Pruning the Unsurprising: Efficient LLM Reasoning via First-Token Surprisal

Anonymous ACL submission

Abstract

Large Reasoning Models (LRMs) have demonstrated remarkable capabilities by scaling up the length of Chain-of-Thought (CoT). However, excessively long reasoning traces pose substantial challenges for training cost and inference latency. While various CoT compression approaches have emerged to address this challenge, they face inherent trade-offs: token-level methods often disrupt syntactic and logical coherence, while step-level methods based on perplexity fail to reliably capture the logically critical reasoning steps because of the dilution of logical information. In this paper, we propose **ASAP** (**A**nchor-guided, **S**urpris**A**l-based **P**runing), a novel coarse-to-fine framework for CoT compression. ASAP first performs anchor-guided pruning to preserve the core reasoning structure, which efficiently reduces the search space for subsequent processing. Leveraging the insight that logical branching choices are concentrated at the onset of reasoning steps, it then enables logic-aware pruning by selecting logically essential reasoning steps based on a novel first-token surprisal metric. Finally, ASAP distills the models to autonomously generate and leverage these concise CoTs at inference time, enabling efficient reasoning. Experiments show that ASAP achieves state-of-the-art accuracy across multiple benchmarks while substantially reducing training and inference costs.

1 Introduction

The emergence of Large Reasoning Models, including OpenAI's o1 (Jaech et al., 2024) and DeepSeek-R1 (Guo et al., 2025), marks a paradigm shift in artificial intelligence. By scaling up Chain-of-Thought (CoT) reasoning (Wei et al., 2022), these models demonstrate emergent capabilities in complex domains such as mathematics (Sun et al., 2025), programming (Shi et al., 2024; Yang et al., 2025b; Hu et al., 2025), and logical reasoning (Liu et al., 2025). However, this performance comes at a prohibitive cost: reasoning traces often span thousands of tokens, introducing substantial latency and memory overhead. Crucially, these lengthy traces often contain substantial redundancy, such as over-explaining simple problems or superficially exploring multiple paths for complex ones (Bi et al., 2024; Xie et al., 2025; Wu et al., 2025; Qu et al., 2025). For instance, the *Original CoT* in Figure 1 contains tangential branches (highlighted in red dashed boxes), such as exploring an alternative manual implementation that is subsequently rejected ("*But that's more code...*"). Furthermore, the reasoning is punctuated by syntactic fillers that contribute little

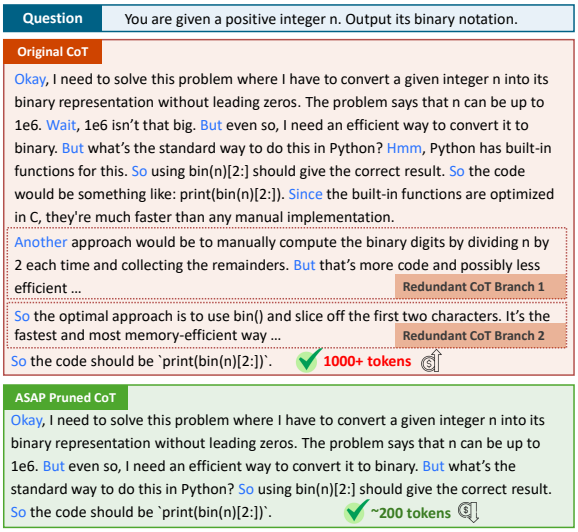


Figure 1: Illustration of CoT pruning by *ASAP*. The *Original CoT* generated by LRMs exhibits two types of redundancy: (1) **Structural Redundancy**, such as digressive branches (highlighted in red dashed boxes), which are removed by our Stage 1 Anchor-guided pruning; and (2) **Logical Redundancy** within valid paths. *ASAP* addresses the latter in Stage 2 by computing the surprisal of the first tokens of reasoning steps (marked in blue) to identify and retain only the critical cognitive pivots.

