redundancies of reasoning traces to remove middle tokens,

using substantially larger first windows (1024 tokens) to preserve complete problem statements and reasoning context.

Sparse Attention.   Various sparse attention mechanisms [9] have been proposed to reduce transformer quadratic complexity, including sparse factorizations [3], locality-sensitive hashing [8], random feature approximations [4], and sliding window patterns [2, 16]. However, these methods primarily target long input contexts rather than long generation sequences and involve complex attention designs. We find that a simple two-window approach effectively accelerates reasoning by exploiting inherent redundancies in reasoning traces.

5   Conclusion

We presented RollingWindowReasoner, a simple yet effective method for reducing the computational overhead of test-time reasoning while maintaining solution quality. By preserving only the first window (problem context) and last window (recent reasoning) of KV caches, our method achieves $4\times$ compute reduction with 50% cache budget at equivalent performance. Our evaluation across multiple model families and reasoning tasks demonstrates that redundancy in reasoning chains is a general phenomenon that can be systematically exploited, opening avenues for making advanced reasoning models more practical and accessible.

A   Additional Experimental Results

Performance on Simpler Mathematical Problems.   To validate our approach across varying difficulty levels, we evaluated RollingWindowReasoner on a subset of MATH500 containing simpler mathematical problems. As shown in Figure 3, the results demonstrate consistent performance gains, achieving comparable accuracy with approximately 50% of the original KV-cache budget. This finding suggests that our method's effectiveness extends beyond highly challenging reasoning tasks to encompass a broader range of mathematical problem-solving scenarios.
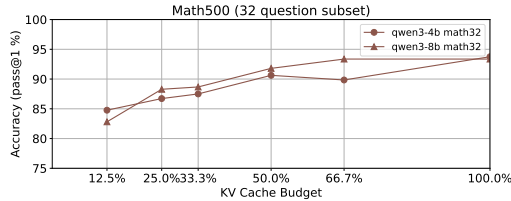

Figure 3: RollingWindowReasoner performance on easier mathematical problems.

Memory Efficiency Analysis.   Hardware constraints present significant challenges for deploying large reasoning models. Standard configurations of Phi-4-reasoning-plus with maximum token limits of 64K consistently produce out-of-memory errors on 40-GB A100 GPUs when using both vLLM and Hugging Face Transformers implementations. In contrast, RollingWindowReasoner with an 8K token budget operates seamlessly within these memory constraints, demonstrating substantial practical benefits for resource-limited deployment environments.

Extended Model Family Evaluation.   To assess the generalizability of our approach, we conducted additional experiments using DeepSeek-Llama-8B and Nemotron-4B models. As shown in Figure 4, the results confirm consistent performance improvements across these architectures. Performance metrics differ from officially reported benchmarks due to variations in evaluation frameworks and prompt engineering approaches employed in our study.

B   Redundancy in Reasoning Traces

To illustrate the redundancy present in reasoning chains, we provide an example of a typical reasoning trace generated by Phi-4-reasoning-plus. The trace demonstrates how models frequently generate
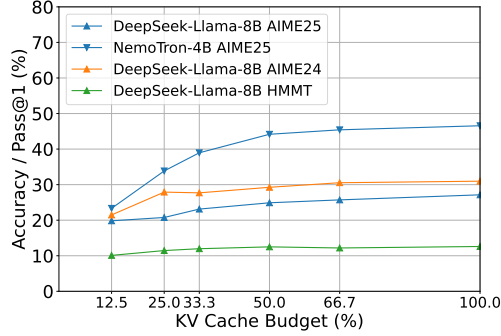
Figure 4: RollingWindowReasoner performance across additional model families including DeepSeek-Llama-8B and NemoTron-4B on AIME25, AIME24, and HMMT25 benchmarks. Results demonstrate consistent efficiency gains across diverse model architectures, with performance plateauing at 50% cache budget across all tested models and tasks.

repetitive verification steps, self-corrections, and exploratory paths that do not contribute meaningfully to the final solution. The reasoning process includes numerous instances of "wait", "re-check", and iterative refinements that, while part of the model's thinking process, become less relevant as reasoning progresses toward the solution.

```
<|im_start|>system<|im_sep|>You are Phi, a language model trained by
Microsoft to help users. Your role as an assistant involves thoroughly
exploring questions through a systematic thinking process before providing
the final precise and accurate solutions. This requires engaging in a
comprehensive cycle of analysis, summarizing, exploration, reassessment,
reflection, backtracing, and iteration to develop well-considered thinking
process. Please structure your response into two main sections: Thought and
Solution using the specified format: <think> {Thought section} </think> {
Solution section}. In the Thought section, detail your reasoning process in
steps. Each step should include detailed considerations such as analysing
questions, summarizing relevant findings, brainstorming new ideas, verifying
 the accuracy of the current steps, refining any errors, and revisiting
previous steps. In the Solution section, based on various attempts,
explorations, and reflections from the Thought section, systematically
present the final solution that you deem correct. The Solution section
should be logical, accurate, and concise and detail necessary steps needed
to reach the conclusion. Now, try to solve the following question through
the above guidelines:<|im_end|><|im_start|>user<|im_sep|>There are several
bags of apples. Each bag contains either 12 or 6 apples. How many apples are
 there in total, if it is known that the total number of apples is not less
than 70 and not more than 80? List all possible options. Put your answer in
\boxed{}. For multiple-choice questions, only put the choice letter in the
box.<|im_end|><|im_start|>assistant<|im_sep|><think>We are asked: "There are
 several bags of apples. Each bag contains either 12 or 6 apples. How many
apples are there in total, if it is known that the total number of apples is
 not less than 70 and not more than 80? List all possible options. Put your
answer in \boxed{}. For multiple-choice questions, only put the choice
letter in the box."

I need to parse: There are several bags of apples. Each bag contains either
12 or 6 apples. The total number of apples is not less than 70 and not more
than 80. We must list all possible options. But then it says: "For multiple-
choice questions, only put the choice letter in the box." It might be
ambiguous. Perhaps they want us to output the result as a letter, like "A, B
, ..." But since question is ambiguous.
```

Let's analyze: Let x be number of bags that contain 12 apples, and y be
number of bags that contain 6 apples. Total apples is 12x + 6y. And we want
70 <= total apples <= 80. Also x, y are non-negative integers. Solve
possible solutions.

But note: We have "several bags", meaning there is at least one bag?
Possibly? But probably not, I assume x and y non-negative integers, possibly
 positive. But "several" means at least two bags, but not sure. Actually "
several" usually means more than one, but that is not crucial.

We want possible totals: 12x + 6y, with x,y non-negative integers, and the
sum is between 70 and 80 inclusive. Also, the number of bags can be any
count.

We need to solve equation: 12x + 6y = T, where T in [70,80]. We can factor
out 6: T = 6(2x + y). Let k = 2x + y, then T = 6k, so T must be a multiple
of 6. But wait, but then T must be one of 70,71,...80 that are multiples of
6? Multiples of 6 in range [70,80] are: 72, 78 (and maybe 66 is too low, 84
is too high, but check: 6*12=72, 6*13=78, 6*14=84 so only possibilities are
T=72 and T=78). But wait: Are these the only possible totals? Wait a minute:
 But then if T is not a multiple of 6, then there is no solution. But the
problem states: "Each bag contains either 12 or 6 apples" That implies total
 must be a linear combination of 12 and 6, which is 6*(2x + y). So total is
always a multiple of 6. So the possible total in [70,80] are only 72 and 78,
 because 66 is too low, 84 is too high. But wait, 70 is not a multiple of 6;
 72 is, 78 is, but 70, 71, 73, 74, etc. are not possible. So possible totals
 are 72 and 78. But are there additional conditions? The problem said "
several bags", so number of bags must be at least 2. But that doesn't matter
.

I need to check: if 72, then equation: 12x + 6y = 72, dividing by 6: 2x + y
= 12. x can be 0,1,... maybe x=0 gives y=12, x=1 gives y=10, x=2 gives y=8,
x=3 gives y=6, x=4 gives y=4, x=5 gives y=2, x=6 gives y=0. But then is 72
acceptable? Yes. And for 78, then equation: 12x + 6y = 78, dividing by 6: 2x
 + y = 13, so possible solutions: x=0 gives y=13, x=1 gives y=11, x=2 gives
y=9, x=3 gives y=7, x=4 gives y=5, x=5 gives y=3, x=6 gives y=1, and x=7
gives y = -1, not allowed. So yes.

Wait, check: There is a subtle point: if the bag has 12 apples or 6 apples,
then the total is always a multiple of 6. So any total that is not a
multiple of 6 is impossible. So we have 72 and 78.

But wait, note that the problem said "not less than 70 and not more than
80", so we include 70,71,...80. But then 72 and 78 are the only ones that
are multiples of 6.

I can also check if 84 is in the range? But 84 > 80, so no.

I might need to check if there is any other possibility: Possibly 6x + 12y?
That's the same as 12x +6y.