to the core logic. This observation raises a fundamental question: *Can we identify and retain only the "cognitive pivots" of reasoning while discarding the redundancy?*

A growing body of research has emerged on CoT compression for efficient reasoning (Qu et al., 2025). Token-level methods like TokenSkip (Xia et al., 2025a) adapt general-purpose context compressors such as LLMLingua-2 (Pan et al., 2024) to prune non-informative tokens. However, indiscriminate token removal risks disrupting the syntactic integrity of the reasoning chain. To address this, step-level pruning methods like SPIRIT (Cui et al., 2025) trim entire reasoning steps, thereby preserving structural coherence. Nevertheless, these approaches face a fundamental challenge: accurately estimating the logical importance of each step. They typically rely on fixed metrics like perplexity (PPL), which measures the overall predictability of a sentence. This holistic measure often dilutes the signal of critical logical leaps with the noise of syntactically predictable but logically trivial content.

In this work, we ground CoT compression from an information-theoretic perspective. Through an empirical analysis of 10 million reasoning tokens (detailed in Section 2), we find that the logical progression within a CoT sequence is not uniformly distributed; instead, its information density is highly concentrated at the beginning of each reasoning step—specifically within the first few tokens (blue-highlighted in Figure 1). These tokens, such as "*But*" (self-correction) or "*So*" (deduction, not continuation), serve as high-entropy **cognitive pivots**. By leveraging the surprisal of these initial tokens, we can distinguish between critical logical transitions and predictable elaborations.

Guided by this insight, we propose **ASAP** (**A**nchor-guided, **S**urpris**A**l-based **P**runing), a coarse-to-fine framework designed to preserve these high-information steps. ASAP in a two-stage cascade that directly addresses the two types of redundancies identified in our study (illustrated in Figure 1): First, it employs **Anchor-guided Pruning** to remove structural redundancies. By generating a concise step-by-step reasoning trace as a logical backbone, it identifies and prunes the irrelevant branches (e.g., the red boxes in Figure 1). Second, it performs **Surprisal-based Refining** to eliminate logic-sparse steps. Leveraging our First-Token Surprisal metric, this stage iteratively filters out steps acting as mere fillers while retaining the high-surprisal cognitive pivots. Finally, we distill these compact, logic-dense CoTs into a target model, enabling it to generate efficient reasoning chains.

We validate our approach through extensive experiments on the DeepSeek-R1-Distill-Qwen-7B and DeepSeek-R1-Distill-Llama-8B (Guo et al., 2025) across diverse domains. The results demonstrate that ASAP establishes a superior Pareto frontier between performance and efficiency. Notably, on the challenging LiveCodeBench v4_v5 benchmark, ASAP achieves **36.19%** Pass@1 while reducing token generation by **23.5%** and inference latency by **43.5%** compared to the strongest baseline.

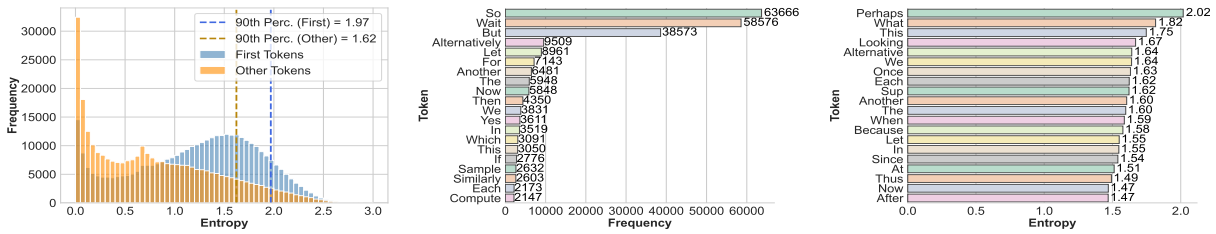
Our main contributions are summarized as follows:

- We present an empirical analysis of the information concentration of CoTs, uncovering that the surprisal of the starting token for each CoT step is a more robust indicator of logical importance than perplexity.
- We propose **ASAP**, a novel CoT compression framework that combines structural alignment with information-guided refinement.
- Extensive experiments on multiple benchmarks demonstrate that models fine-tuned on CoTs pruned by ASAP achieve state-of-the-art accuracy while substantially reducing computational costs.

2 Empirical Analysis

To investigate the intrinsic distribution of logical information in CoTs, we conducted a large-scale analysis on 10 million tokens generated by DeepSeek-R1-Distill-Qwen-32B (Guo et al., 2025) across diverse reasoning benchmarks (AIME and LiveCodeBench (Jain et al., 2024)).

Information Concentration in CoTs. We analyze the entropy distribution of the *first token* of each reasoning step compared to all subsequent tokens. Entropy, in this context, quantifies the model's uncertainty regarding the next state transition (Shannon, 1948; Malinin and Gales, 2020; Kuhn et al., 2023; Wang et al., 2025; Cheng et al., 2025). As illustrated in Figure 2(a), a distinct concentration is observed. Starting tokens (blue) exhibit a dispersed distribution with a significantly



(a) Entropy Distribution (b) High-Frequency First Tokens (c) High-Entropy Tokens

Figure 2: Empirical analysis of 10M tokens from DeepSeek-R1-Distill-Qwen-32B. (a) The entropy distribution reveals a clear information concentration: first tokens (blue) exhibit significantly higher uncertainty (entropy) compared to body tokens (orange), which are highly deterministic. (b) The most frequent first tokens are a mixture of logical operators (e.g., *Wait*) and ubiquitous syntactic connectors (e.g., *So*). (c) High-entropy states filter out predictable fillers like *So* or *Then*, while exclusively highlighting cognitive pivots such as *Perhaps*, *What*, and *Alternative*.

higher 90th percentile. In contrast, other tokens (orange) are heavily concentrated near zero entropy, indicating that once a reasoning step is initiated, its subsequent elaboration is largely deterministic and syntactically driven. This empirical evidence confirms that the logical branching points, where the model actively deliberates on the reasoning path, are structurally concentrated at the beginning of each step.

Identifying Cognitive Pivots with Entropy. Having identified the informative start tokens, we perform a more in-depth analysis of real logical pivots among the start tokens. We aim to distinguish between superficial syntactic connectors and real logical pivots. Figure 2(b) presents high-frequency start tokens that appear in CoTs, which mix connectors ("*So*", "*Let*") with reasoning markers ("*Wait*", "*But*"). However, when focusing on tokens generated in high-entropy states (Figure 2(c)), a qualitative shift emerges. Predictable connectors like "*So*" and "*Then*" are effectively suppressed due to their low uncertainty. Instead, the distribution is dominated by terms representing cognitive pivots and state transitions, such as: 1) Exploration: "*Alternative*", "*Another*" (proposing hypotheses or new paths). 2) Causality: "*Because*", "*Since*" (providing formal justification). 3) Self-Correction: "*Perhaps*", "*What*" (indicating error detection or logic reversal).

This analysis demonstrates that high entropy is a robust indicator of logical salience. Since the actual next token is known in the given training sequence, we operationalize this insight by using *First-Token Surprisal* as a proxy to identify and preserve these critical reasoning hops in our proposed framework (Fu et al., 2025).

3 Methodology

3.1 Overall Framework

Formally, we consider a supervised reasoning task defined by a dataset $\mathcal{D} = \{(Q_i, C_i, A_i)\}_{i=1}^{N}$, where Q_i is the query, A_i is the predicted answer, and C_i represents the original CoT generated by the LRMs. C_i is a sequence of reasoning steps $C_i = \{s_1, s_2, \ldots, s_L\}$. Our goal is to compress each C_i into a concise pruned one C_i' such that $|C_i'| \ll |C_i|$ while the LRMs maintains the quality of generated reasoning steps and answers when fine-tuned on the dataset $\mathcal{D}' = \{(Q_i, C_i', A_i)\}_{i=1}^{N}$.