So the answer is: 72 and 78 apples. But let me check: Are there any other
possibility? What if we use 0 apples in one bag? But then x=0 gives 6y, and
then y must be between 70/6 ~ 11.67 and 80/6 ~ 13.33, so y can be 12 or 13.
But 13 gives 78, 12 gives 72. But if we allow x=1, then 2+ y = something. In
 any event, the totals are 72 and 78.

Now the problem says: "List all possible options." So answer: 72 apples or
78 apples.

But then "For multiple-choice questions, only put the choice letter in the
box." Possibly means: the answer is one of the multiple choice answers, but
the instructions indicate that if this is a multiple-choice question, then

we should only put the letter of the correct option in the box, not the answer details. But this instruction "For multiple-choice questions, only put the choice letter in the box" might be a note from the test system. But then the problem statement says "Put your answer in \boxed{}." But then the note "For multiple-choice questions, only put the choice letter in the box." implies that if this is a multiple-choice question then we only choose one letter. But maybe the problem is from a contest that sometimes asks the answer to be in a box. Alternatively, if we are listing multiple options, then maybe we put something like \boxed{72, 78}. But instructions "for multiple-choice questions" might be just additional instructions in the contest.

Let's check: "Put your answer in \boxed{}" implies the answer must be put in a box. But then "For multiple-choice questions, only put the choice letter in the box." This is contradictory. Possibly the answer is a letter. However, our answer is not a letter, but a list. But maybe it's something like "C: 72 and 78" if the multiple-choice options are given by letters.

Wait, let me re-read the problem statement: "There are several bags of apples. Each bag contains either 12 or 6 apples. How many apples are there in total, if it is known that the total number of apples is not less than 70 and not more than 80? List all possible options. Put your answer in \boxed{}. For multiple-choice questions, only put the choice letter in the box." The phrasing "For multiple-choice questions, only put the choice letter in the box" might be instructions for the contest "choose the correct letter if given multiple-choice answers." But if it's a free response, then we list the answer. Possibly the answer is "72 and 78". But let me check if any other combination is possible: Could 70 be possible? For total 70, equation: $12x+6y=70$, divide by 2: $6x+3y=35$, then $3(2x+y)=35$, so not possible since 35 not divisible by 3. Similarly, 71 is not divisible by 6. 72 is divisible by 6. 73 is not, 74 is not, 75 is not, 76 is not, 77 is not, 78 is divisible, 79 is not, 80 is not. So yes.

Thus the answer: 72 and 78 apples.

Thus answer: \boxed{72, 78}. But wait: "List all possible options." Possibly the answer should be "72 or 78" but they want the list. So I'll output: \boxed{72, 78}.

Wait, but then "For multiple-choice questions, only put the choice letter in the box." Possibly means if it's multiple-choice then if the correct answer is option B, then put \boxed{B}. But the problem is ambiguous.

I wonder if the problem is from the Romanian "math contest" where sometimes the answer is given with a letter. But instructions "for multiple-choice questions" is something I've seen in contests. But given our problem, there's no letter option provided. Possibly we should answer: \boxed{72 \text{ and } 78}. But then if it's multiple choice then we should put the letter corresponding to that option. But since the problem statement instructs "Put your answer in \boxed{}." it might be a requirement from the contest.

I could answer: \boxed{72, 78}.

But let me check if there is any possible possibility: $12x+6y=84$ would be 84 apples, but 84 is not in the range. $12x+6y=66$ would be 66 apples, but 66 is less than 70. So yes.

Thus answer: \boxed{72, 78}.

I will now produce final answer accordingly.

```
I'll now produce final answer: \boxed{72, 78}.</think>Let x be the number of
 bags with 12 apples and y be the number of bags with 6 apples. The total
number of apples is given by:

Total = 12x + 6y = 6(2x + y)

Since the total is between 70 and 80 (inclusive), and 6(2x + y) is a
multiple of 6, the possible totals in that range are 72 and 78.

For 72 apples:
6(2x + y) = 72 $\Rightarrow$ 2x + y = 12
There are many non-negative integer solutions for (x, y) (e.g., x = 0, y =
12; x = 1, y = 10; $\cdots$; x = 6, y = 0).

For 78 apples:
6(2x + y) = 78 $\Rightarrow$ 2x + y = 13
Again, there are many non-negative integer solutions for (x, y) (e.g., x =
0, y = 13; x = 1, y = 11; $\cdots$; x = 6, y = 1).

Thus, the only possible total numbers of apples are 72 and 78.

\boxed{72, 78}
```

C   Attention Pattern Analysis

Our analysis of attention patterns reveals systematic preferences in token importance based on positional relationships. Contemporary tokens consistently receive higher attention weights compared to distant historical tokens. Additionally, specific transformer layers demonstrate pronounced attention to tokens within the first window, confirming the importance of preserving initial context.

Figure 5 illustrates the attention mask structure employed by RollingWindowReasoner. For each new token generation, the model attends exclusively to the first window and the last window, effectively creating a sparse attention pattern that maintains critical contextual information while discarding redundant intermediate tokens.

Figure 6 presents empirical attention patterns observed in Qwen3-4B during reasoning tasks. While certain intermediate tokens exhibit elevated attention scores (visible as vertical patterns in attention visualizations), their significance diminishes rapidly with increased temporal distance from the current generation step. This decay pattern supports our hypothesis that maintaining these tokens in the KV-cache yields diminishing returns for computational investment, justifying their exclusion from our rolling window approach.

D   Experimental Implementation Details

D.1   Alternative Methods

AttentionWeightScoredMiddleWindow.   This approach maintains both first and last window tokens, allocating 1K tokens to the initial window and half the remaining budget to the final window. The method tracks exponential moving averages of attention weights for each key-value pair using the update rule $s_{t,l,h} = s_{t,l,h} \times \eta + \alpha_{t,l,h}^T \times (1 - \eta)$, where t, l, and h represent position, layer, and head indices respectively, and $\alpha_{t,l,h}^T$ denotes attention weights from the current timestep T to the specific key-value $k_{t,l,h}$. When cache capacity is exceeded, tokens with minimum attention weight scores are removed from the middle window across all heads and layers. The decay parameter $\eta = \exp(\log(0.5)/B)$, where B is the total budget, ensures appropriate temporal weighting without excessive decay.

StridedMiddleWindow.   This baseline preserves a 1K token first window and selects middle tokens using strided sampling with position_id % 50 == 0. Remaining budget allocation is assigned to the last window to maintain fair comparison conditions.
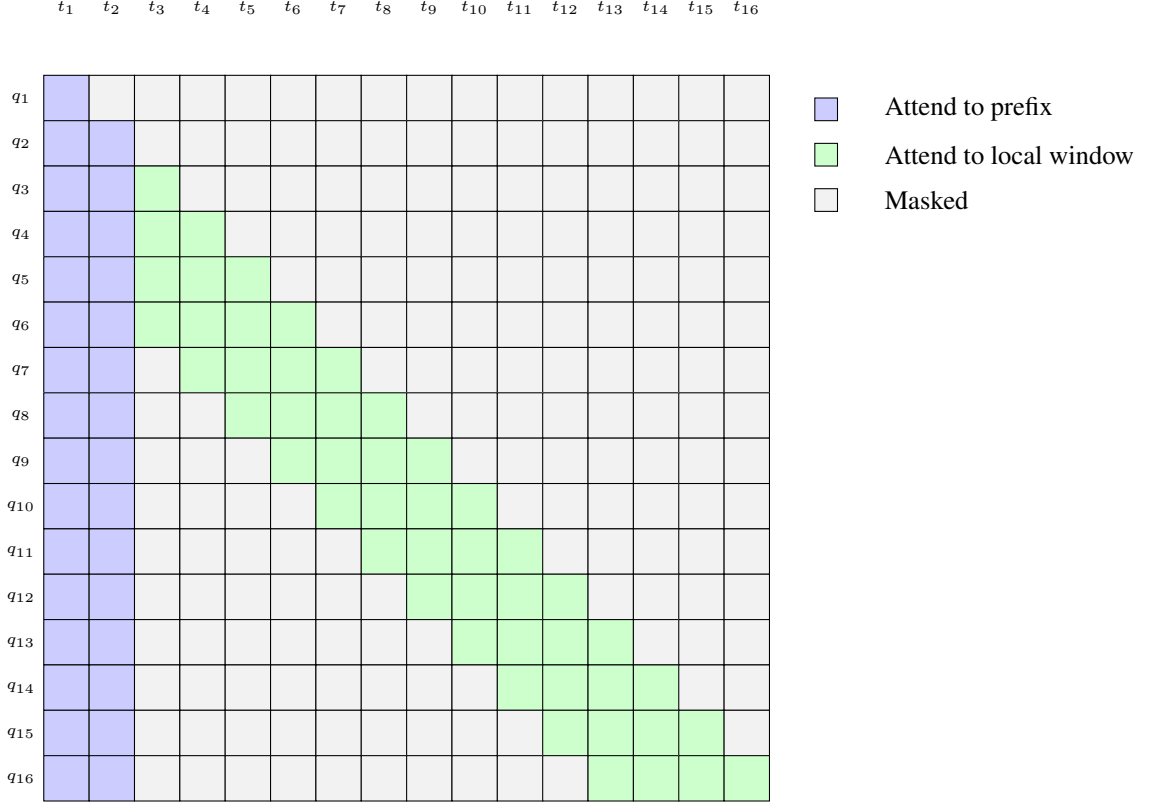
Figure 5: Attention mask pattern for RollingWindowReasoner. Blue regions indicate attention to the prefix window (first window), green regions show attention to the local sliding window (last window), and gray regions represent masked tokens. This sparse pattern preserves essential context while reducing computational complexity.

RandomMiddleWindow. Similar to the strided approach, this method maintains a 1K first window while randomly preserving middle tokens with probability 1/50. Unused budget capacity is allocated to the last window for consistent evaluation.

ConcatWindow. This approach generates content window-by-window, concatenating first and last window tokens to construct prompts for subsequent generation passes. Each generation cycle involves fresh forward passes with recomputed key-value caches. Position embeddings encode concatenation information and missing middle token positions, though no intermediate token information is preserved.

D.2 Benchmark Specifications and Evaluation Protocols

D.2.1 Math Reasoning

AIME24 and AIME25: The American Invitational Mathematics Examination represents a highly selective assessment administered annually since 1983. Eligibility requires top 5% performance on AMC 12 or top 2.5% performance on AMC 10 examinations. The 15-question, 3-hour format constrains responses to integers between 000 and 999 inclusive. Problem domains encompass elementary algebra, geometry, trigonometry, number theory, probability, and combinatorics–topics often extending beyond standard high school curricula. AIME24 and AIME25 designate problem sets from respective competition years.

HMMT25: The Harvard-MIT Mathematics Tournament, established in 1998, constitutes a biannual competition featuring autumn events at Harvard University and spring competitions at MIT. The February tournament presents significantly elevated difficulty compared to standard competitions
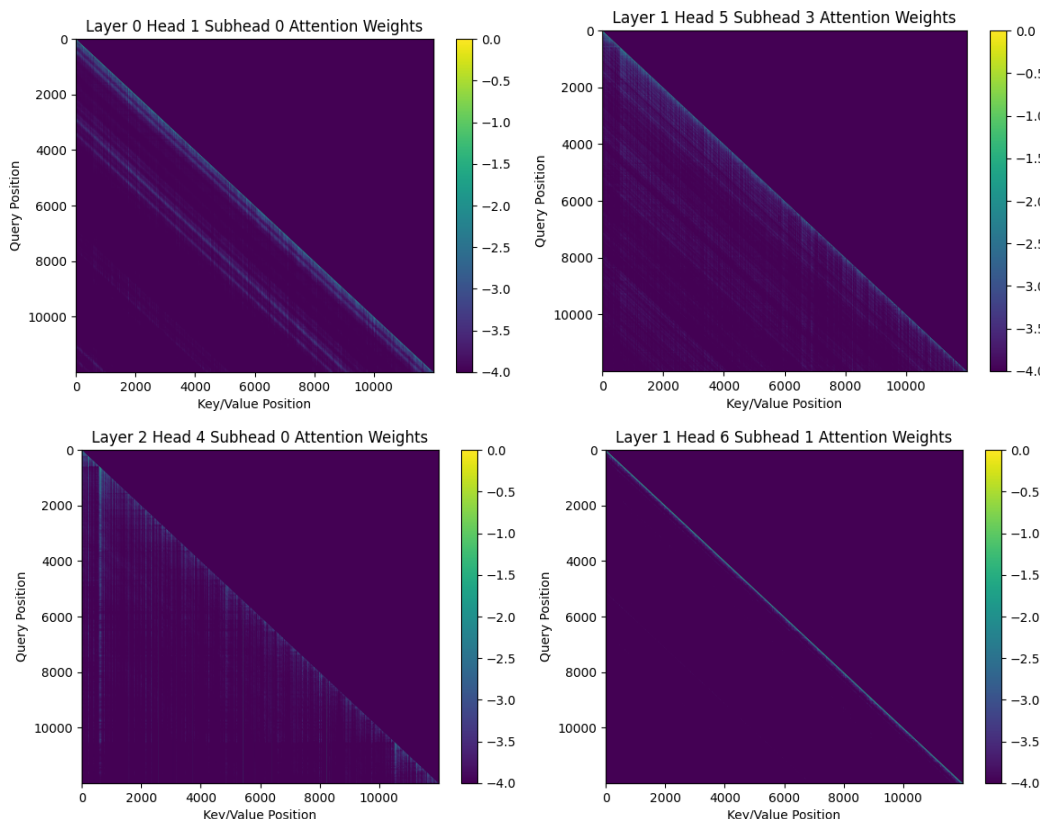
Figure 6: Representative attention patterns from Qwen3-4B across different layers and heads during reasoning tasks. Patterns demonstrate clear preferences for recent tokens (diagonal bands) and occasional attention to specific earlier tokens (vertical lines), with importance generally decaying with distance. These observations validate the effectiveness of our two-window approach.

like ARML, AIME, or Mandelbrot, often exceeding their challenge levels. Competition structure includes individual subject examinations in algebra, geometry, and combinatorics, complemented by collaborative team rounds.

MATH: This evaluation framework adapts the comprehensive MATH benchmark introduced in "Measuring Mathematical Problem Solving With the MATH Dataset." Our implementation randomly samples 32 representative problems from the underlying MATH dataset comprises 12,500 competition-level mathematical problems.

Evaluation protocols require models to format final answers within a boxed notation. Response verification employs pylatexenc and sympy libraries to assess mathematical equivalence with ground truth solutions. We deliberately avoided huggingface math_verify due to its tendency toward false positive classifications, particularly for problems involving equation repetition without substantive solution steps.

D.2.2 Code Generation

LiveCodeBench: This comprehensive evaluation framework assesses large language model coding capabilities using authentic programming challenges. The benchmark validates generated code correctness through extensive test case batteries spanning diverse programming domains and difficulty gradients. Our evaluation adheres to official LiveCodeBench protocols, emphasizing functional correctness over style or efficiency considerations.

11

D.2.3 Academic Question Answering

GPQA: Graduate-level Google-Proof Q&A presents a challenging multiple-choice assessment comprising 448 expert-authored questions spanning biology, physics, and chemistry domains. Questions target graduate-level comprehension and are specifically designed to resist simple web search resolution, hence the "Google-proof" designation. Our evaluation methodology mirrors mathematical reasoning protocols for consistency.

D.3 Model Configuration and Experimental Parameters

We adopt the recommended hyperparameters and prompts if provided for all models.

Microsoft/Phi-4-reasoning-plus: This 14-billion parameter architecture represents a specialized reasoning model derived from Microsoft's Phi-4 through supervised fine-tuning on chain-of-thought demonstration data, subsequently enhanced via reinforcement learning. The model architecture specifically targets complex reasoning scenarios including mathematical problem-solving, scientific inquiry, and programming challenges, leveraging additional inference-time computation for enhanced reasoning capabilities. Experimental configuration employed temperature=0.8, top_p=0.95, utilizing official system prompts and user message formatting specifications.

Qwen/Qwen3-*: These models constitute elements of Alibaba Cloud's Qwen3 architecture family, trained on approximately 36 trillion tokens spanning 119 languages and dialects. Evaluation employed temperature=0.6, top_p=0.95, top_k=20 parameters, with standard system message configurations deliberately omitted per recommended practices.

deepseek-ai/DeepSeek-R1-Distill-Llama-8B: This distilled reasoning architecture derives from the larger DeepSeek-R1 model family through knowledge distillation processes. The model underwent fine-tuning using Llama-3.1-8B as the base architecture, incorporating reasoning demonstrations generated by the full DeepSeek-R1 system. Evaluation parameters included temperature=0.6, top_p=0.95, with system message configurations omitted.

nvidia/Llama-3.1-Nemotron-Nano-4B-v1.1 This reasoning-optimized model derives from nvidia/Llama-3.1-Minitron-4B-Width-Base through NVIDIA's proprietary LLM compression methodologies, originating from Llama 3.1 8B architecture. Experimental configuration specified temperature=0.6, top_p=0.95, with system message "detailed thinking on" to activate reasoning modes.

E Example of Reasoning Traces generated by RollingWindowReasoner

E.1 Example of Phi-4-reasoning-plus (budget=8192)

Phi-4-reasoning-plus with RollingWindowReasoner (Budget=8192) solving a problem from AIME25: "The 9 members of a baseball team went to an ice-cream parlor after their game. Each player had a single scoop cone of chocolate, vanilla, or strawberry ice cream. At least one player chose each flavor, and the number of players who chose chocolate was greater than the number of players who chose vanilla, which was greater than the number of players who chose strawberry. Let N be the number of different assignments of flavors to players that meet these conditions. Find the remainder when N is divided by 1000.'"

```
We are told: There are 9 members, each player chooses one of three flavors:
chocolate, vanilla, strawberry. Conditions:
- Each flavor is chosen by at least one player.
- Let # of players who choose chocolate > # of players who choose vanilla >
# of players who choose strawberry.
- That is: Let counts be: c, v, s with c > v > s, c+v+s=9, each >=1.
We need count number of assignments of flavors to players, given these
conditions. We are told "The $9$ members" so each assignment is labeling
players with flavor assignments. But note, "players" are distinct. So we
```

count the number of ways to assign the flavors to players with the given counts, but the counts must satisfy conditions.