We propose **ASAP**, a coarse-to-fine framework tailored to the redundancy of "Original CoT" (C), as illustrated in Figure 3. Stage 1 (Anchor-guided Pruning) reduces *structural redundancy* (e.g., dead ends) by aligning the CoT with a generated logical backbone. The LLM generates a "Direct Thought" ($\mathcal{P}$) from the (Q, A) pairs. $\mathcal{P}$ acts as an anchor to prune the C into a "Coarse-grained Pruned CoT" (C_{coarse}). Stage 2 (Surprisal-based Refining) reduces *logical redundancy* (e.g., syntactic fillers) by filtering non-informative steps. We approximate the information of each step in C_{coarse} with their surprisal of start tokens and prune low-surprisal steps, yielding the final "Fine-grained Pruned CoT" (C'). Finally, all $\{(Q, C', A)\}$ are utilized to fine-tune the target model.

3.2 Anchor-guided Pruning

Directly pruning raw CoTs is challenging due to the noise and unstructured digressions inherent in LLM reasoning (Zhou et al., 2024). To address this, we first construct a high-level logical skeleton to narrow the pruning space.

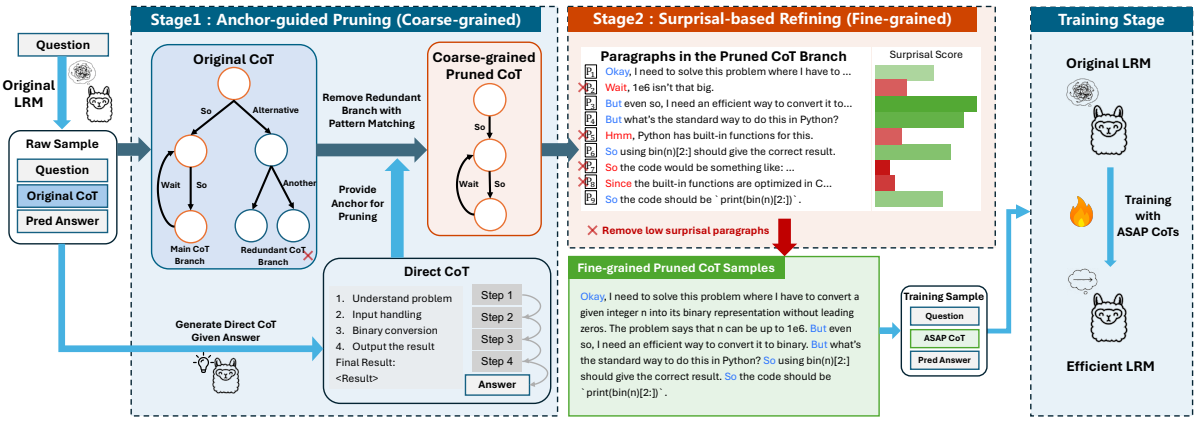


Figure 3: The overall framework of **ASAP**. The pipeline consists of three phases: (1) **In Stage 1**, the LLM generates a "Direct Thought" ($\mathcal{P}$) from the (Question, Answer) pair. $\mathcal{P}$ acts as an anchor to prune the "Original CoT" (C) into a "Coarse-grained Pruned CoT" (C_{coarse}). (2) **In Stage 2**, we compute the *First-Token Surprisal* for each step in C_{coarse}. High-surprisal steps are retained, while low-surprisal fillers are pruned, yielding the final "Fine-grained Pruned CoT" (C'). (3) **In Training Stage**, the data with ASAP pruned CoTs is used to fine-tune the LRM for efficient inference.

Generate Direct Thoughts. We prompt the LLM to infer a concise reasoning path called "Direct Thought" ($\mathcal{P}$) based on the (Q, A) pair (see Appendix B for prompts). Unlike exploratory CoTs, $\mathcal{P}$ is generated as a structured, step-by-step explanation that outlines how to derive the answer from the question, exemplified in Appendix B. This $\mathcal{P}$ acts as a reference anchor, outlining the least reasoning trajectory required to solve the problem.