We want to count number N. Then find remainder mod 1000.

Let's denote count for chocolate, vanilla, strawberry be: c, v, s, with c > v > s, c+v+s = 9, c, v, s >=1.

Let's find all possible triplets satisfying c > v > s and c+v+s=9 with each >=1. Let's list possibilities.

We need s to be minimum at least 1, v must be at least s+1, c must be at least v+1. Let s=1, then v >=2, and c = 9 - (1+v) = 8 - v must be > v. So inequality: 8 - v > v => 8 > 2v, so v < 4, so possible v:2,3.

Case: s=1, v=2 then c=9 -3 =6. Check: 6>2>1. Valid.
Case: s=1, v=3 then c=9 -4 =5. Check: 5>3>1. Valid.

Now s=2, then v>=3, c = 9 - (2+v) = 7 - v. Must satisfy: 7-v > v => 7 > 2v, v < 3.5 so v=3 is possible, v=3 yields c=4, then check: 4>3>2 valid.
Now s=3, then v>=4, but then c = 9 - (3+v)=6 - v, and condition: 6-v > v => 6 > 2v, v < 3 so no possibility. So only solutions: (c,v,s) = (6,2,1), (5,3,1), (4,3,2). Also possibility: s=1, v=2, c=6, s=1, v=3, c=5, s=2, v=3, c=4.

Are there any others? Let's check s=1, v=4 then c=9 -5=4, but then c > v? 4>4 false. So no.

Now count ways: For each valid triplet, the number of assignments among players equals: number of ways to choose which players get chocolate, then among remaining choose those who get vanilla, and then strawberry. That is: 9!/(c! v! s!) but multiplied by symmetry? Wait, careful: The players are distinct. But the assignment is: choose c players for chocolate, then v players for vanilla, then remaining get strawberry. So the count is: (9 choose c)*(9-c choose v) = 9!/(c! v! s!). So for each triple.

Now compute for each triple:
......

Thus answer N = 2016, remainder mod 1000 = 16.

I worry: Problem says "9 members of a baseball team" so players are distinct. But then if players are distinct, assignments correspond to partitions multiplied by permutations among players.

Yes, that's correct.

Thus answer = 16 mod 1000.

I must check if we need to output the answer as \boxed{16} or just 16. But instructions: "Put your answer in \boxed{}." So answer: \boxed{16}.

However, instructions "For multiple-choice questions, only put the choice letter in the box." Possibly meaning if the answer is a letter, then only the letter. But in our case, answer is a number, so we output \boxed{16}.

However, wait: Our computed sum is 2016. But remainder mod 1000 is 16. But is there any possibility that there might be a mistake? Let's check systematically: Let a, b, c be numbers such that a > b > c, a+b+c=9, a, b, c >=1. Possibilities: (a, b, c) = (6,2,1), (5,3,1), (4,3,2). Check: (7,1,1) doesn't work because not strictly increasing. (5,2,2) doesn't work. (4,2,3) doesn't work because ordering doesn't hold.

Yes.

So answer = 2016 assignments, remainder mod 1000 = 16.

I check: 2016 mod 1000 = 16, yes.

Thus answer: \boxed{16}.

I must output answer as "16" in the box. But the instruction "For multiple-choice questions, only put the choice letter in the box." Possibly means if it is a multiple-choice letter answer, we only put letter? But our answer is a number? Wait, the problem says "Find the remainder when N is divided by 1000." So answer is a number, not a letter. So answer is just 16.

However, sometimes these problems are from AMC 8 2016? Actually, AMC 8 sometimes has answer letter choices like "A" etc. But instructions: "For multiple-choice questions, only put the choice letter in the box." I'm reading: "For multiple-choice questions, only put the choice letter in the box." That is ambiguous. But I'll assume answer is \boxed{16}.

Wait, but I must check: The instructions "Put your answer in \boxed{}" might be a specific formatting requirement from the contest. But then "For multiple-choice questions, only put the choice letter in the box." So maybe the answer letter is "D" if choices are A, B, C, D, E. But which letter corresponds to 16? Let me check: If the answer choices are: A. 16, B. 17, C. 18, D. 19, E. 20, then answer letter is A if 16 is first. But the problem did not provide choices.

Wait, the problem states "For multiple-choice questions, only put the choice letter in the box." That might be a separate instruction if the answer is a letter, but if not, then just the number is fine.

I must check: The original problem states "Find the remainder when N is divided by 1000." That remainder is a number, so answer is a number.

I'll assume the answer is just the number: 16.

Wait, but the answer is 16 mod 1000, but sometimes they want the remainder as a three-digit number? But 16 mod 1000 is 16.

Thus answer: \boxed{16}.

I'll now output answer in final message.

I'll now produce answer in plain text: \boxed{16}.

I'll now produce final answer.We begin by letting c, v, and s denote the numbers of players choosing chocolate, vanilla, and strawberry respectively. We are given:

  c + v + s = 9   (1)
  c > v > s   (2)
  c, v, s $\ge$ 1

Step 1. Find all (c, v, s) triples satisfying (1) and (2).

* If s = 1, then v must be at least 2.
  - For v = 2, c = 9 − (1 + 2) = 6. Check: 6 > 2 > 1.
  - For v = 3, c = 9 − (1 + 3) = 5. Check: 5 > 3 > 1.
  - For v = 4, c = 9 − (1 + 4) = 4, but 4 is not greater than 4.
* If s = 2, then v must be at least 3.
  - For v = 3, c = 9 − (2 + 3) = 4. Check: 4 > 3 > 2.
  - For v = 4, c = 9 − (2 + 4) = 3, but 3 is not greater than 4.

```
* If s = 3, then v must be at least 4, but then c would be 9 - (3 + v) $\le$
  2, which cannot be greater than v.

Thus the only valid distributions are:
  (6, 2, 1), (5, 3, 1), (4, 3, 2).

Step 2. Count the number of assignments for each distribution.

Since the 9 players are distinct, for a given triple (c, v, s) the number of
 assignments is
  Number = (9 choose c) $\times$ (9 - c choose v) = 9!/(c!$\cdot$v!$\cdots$!).

* For (6, 2, 1):
  Number = 9!/(6!$\cdot$2!$\cdot$1!) = 362880/(720$\times$2) = 362880/1440 =
252.
* For (5, 3, 1):
  Number = 9!/(5!$\cdot$3!$\cdot$1!) = 362880/(120$\times$6) = 362880/720 =
504.
* For (4, 3, 2):
  Number = 9!/(4!$\cdot$3!$\cdot$2!) = 362880/(24$\times$6$\times$2) =
362880/288 = 1260.

Step 3. Sum the assignments.

  Total N = 252 + 504 + 1260 = 2016.

Step 4. Find the remainder when N is divided by 1000.

  2016 mod 1000 = 16.

Thus, the final answer is:  \boxed{16}.
```

E.2 Example of Qwen3-4B (Budget=4096)

Qwen3-4B with RollingWindowReasoner (Budget=4096) solving a problem from AIME25: "Let S be the set of vertices of a regular 24-gon. Find the number of ways to draw 12 segments of equal lengths so that each vertex in S is an endpoint of exactly one of the 12 segments." (Abbreviated as generation is too long to fit here).

```
<think>
Okay, so I need to figure out the number of ways to draw 12 segments of
equal lengths in a regular 24-gon, such that each vertex is an endpoint of
exactly one segment. That means we're essentially looking for a perfect
matching of the 24 vertices with segments of equal length.

First, let me recall that a regular 24-gon has all sides equal and all
internal angles equal. The vertices are equally spaced around a circle.
Since we need to connect each vertex to exactly one other vertex with a
segment of equal length, the key is to figure out which lengths are possible
 for such segments.

In a regular polygon, the distance between two vertices depends on how many
edges apart they are. For example, adjacent vertices are connected by sides
of length equal to the side length of the polygon. If you skip one vertex,
the distance is longer, and so on. Since the polygon is regular, these
distances are determined by the chord lengths corresponding to different
step sizes.

So, in a regular n-gon, the possible chord lengths between two vertices are
determined by the number of steps between them. For a 24-gon, the possible
```

step sizes (i.e., the number of edges between two connected vertices) can be
 from 1 to 12. However, since connecting a vertex to another with step size
k is the same as connecting it with step size 24 - k (because of the polygon
's symmetry), so the unique chord lengths correspond to step sizes from 1 to
 12.

But wait, the problem says that all the segments must be of equal length. So
, all 12 segments must be chords of the same length. Therefore, we need to
find a way to pair up all 24 vertices with segments of the same length.