Pruning with Pattern Matching. Guided by the anchor $\mathcal{P}$, we prompt the LLM to prune the original CoT C. Specifically, the LLM is instructed to: 1) remove unnecessary reasoning steps from C; 2) retain all key supporting content that aligns with the logic of $\mathcal{P}$; and 3) crucially, preserve the original wording without introducing new information. The prompt used for pruning is shown in Appendix B. The goal is to extract the subsequence of C that semantically aligns with $\mathcal{P}$ while discarding irrelevant branches (as shown in the "Original CoT" block of Figure 3).

Crucially, to mitigate LLM hallucination during compression, we enforce an extractive constraint, which validates structural and semantic alignment with C. Specifically, we design a pattern-matching algorithm that verifies whether each step in C_{coarse} corresponds to a matching step in C while preserving their original order. The matching is performed using Gestalt Pattern Matching (Black, 2004) as a text similarity metric. A pruning is considered valid only if all steps in C_{coarse} achieve a similarity score above a predefined threshold τ when matched against sequential steps in C. The full pattern-matching algorithm is detailed in Algorithm 1 (see Appendix A). We leverage high-temperature sampling, which provides the necessary diversity to efficiently re-prompt failed cases, ensuring that a valid C_{coarse} can be eventually generated.

3.3 Surprisal-based Refining

Following the coarse-grained pruning, the resulting C_{coarse} may still contain verbose steps that contribute little to the logic. Grounded in our empirical finding that logical information is concentrated (Section 2), we perform a meticulous, logic-aware refinement in C_{coarse} to identify more subtle redundancies within the core reasoning path.

First-Token Surprisal as Logical Importance. We introduce *First-Token Surprisal* as a novel metric to precisely quantify the logical importance of each step, enabling us to filter out the least informative ones and produce the final highly condensed CoT. Let a reasoning step s be a sequence of tokens $s = (x_1, x_2, \ldots, x_T)$. The informational value of s within the context of previous steps $\mathcal{C}_{pre}$ is typically estimated by its joint probability. However, our analysis reveals that the *first token* x_1 serves as the "cognitive pivot" carrying the majority of the uncertainty. Therefore, we define the *First-Token Surprisal* $\mathcal{S}(s)$ as:

$$\mathcal{S}(s \mid \mathcal{C}_{pre}) = -\log P_\theta(x_1 \mid \mathcal{C}_{pre}) \qquad (1)$$

where P_θ denotes the probability distribution of the LRM. A high $\mathcal{S}(s)$ indicates a high-information

transition (e.g., initiating a new deduction or self-correction), whereas a low score suggests a deterministic continuation or syntactic filler.

Pruning using First-Token Surprisal. We formulate the fine-grained pruning as a constrained maximization problem. Our goal is to select a subset of steps $S' \subset C_{coarse}$ that maximizes the total logical information subject to a length budget L_{max}:

$$\begin{aligned} S^* = &\arg\max_{S' \subseteq C_{coarse}} \sum_{s \in S'} \mathcal{S}(s) \\ \text{s.t.} \quad &\sum_{s \in S'} \text{len}(s) \leq L_{max} \end{aligned} \tag{2}$$

This formulation explicitly prioritizes steps with high information density. To solve this efficiently, we employ a greedy iterative strategy. We calculate the surprisal score for all steps in C_{coarse} and iteratively remove the step with the lowest $\mathcal{S}(s)$, while the relative order of steps in S' is preserved. The detailed procedure is provided in Algorithm 2 (see Appendix A). This process yields the final fine-grained CoT C', which retains the critical "aha moments" (Guo et al., 2025) while meeting efficiency constraints.

3.4 Supervised Fine-tuning

Following the pruning, we construct the final training dataset $\mathcal{D}' = \{(Q_i, \mathbf{C}'_i, A_i)\}_{i=1}^{N}$. For each instance, we concatenate the pruned CoT ($\mathbf{C}'_i$) and the final answer (A_i) to form the complete target response R_i. We then fine-tune the LRM to minimize the standard negative log-likelihood of the target response tokens, conditioned on the input question. Formally, the loss is defined as:

$$\mathcal{L} = -\sum_{i=1}^{N} \sum_{j=1}^{|R_i|} \log P_\theta(r_{i,j} | Q_i, r_{i,<j}) \tag{3}$$

where $r_{i,j}$ is the j-th token of the target response R_i, and θ represents the parameters of the model being fine-tuned. This supervised fine-tuning process effectively distills the knowledge from our pruning framework into the model. By training on these compact, logically salient examples, the model learns to internalize efficient reasoning patterns.

4 Experiments

4.1 Experimental Setup

Models and Datasets. All experiments are conducted on the DeepSeek-R1-Distill-Qwen-7B and DeepSeek-R1-Distill-Llama-8B (Guo et al., 2025), with DeepSeek-R1-Distill-Qwen-7B as the default backbone across all settings. For the code reasoning domains, we use the Python subset of the CodeForces-CoTs (Hugging Face, 2025) dataset. For the math reasoning domain, we adopt the OpenR1-Math (Hugging Face, 2025) dataset and randomly sample 10K instances to match the size of the code subset, ensuring a balanced comparison across domains. The datasets consist of high-quality Chain-of-Thought (CoT) samples generated by DeepSeek-R1, making it particularly suitable for training competitive reasoning tasks. Detailed implementation settings (hyperparameters, hardware, etc.) are provided in Appendix C.

Benchmarks. We evaluate our method on a suite of widely used benchmarks that cover both code generation and mathematical reasoning tasks. For code generation, we adopt HumanEval+ (Chen et al., 2021; Liu et al., 2023), LiveCodeBench `v1_v3`, LiveCodeBench `v4_v5` (Jain et al., 2024), and LeetCodeDataset (Xia et al., 2025b). For mathematical reasoning, we evaluate on GSM8K (Cobbe et al., 2021), MATH500 (Hendrycks et al., 2021), AIME24, and AIME25.

Baselines. We compare our method against a comprehensive set of baselines. **Zero-shot** refers to the original model without any task-specific fine-tuning. **Original** denotes the model fine-tuned on the uncompressed CoTs from the training data. Among compression approaches, Selective Context (Li et al., 2023) prunes redundant lexical units based on self-information; **LLMLingua-2** (Pan et al., 2024) distills GPT-4's token importance signals into a lightweight Transformer encoder trained as a token classifier; **TokenSkip** (Xia et al., 2025a) learns to skip less informative tokens to achieve controllable compression; and **SPIRIT** (Cui et al., 2025) identifies critical reasoning steps by measuring perplexity shifts. Except for the zero-shot setting, all methods involve fine-tuning on CoTs processed according to their respective compression strategies.

Metrics. We evaluate both accuracy and inference efficiency of each approach across three metrics: **Pass@1 (Acc)**, which measures the percentage of problems correctly solved on the first attempt; **Tokens (Tok)**, which denotes the average number of tokens generated by the LRMs; and **La-**

Methods	HE+			LCBv1_v3			LCBv4_v5			LCD		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
Zero-shot	68.29	3051	1.16	42.16	7088	3.59	25.37	8336	5.15	19.74	8680	4.95
Original	75.61	2973	1.12	52.12	6611	3.15	30.97	8289	4.83	25.00	8485	4.72
Selective Context	54.88	2979	1.13	30.23	7025	3.75	16.79	8558	5.35	15.79	8461	4.90
LLMLingua-2	68.29	3075	1.19	38.89	6953	3.60	22.76	8474	5.31	17.54	8513	4.81
TokenSkip	73.78	2823	1.07	32.35	7095	3.85	20.15	8400	5.37	18.42	8503	4.87
SPIRIT	75.61	2764	1.07	50.82	6524	3.09	33.58	7892	4.62	25.00	8186	4.45
ASAP	**78.66**	**2464**	**0.98**	**54.74**	**5177**	**2.09**	**36.19**	**6035**	**2.61**	**27.63**	**7541**	**3.48**