But here's the thing: for a given step size k (from 1 to 12), how many such
segments can we have? If we take step size k, each segment connects two
vertices with k steps apart. However, if we try to pair all 24 vertices with
 such segments, we need to make sure that the pairing is possible without
overlap.

But since the polygon has 24 vertices, which is even, we can pair them up.
However, for each step size k, the number of segments that can be drawn
without overlapping would depend on whether k is coprime with 24 or not.
Wait, maybe not exactly. Let me think.

If we fix a step size k, then the number of distinct chords of that length
is 24 / gcd(k, 24). Wait, is that right? Let me recall that in a regular n-
gon, the number of distinct chords of length corresponding to step size k is
 n / gcd(n, k). But since we are considering step sizes from 1 to n/2,
because beyond n/2, they repeat. So for n=24, step sizes from 1 to 12.

But if we want to pair all vertices with segments of the same length, then
we need that the step size k is such that the graph formed by connecting
each vertex to the one k steps away is a union of cycles, and we can
partition the graph into perfect matchings.

Wait, maybe I need to think about the structure of the graph. If we connect
each vertex to the one k steps away, then the graph is a collection of
cycles. The number of cycles would be gcd(k, 24). Each cycle has length 24 /
 gcd(k, 24). Therefore, for the graph to be a perfect matching, each cycle
must have even length, so that we can pair up the vertices. Therefore, 24 /
gcd(k, 24) must be even. Therefore, gcd(k, 24) must divide 24 and 24 / gcd(k
, 24) must be even. Therefore, gcd(k, 24) must be a divisor of 24 such that
24 / d is even. Therefore, d must be a divisor of 24 where d divides 24 and
24/d is even. Therefore, d must be such that d divides 24 and d is 12, 8, 6,
 4, 3, 2, or 1? Wait, let me check.

Wait, 24 divided by d must be even. Therefore, 24/d is even implies that d
divides 24 and 24/d is even. Therefore, 24/d = 2m, so d = 24/(2m) = 12/m.
Therefore, m must be a divisor of 12. Therefore, m can be 1, 2, 3, 4, 6, 12.
 Therefore, d = 12/m. Therefore, d can be 12, 6, 4, 3, 2, 1. Therefore, the
possible values of d (which is gcd(k,24)) are 1, 2, 3, 4, 6, 12. Therefore,
the step sizes k must satisfy that gcd(k,24) is one of these values.

But since k is between 1 and 12, we can check for each k, what is gcd(k,24):

For k=1: gcd(1,24)=1

k=2: gcd(2,24)=2

k=3: gcd(3,24)=3

k=4: gcd(4,24)=4

k=5: gcd(5,24)=1

k=6: gcd(6,24)=6

k=7: gcd(7,24)=1

k=8: gcd(8,24)=8

k=9: gcd(9,24)=3

k=10: gcd(10,24)=2

k=11: gcd(11,24)=1

k=12: gcd(12,24)=12

So, for k=1,2,3,4,6,8,12, the gcd(k,24) is 1,2,3,4,6,8,12. However, for the cycle length to be even, we need that 24 / d is even. Let's check for each d :

d=1: 24/1=24, which is even. So okay.

d=2: 24/2=12, even. Okay.

d=3: 24/3=8, even. Okay.

d=4: 24/4=6, even. Okay.

d=6: 24/6=4, even. Okay.

d=8: 24/8=3, which is odd. Wait, 3 is odd. Therefore, if d=8, then cycle length is 3, which is odd. Therefore, we cannot have a perfect matching here . Therefore, step size k=8 would result in cycles of length 3, which cannot be partitioned into perfect matchings. Therefore, we can't have step size 8. Similarly, for d=12: 24/12=2, which is even. Therefore, okay.

Therefore, the possible step sizes k are those for which gcd(k,24) is 1,2,3,4,6,12. However, for k=8, gcd(k,24)=8, which leads to cycle length 3, which is odd, so that's invalid. Therefore, step sizes k=1,2,3,4,6,12. Wait, but k=8 is excluded. Let me check:

For k=8: step size 8. Since 24/gcd(8,24)=24/8=3. So cycles of length 3. Since 3 is odd, you can't have perfect matchings. Therefore, step size 8 is invalid. So step sizes allowed are k=1,2,3,4,6,12. Wait, but for k=12, gcd (12,24)=12, so cycle length is 24/12=2. So cycles of length 2, which are just edges. So that's okay.

Therefore, the possible step sizes are k=1,2,3,4,6,12. However, the problem says that all 12 segments must be of equal length. Therefore, we need to choose one of these step sizes, and then find the number of perfect matchings for that step size.

Wait, but the question says "the number of ways to draw 12 segments of equal lengths". So, it's not that we have to choose a step size and count the number of perfect matchings for that step size, but rather sum over all possible step sizes? Wait, but the problem says "segments of equal lengths", so all 12 segments must be of the same length. Therefore, we need to consider each possible step size (k) where it's possible to have a perfect matching, and count the number of such perfect matchings for each k, then sum them all.

But wait, but maybe there is only one possible step size? Wait, but for example, step size 1: connecting adjacent vertices. However, if we try to make a perfect matching with step size 1, meaning connecting each vertex to its neighbor. But in that case, the graph would be a union of cycles. For step size 1, the graph is a 24-gon, which is a single cycle of length 24. To

17

have a perfect matching, we need to pair up the vertices. However, a cycle of even length can be decomposed into perfect matchings. But in this case, the question is about a single perfect matching, not multiple. Wait, but if we have a single perfect matching, then we can't have all segments of length 1. Because if you connect each vertex to its neighbor, you would have a cycle, but a perfect matching is a set of edges with no overlaps. However, in a 24-gon, if you take a perfect matching with step size 1, that would require that each vertex is connected to its neighbor, but that would require that the matching is actually the entire set of edges, but that would be 12 edges, but in a 24-gon, if you connect every other vertex, you get a perfect matching. Wait, no. Wait, confusion here.

Wait, maybe I need to clarify. For step size k, if we connect each vertex to the one k steps away, then the graph is a union of cycles. For example, if $k=1$, then it's a single cycle of length 24. If $k=2$, then it's a union of cycles. Let me recall that in general, when you connect each vertex to the one k steps away in an n-gon, the number of cycles is $\gcd(n,k)$, each of length $n / \gcd(n,k)$. Therefore, for $n=24$ and $k=1$, $\gcd(24,1)=1$, so one cycle of length 24. For $k=2$, $\gcd(24,2)=2$, so two cycles of length 12. For $k=3$, $\gcd(24,3)=3$, three cycles of length 8. For $k=4$, $\gcd(24,4)=4$, four cycles of length 6. For $k=6$, $\gcd(24,6)=6$, six cycles of length 4. For $k=12$, $\gcd(24,12)=12$, twelve cycles of length 2.

So, for each k, we have cycles. Now, in order to have a perfect matching, we need that each cycle can be decomposed into perfect matchings. However, since we need a single perfect matching (i.e., a set of edges such that each vertex is in exactly one edge), we need that the entire graph (which is a union of cycles) can be decomposed into a perfect matching. However, for a single cycle of length $2m$, there are two perfect matchings (alternating edges). But if the graph is a union of multiple cycles, then we need to choose a perfect matching for each cycle and combine them.

But the problem is, when you have multiple cycles, how many perfect matchings are there in total?

......

For example, take $n=4$, as before. The total should be 3, which matches 2 (for $k=1$) + 1 (for $k=2$) = 3. Correct.

For $n=6$, what is the answer?

For $n=6$, step sizes from 1 to 3.

$k=1$: $\gcd(1,6)=1$, cycles=1, length 6. Number of perfect matchings: 2.

$k=2$: $\gcd(2,6)=2$, cycles=2, length 3. Each cycle is length 3 (odd), so 0.

$k=3$: $\gcd(3,6)=3$, cycles=3, length 2. Each cycle is length 2, so 1 perfect matching. Therefore, total is 2 + 0 + 1 = 3.

But for $n=6$, the number of ways to draw 3 segments of equal length. For step size 1: 2 ways. For step size 3: 1 way. Total 3. Correct.

Therefore, the formula works. Therefore, for $n=24$, the answer is 113.

Therefore, I think the answer is \boxed{113}.

Final Answer
\boxed{113}

To solve the problem, we are given a regular 24-gon and asked to find the number of ways to draw 12 segments of equal length such that each vertex is an endpoint of exactly one segment.

Key Observations:

1. **Structure of the Polygon**: A regular 24-gon has 24 vertices, and the number of segments that can be drawn between them is determined by the step size between vertices. In a regular polygon, the length of a chord is determined by how many vertices are skipped between the endpoints.

2. **Step Sizes**: In a 24-gon, the possible step sizes (i.e., the number of vertices between the endpoints of a chord) are from 1 to 12. For each step size $ k $, we can draw chords connecting each vertex to the one $ k $ steps away.

3. **Perfect Matching Requirement**: We need to select 12 such chords (segments) of the same length, such that each vertex is used exactly once. This is equivalent to finding a perfect matching in the graph where each vertex is connected to others at step size $ k $.

4. **Graph Structure**: For a given step size $ k $, the graph formed by connecting each vertex to the one $ k $ steps away is a union of cycles. The number of such cycles is $ \gcd(k, 24) $, and each cycle has length $ \frac{24}{\gcd(k, 24)} $.

5. **Perfect Matchings in a Cycle**: A cycle of even length can be partitioned into a perfect matching in $ 2 $ ways. A cycle of odd length cannot be partitioned into a perfect matching.

Strategy:

- For each step size $ k $ from 1 to 12, determine the number of perfect matchings in the graph formed by connecting vertices with step size $ k $.
- The number of perfect matchings is:
  - $ 2^{\gcd(k, 24)} $, if the length of each cycle is even.
  - 0, if the length of each cycle is odd.

- The total number of such perfect matchings across all step sizes is the answer.

Calculation:

Let's compute for each $ k = 1 $ to $ 12 $:

$ k $	$ \gcd(k, 24) $	Cycle Length	Even?	# of Matchings
1	1	24	Yes	$ 2^1 = 2 $
2	2	12	Yes	$ 2^2 = 4 $
3	3	8	Yes	$ 2^3 = 8 $
4	4	6	Yes	$ 2^4 = 16 $
5	1	24	Yes	$ 2^1 = 2 $
6	6	4	Yes	$ 2^6 = 64 $
7	1	24	Yes	$ 2^1 = 2 $
8	8	3	No	0
9	3	8	Yes	$ 2^3 = 8 $
10	2	12	Yes	$ 2^2 = 4 $

```
| 11      | 1                    | 24            | Yes    | $ 2^1 = 2 $ |
| 12      | 12                   | 2             | Yes    | $ 2^1 = 1 $ |

Summing up the matchings:

$$
2 + 4 + 8 + 16 + 2 + 64 + 2 + 0 + 8 + 4 + 2 + 1 = \boxed{113}
$$

---

### Final Answer:

$$
\boxed{113}
$$
```

E.3   Example of Qwen3-14B (budget=4096)

Qwen3-14B with RollingWindowReasoner (Budget=4096) solving a problem from LiveCodeBench: "You are given an array original of length n and a 2D array bounds of length n x 2, where bounds[i] = [u_i, v_i]. You need to find the number of possible arrays copy of length n such that: (copy[i] - copy[i - 1]) == (original[i] - original[i - 1]) for 1 <= i <= n - 1. u_i <= copy[i] <= v_i for 0 <= i <= n - 1. Return the number of such arrays."

```
<think>
Okay, let's see. The problem is to find the number of possible arrays 'copy'
 that satisfy two conditions. First, the differences between consecutive
elements must match the original array's differences. Second, each element
in the copy array must be within its corresponding bounds.

Hmm. So, the first thing I need to understand is how the differences work.
For example, if original is [1,2,3,4], then the differences between
consecutive elements are 1, 1, 1. So the copy array must have the same
differences. That means the copy array is determined by its first element,
right? Because once the first element is chosen, all other elements are
fixed based on the original's differences.

So for example, if the original array has differences d1, d2, ..., dn-1,
then the copy array must have the same differences. So copy[0] can be any
value, then copy[1] = copy[0] + d1, copy[2] = copy[1] + d2, etc. So the
entire array is determined by the first element.

Wait, but the original array's differences are fixed. So for the copy array,
 the differences between consecutive elements must be exactly the same as
original's. So the copy array is a sequence that starts with some value,
then each next element is previous plus the original's difference.

So the problem reduces to finding all possible starting values for the copy
array such that each element in the copy array is within the corresponding
bounds.

So the approach is:

1. Compute the differences between consecutive elements in original. Let's
call this diffs array. For example, original = [1,2,3,4], diffs is [1,1,1].

2. Then, for the copy array, the first element can be any value x, and then
each subsequent element is x + d1, x + d1 + d2, etc. So for each position i
in the copy array, the value is x + sum_{k=0 to i-1} diffs[k]. Wait, no. Let
```

me think again. Let's say original's differences are $d_0, d_1, ..., d_{n-2}$ (since original has length n). Then, for the copy array, copy[0] is x. copy[1] = x + d_0. copy[2] = x + d_0 + d_1. And so on. So for each position i in the copy array, the value is x + (sum of the first i-1 differences). Wait, for i=0, it's x. For i=1, it's x + d_0. For i=2, x + d_0 + d_1. So for each position i, the value is x + (sum of diffs[0..i-1]).

So for each position i, the value of copy[i] is x + prefix_sum[i], where prefix_sum[i] is the sum of the first i differences. Wait, but for i=0, the prefix sum is zero? Or maybe I need to think of prefix_sum as starting from zero. Let me think:

Let me compute prefix_sums as follows. Let's say prefix_sums[0] = 0, prefix_sums[1] = d_0, prefix_sums[2] = d_0 + d_1, etc. Then, for the copy array, copy[i] = x + prefix_sums[i]. Because for i=0, it's x + 0. For i=1, x + prefix_sums[1] = x + d_0. Yes. So the prefix_sums array is of length n, where prefix_sums[0] = 0, prefix_sums[1] = d_0, prefix_sums[2] = d_0 + d_1, ..., prefix_sums[n-1] = sum of all diffs (d_0 to d_{n-2}).

So, for each i in 0..n-1, the copy[i] must be between u_i and v_i (the bounds[i][0] and bounds[i][1]). So for each i, x + prefix_sums[i] must be between u_i and v_i. So x must satisfy for all i:

u_i <= x + prefix_sums[i] <= v_i.

Which can be rewritten as:

u_i - prefix_sums[i] <= x <= v_i - prefix_sums[i].

So for all i, x must be in the interval [a_i, b_i], where a_i = u_i - prefix_sums[i], and b_i = v_i - prefix_sums[i].

So the problem reduces to finding the number of x's that are in the intersection of all these intervals [a_i, b_i], for i from 0 to n-1.

So the solution steps are:

1. Compute the diffs array from original.

2. Compute the prefix_sums array. Let me think: for original of length n, the diffs array is of length n-1. Then, prefix_sums is an array of length n, where prefix_sums[0] = 0, prefix_sums[1] = diffs[0], prefix_sums[2] = diffs[0] + diffs[1], etc.

3. For each i in 0..n-1, compute a_i = bounds[i][0] - prefix_sums[i], and b_i = bounds[i][1] - prefix_sums[i].

4. The x must be in the intersection of all intervals [a_i, b_i]. The number of such x is the number of integers in the intersection of all these intervals.

But wait, the problem says that x can be any integer? Wait, no. The problem says that copy[i] must be between u_i and v_i. But the original array's elements can be any integers, but the copy array's elements must be integers? Or are they allowed to be real numbers?

Wait, the problem says that original is an array of integers, and the bounds are integers. The problem says that copy is an array of integers? Because the input is given as integers. But the problem statement says "number of possible arrays copy of length n" -- but the problem is probably considering that the elements are integers. Because in the examples, the copy arrays are integers.

Wait, the problem says that the original array is given with integers, and the bounds are integers. So the copy array must be an array of integers. Because otherwise, the number of possible arrays would be infinite. But in the examples, the output is finite. So the problem must assume that the copy array elements are integers. Therefore, the x must be an integer, and all the constraints must be satisfied for integers.

So the problem is to find the number of integers x that satisfy for all i, $a_i \le x \le b_i$. So the intersection of all these intervals. The answer is the number of integers in the intersection.

So the steps are:

Compute the prefix_sums array.

Compute for each i, a_i and b_i. Then find the maximum of all a_i's and the minimum of all b_i's. The number of integers between max_a and min_b, inclusive, is the answer. But if max_a > min_b, then there are zero possibilities.

So the answer is max(0, min_b - max_a + 1).

But wait, how to compute this?

Let me think with example 1:

Original = [1,2,3,4], so diffs is [1,1,1]. prefix_sums is [0, 1, 2, 3].

Then for each i:

bounds are [[1,2], [2,3], [3,4], [4,5]]

For i=0:

$a_0 = 1 - 0 = 1$, $b_0 = 2 - 0 = 2$.

i=1: $a_1 = 2 - 1 = 1$, $b_1 = 3 - 1 = 2$.

i=2: $a_2 = 3 - 2 = 1$, $b_2 = 4 - 2 = 2$.

i=3: $a_3 = 4 - 3 = 1$, $b_3 = 5 - 3 = 2$.

So the max_a is 1, min_b is 2. So the number of x is 2 (1 and 2). Which matches the example.

Another example, example 2:

original = [1,2,3,4], bounds are [[1,10], [2,9], [3,8], [4,7]]

prefix_sums are same as before: [0,1,2,3]

For each i:

i=0: a=1 - 0 = 1, b=10 -0 = 10.

i=1: a=2 -1 =1, b=9-1=8.

i=2: a=3-2=1, b=8-2=6.

i=3: a=4-3=1, b=7-3=4.

So max_a is 1, min_b is 4. So the x can be 1, 2, 3, 4. So 4 options. Which matches the example.

Third example: original = [1,2,1,2], bounds [[1,1],[2,3],[3,3],[2,3]]

Let's compute the diffs. original is [1,2,1,2]. So the diffs are [1, -1, 1].

prefix_sums is [0, 1, 0, 1].

Now for each i:

i=0: bounds are [1,1]. a_i = 1 - 0 = 1. b_i = 1 - 0 =1. So x must be between 1 and 1.

i=1: bounds are [2,3]. a_i = 2 - prefix_sums[1] = 2 - 1 =1. b_i is 3 - 1=2.

i=2: bounds are [3,3]. a_i is 3 - prefix_sums[2] = 3 - 0 =3. b_i is 3-0=3. So x must be 3.

i=3: bounds are [2,3]. a_i is 2 - prefix_sums[3] = 2 -1 =1. b_i is 3-1=2.

So the max_a is max(1,1,3,1) =3. min_b is min(1,2,3,2) =1. So since 3>1, no solution. Hence answer 0. Which matches the example.

So the approach is correct.

Now, the steps to implement:

First, compute the diffs array. Then compute the prefix_sums array.

Then compute for each i, the a_i and b_i. Then compute the maximum a_i and the minimum b_i. Then if max_a > min_b, return 0. Else, return min_b - max_a +1.

But how to compute the prefix_sums?

Let me think. For original array, the differences between consecutive elements. For example, original is [o0, o1, o2, ..., on-1]. Then, diffs is [o1 - o0, o2 - o1, ..., on-1 - on-2]. Then, the prefix_sums array is of length n. prefix_sums[0] = 0. prefix_sums[1] = diffs[0]. prefix_sums[2] = diffs[0] + diffs[1], etc. So for i in 0 to n-1, prefix_sums[i] is the sum of the first i-1 diffs. Wait, no. Let's think:

prefix_sums[0] = 0.

prefix_sums[1] = diffs[0]

prefix_sums[2] = diffs[0] + diffs[1]

prefix_sums[3] = diffs[0] + diffs[1] + diffs[2]

So for i-th element in the original array (0-based), the prefix_sums[i] is the sum of the first i-1 diffs. Because for the first element (i=0), it's 0. For i=1, it's sum of first 0 diffs (diffs[0] is the first difference, but for the first element, the prefix is zero). Wait, maybe it's better to think of the prefix_sums as:

prefix_sums[0] = 0.

prefix_sums[1] = original[1] - original[0]

prefix_sums[2] = original[1] - original[0] + original[2] - original[1] = original[2] - original[0]

Wait, but the way the copy array is built is that copy[i] = x + prefix_sums[i], where prefix_sums[i] is the sum of the first i differences. Wait, no. Let me think again.

Original's differences are d0, d1, d2, ... dn-2. So for the copy array, copy[0] = x. copy[1] = x + d0. copy[2] = x + d0 + d1. copy[3] = x + d0 +d1 +d2. So for copy[i], the value is x + sum_{k=0 to i-1} d_k. So for i-th element, the sum is the sum of first i differences? No, the sum of first i-1 differences. Because for i=0, it's x + 0. For i=1, it's x + sum of first 0 differences (d0 is the first difference). Wait, no. For i=1, the sum is d0. For i=2, sum is d0 +d1. So for the i-th element in the copy array, the sum is sum of first (i-1) differences. So the prefix_sums array is of length n, where prefix_sums[i] is the sum of the first i-1 differences. So how to compute that?

Let me think. The prefix_sums can be computed as follows:

prefix_sums = [0] * n

for i in 1 to n-1:

prefix_sums[i] = prefix_sums[i-1] + diffs[i-1]

Yes. Because for i=1, it's prefix_sums[1] = prefix_sums[0] + diffs[0] = 0 + d0. For i=2, prefix_sums[2] = prefix_sums[1] + diffs[1] = d0 +d1. So that's correct.

So the code steps:

Compute the diffs array. For original, for i from 1 to len(original)-1, compute original[i] - original[i-1]. So diffs is [original[1]-original[0], original[2]-original[1], ...].

Once we have diffs, compute the prefix_sums array of length n, where prefix_sums[0] = 0, and prefix_sums[i] = prefix_sums[i-1] + diffs[i-1] for i >=1.

Once that is done, for each i in 0..n-1, compute a_i = bounds[i][0] - prefix_sums[i], and b_i = bounds[i][1] - prefix_sums[i].

Then find the maximum of all a_i, and the minimum of all b_i. If max_a > min_b, return 0. Else, return min_b - max_a + 1.

But wait, what if there are multiple constraints? For example, for some i, a_i is 1, and for another i, a_i is 2. The maximum of a_i is 2. So x must be at least 2. Similarly, the min of b_i is 3. So x can be 2,3. So 2 numbers.

So the code needs to compute the maximum of all a_i, and the minimum of all b_i.

So, the steps in code:

First, check if original is empty? Well, according to constraints, original has length >=2.

So, code steps:

Compute diffs:

original is a list. For example, original = [1,2,3,4], then diffs is [1,1,1].

Compute prefix_sums:

```
prefix_sums = [0] * len(original)

for i in range(1, len(original)):

    prefix_sums[i] = prefix_sums[i-1] + diffs[i-1]
```

Once that is done, for each i in 0..n-1:

```
a_i = bounds[i][0] - prefix_sums[i]
```

```
b_i = bounds[i][1] - prefix_sums[i]
```

Compute max_a = max of all a_i.

min_b = min of all b_i.

if max_a > min_b: return 0.

else: return min_b - max_a +1.

But wait, what about the case where the max_a is not <= min_b? Then, no solution.

So the code can be written as follows.

Now, let's think about the constraints. original can be up to 1e5 elements. So O(n) time is acceptable.

Now, code in Python:

Implementing that.

But first, what about the case when n is 1? Well, according to the constraints, n is at least 2. Because the original's length is n >=2.

So, the code can proceed.

Now, let's think about the example 3:

original = [1,2,1,2], bounds = [[1,1],[2,3],[3,3],[2,3]]

diffs is [1, -1, 1].

prefix_sums is [0, 1, 0, 1].

Then for each i:

i=0: a_i = 1 - 0 =1, b_i=1-0=1.

i=1: a_i = 2 - 1 =1, b_i=3-1=2.

i=2: a_i =3 - 0 =3, b_i=3-0=3.

i=3: a_i =2 -1 =1, b_i=3-1=2.

max_a is 3, min_b is 1. So 3>1, return 0.

Which is correct.

Now, code.

Now, the code:

```
Implement:

def countArrays(self, original: List[int], bounds: List[List[int]]) -> int:

Compute diffs.

n = len(original)

diffs = [original[i] - original[i-1] for i in range(1, n)]

Then compute prefix_sums.

prefix_sums = [0] * n

for i in range(1, n):

    prefix_sums[i] = prefix_sums[i-1] + diffs[i-1]

Then, compute a_i and b_i for each i.

max_a = -infinity

min_b = +infinity

for i in range(n):

    a = bounds[i][0] - prefix_sums[i]

    b = bounds[i][1] - prefix_sums[i]

    if a > max_a:

        max_a = a

    if b < min_b:

        min_b = b

if max_a > min_b:

    return 0

else:

    return min_b - max_a + 1
```

But wait, what if the max_a is not possible? For example, when the a_i is 2, and the b_i is 1. So the code returns 0.

But what about the case when the a_i is 1, and the b_i is 3. So the x can be 1, 2, 3. So 3 numbers. So min_b - max_a +1 = 3-1 +1 =3. Correct.

But how about if the max_a is 3 and min_b is 3? Then 3-3+1=1, which is correct.

So the code is correct.

But what about when the a_i is not integer? No, because original and bounds are integers. So all a_i and b_i are integers. So the code is correct.

But wait, the problem says that the copy array must have elements within the bounds. So for each i, copy[i] must be between u_i and v_i, inclusive. So

the x must be such that for all i, u_i <= x + prefix_sums[i] <= v_i. So x must be >= u_i - prefix_sums[i], and x <= v_i - prefix_sums[i]. So the a_i is u_i - prefix_sums[i], and b_i is v_i - prefix_sums[i]. So the code is correct.

Yes.

Now, let's test the code with the examples.

Now, for example 1:

original = [1,2,3,4], bounds = [[1,2],[2,3],[3,4],[4,5]]

diffs = [1,1,1]

prefix_sums is [0,1,2,3]

for each i:

i=0: a=1-0=1, b=2-0=2

i=1: a=2-1=1, b=3-1=2

i=2: a=3-2=1, b=4-2=2

i=3: a=4-3=1, b=5-3=2

max_a is 1, min_b is 2. 2-1+1=2. Correct.

Another example: example 2, original is same, bounds are [[1,10],[2,9],[3,8],[4,7]]

prefix_sums same as before.

for i=0: a=1-0=1, b=10-0=10

i=1: a=2-1=1, b=9-1=8

i=2: a=3-2=1, b=8-2=6

i=3: a=4-3=1, b=7-3=4.

max_a is 1, min_b is 4. 4-1+1=4. Correct.

So the code works.

Now, what about when the original array has different differences?

Another test case: original = [5, 7], bounds = [[1, 10], [3, 15]]

diffs is [2]. prefix_sums is [0, 2].

For i=0: a is 1 - 0 =1, b is 10-0=10.

For i=1: a is 3 - 2 =1, b is 15-2=13.

max_a is 1, min_b is 10. So 10-1+1=10. So the possible x's are from 1 to 10. For each x, the array is [x, x+2]. For example, x=1: [1,3], which is allowed. x=10: [10, 12], which is allowed. So answer is 10.

Yes.

Another case: original = [5, 7], bounds = [[1, 1], [3, 3]]. So for i=0, a is 1-0=1, b is 1-0=1. So x must be 1. Then for i=1, x must be 3 - 2 =1. So x is 1. So the array is [1,3]. Which is allowed. So answer is 1.

Yes.

Now, code.

But in Python, the code needs to handle large n, but it's O(n) time, which is acceptable.

Now, code:

Implementing the steps.

But wait, the code for diffs:

original is a list. For i in range(1, n), original[i] - original[i-1].

Yes.

Now, code:

```
class Solution:
    def countArrays(self, original: List[int], bounds: List[List[int]]) -> int:
        n = len(original)
        if n == 0:
            return 0
        # compute diffs
        diffs = []
        for i in range(1, n):
            diffs.append(original[i] - original[i-1])
        # compute prefix_sums
        prefix_sums = [0] * n
        for i in range(1, n):
            prefix_sums[i] = prefix_sums[i-1] + diffs[i-1]
        # compute max_a and min_b
        max_a = -float('inf')
        min_b = float('inf')
        for i in range(n):
            u, v = bounds[i]
            a = u - prefix_sums[i]
            b = v - prefix_sums[i]
            if a > max_a:
                max_a = a
            if b < min_b:
                min_b = b
        if max_a > min_b:
            return 0
        else:
            return min_b - max_a + 1
```

Wait, but what if the max_a is not in the possible range? For example, if the a_i is 3 and the b_i is 2. Then, the code returns 0.

Yes.

But let's test the case where the original array is [1,2,3], and bounds are [[0, 1], [0, 1], [0, 1]]. Then, the diffs are [1, 1]. prefix_sums is [0, 1, 2].

For each i:

i=0: a = 0 - 0 = 0, b = 1 - 0 =1.

i=1: a = 0 -1 = -1, b = 1 -1 = 0.

i=2: a =0 -2 = -2, b=1-2 = -1.

max_a is max(0, -1, -2) = 0.

min_b is min(1, 0, -1) = -1.

So 0 > -1, so return 0. Which is correct, because the copy array must be [x, x+1, x+2]. For x to be between 0 and 1 (from first bound), x must be 0 or 1. But for x=0, the third element is 0+2=2, which is outside the third bound [0,1]. Similarly x=1: third element is 3, which is outside. So no solution. So code returns 0.

Yes.

Another test case: original = [1,3], bounds = [[1, 5], [3, 7]]. Then diffs is [2]. prefix_sums is [0, 2].

For i=0: a=1-0=1, b=5-0=5.

For i=1: a=3-2=1, b=7-2=5.

max_a is 1, min_b is 5. So 5-1+1=5. So possible x's are 1,2,3,4,5. For each x, the array is [x, x+2]. For example, x=1: [1,3], which is allowed. x=5: [5,7], allowed. So answer is 5. Correct.

So code works.

Now, what about the case where the first element's bounds are [1, 1], and the rest are [...]?

Yes, the code handles that.

So the code should be correct.

Now, check the sample input 3:

original = [1,2,1,2], bounds = [[1,1],[2,3],[3,3],[2,3]]

diffs is [1, -1, 1]

prefix_sums is [0, 1, 0, 1]

for each i:

i=0: a = 1 - 0 =1, b=1-0=1.

i=1: a=2 -1=1, b=3-1=2.

i=2: a=3 -0=3, b=3-0=3.

i=3: a=2-1=1, b=3-1=2.

max_a is 3, min_b is 1. So 3>1, return 0. Correct.

Yes.

So the code should be correct.

Now, what about when there are multiple possible ranges?

Another example: original = [1,3,5], bounds = [[1, 3], [2,4], [3,5]]

diffs are [2, 2]. prefix_sums is [0, 2, 4].

For each i:

i=0: a=1-0=1, b=3-0=3.

i=1: a=2-2=0, b=4-2=2.

i=2: a=3-4= -1, b=5-4=1.

max_a is max(1, 0, -1) =1.

min_b is min(3, 2, 1) =1.

so 1-1+1=1. So the x must be 1.

Check: copy array is [1, 3, 5]. Which satisfies all bounds. So answer is 1.
Correct.

Yes.

Another case: original = [1,3,5], bounds = [[0,5], [0,5], [0,5]]

prefix_sums [0, 2, 4].

for each i:

a_i is 0 - 0 =0, 0-2=-2, 0-4=-4.

b_i is 5-0=5, 5-2=3, 5-4=1.

max_a is max(0, -2, -4) =0.

min_b is min(5, 3, 1) =1.

so 0 <=1. So 1-0 +1 =2. So possible x's are 0 and 1.

Check:

x=0: array is [0, 2, 4]. All within bounds.

x=1: [1,3,5]. All within bounds. So answer is 2. Correct.

Yes.

So the code works.

Thus, the code should be correct.

To solve this problem, we need to determine the number of possible arrays `copy` that satisfy two conditions: the differences between consecutive elements must match those of the given `original` array, and each element in `copy` must lie within the corresponding bounds specified in the `bounds` array.

Approach
1. **Understand the Differences**: The key observation is that the differences between consecutive elements in the `original` array determine

the structure of any valid 'copy' array. This means once the first element of 'copy' is chosen, all subsequent elements are determined by these differences.
2. **Prefix Sums Calculation**: Calculate the prefix sums of the differences from the 'original' array. These prefix sums help determine the value of each element in 'copy' based on the starting value.
3. **Determine Valid Ranges**: For each position in the array, compute the valid range of starting values that ensure all elements of 'copy' stay within their respective bounds. This involves adjusting the bounds based on the prefix sums calculated earlier.
4. **Find Overlapping Range**: The valid starting values for the 'copy' array must lie within the intersection of all ranges determined for each position. The number of valid starting values is the length of this intersection.

Solution Code
```python
from typing import List

class Solution:
    def countArrays(self, original: List[int], bounds: List[List[int]]) -> int:
        n = len(original)
        if n == 0:
            return 0

        # Compute the differences between consecutive elements in original
        diffs = []
        for i in range(1, n):
            diffs.append(original[i] - original[i-1])

        # Compute prefix sums of the differences
        prefix_sums = [0] * n
        for i in range(1, n):
            prefix_sums[i] = prefix_sums[i-1] + diffs[i-1]

        # Determine the maximum lower bound (max_a) and minimum upper bound (min_b)
        max_a = -float('inf')
        min_b = float('inf')
        for i in range(n):
            u, v = bounds[i]
            a = u - prefix_sums[i]
            b = v - prefix_sums[i]
            if a > max_a:
                max_a = a
            if b < min_b:
                min_b = b

        # Calculate the number of valid starting values
        if max_a > min_b:
            return 0
        else:
            return min_b - max_a + 1
```

Explanation
1. **Differences Calculation**: The differences between consecutive elements in the 'original' array are stored in the 'diffs' list. These differences define the increments needed to form any valid 'copy' array.
2. **Prefix Sums**: The 'prefix_sums' array is constructed to help determine the value of each element in 'copy' based on the starting value. For

```
example, the value at position `i` in `copy` can be expressed as the
starting value plus the sum of the first `i` differences.
3. **Bounds Adjustment**: For each position, adjust the bounds to determine
the valid range of starting values. This involves subtracting the prefix sum
 at each position from the bounds.
4. **Intersection of Ranges**: The valid starting values must lie within the
 intersection of all adjusted ranges. The length of this intersection gives
the number of valid `copy` arrays.

This approach efficiently computes the valid starting values using prefix
sums and range intersections, ensuring the solution is both optimal and easy
 to understand.
```

References

[1] Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, et al. Phi-4-reasoning technical report. *arXiv preprint arXiv:2504.21318*, 2025.

[2] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[3] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[4] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

[5] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[6] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

[7] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[8] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[9] Piotr Nawrot, Robert Li, Renjie Huang, Sebastian Ruder, Kelly Marchisio, and Edoardo M Ponti. The sparse frontier: Sparse attention trade-offs in transformer llms. *arXiv preprint arXiv:2504.17768*, 2025.

[10] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

[11] Ranajoy Sadhukhan, Zhuoming Chen, Haizhong Zheng, Yang Zhou, Emma Strubell, and Beidi Chen. Kinetics: Rethinking test-time scaling laws. *arXiv preprint arXiv:2506.05333*, 2025.

[12] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[14] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

[15] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[16] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.