Table 1: Experimental results of different methods on code generation benchmarks with DeepSeek-R1-Distill-Qwen-7B. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds. The best results are highlighted in bold, and the second-best are underlined.

Methods	GSM8K			MATH500			AIME24			AIME25		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
Zero-shot	83.55	1301	0.27	60.40	2629	0.70	36.67	8352	6.76	40.00	8145	6.67
Original	86.35	1250	0.26	63.80	2511	0.66	**46.67**	8034	6.43	**43.33**	8026	6.75
Selective Context	75.44	1108	0.24	52.20	2507	0.66	16.67	9610	7.40	10.00	9329	7.40
LLMLingua-2	79.98	1128	0.24	54.60	2802	0.76	36.67	8369	6.60	23.33	8919	7.42
TokenSkip	85.37	1303	0.27	65.60	2483	0.65	36.67	8073	6.61	33.33	8465	7.44
SPIRIT	88.55	1118	0.23	64.20	2144	0.57	**46.67**	7198	5.78	**43.33**	7817	6.57
ASAP	**90.75**	**753**	**0.16**	**70.80**	**1649**	**0.43**	**46.67**	**5552**	**5.04**	36.67	**5434**	**5.10**

Table 2: Experimental results of different methods on mathematical reasoning benchmarks with DeepSeek-R1-Distill-Qwen-7B. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds. The best results are highlighted in bold, and the second-best are underlined.

tency (Lat), which measures the average time (in seconds) required for the model generation.

4.2 Main Results

Tables 1 and 2 present the results of various methods on all benchmarks. The results show that the model fine-tuned on CoTs pruned by ASAP consistently achieves the best trade-off between accuracy and efficiency. It achieves the best accuracy while generating the fewest tokens, leading to the lowest generation latency.

We notice a clear distinction between token-level and step-level pruning strategies. Token-level baselines such as Selective Context, LLMLingua-2, and TokenSkip exhibit a significant performance degradation compared to the original CoTs. This is because the token removal disrupts the syntactic structure and semantic coherence of the original reasoning steps. Consequently, the fine-tuning data becomes fragmented and grammatically unnatural, making it difficult for the model to learn the intended logical flow of the CoT. Step-level methods, such as SPIRIT, perform significantly better than token-level pruning methods, due to the preservation of sentence-level integrity. While SPIRIT improves efficiency over the Original with comparable accuracy, our method achieves higher efficiency and accuracy at the same time. This improvement is particularly pronounced on the challenging LiveCodeBench v4_v5 benchmark: ASAP reduces the average number of generated tokens by **23.5%** (from 7892 to 6035) and lowers generation latency by **43.5%** (from 4.62s to 2.61s), while also achieving a **7.8%** improvement in accuracy (Pass@1 increases from 33.58% to 36.19%).

4.3 Ablation and Analysis

Effect of Different Components. To validate the contribution of each component, we conduct an ablation study on three model variants. 1) *w/o Anchor-guided Pruning*: which skips Stage 1 and applies only surprisal-based pruning to the origi-

Variants	Acc ↑	Tok ↓	Lat ↓
ASAP	**36.19**	**6035**	**2.61**
w/o Anchor-guided Pruning	35.07	7735	4.60
w/o Surprisal-based Refining	31.72	8061	4.83
w/o Both Pruning	30.97	8289	4.83

Table 3: Ablation study of different pruning strategies on LiveCodeBench v4_v5. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Methods	LCB			AIME		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
Zero-shot	25.00	8508	8.90	33.33	8445	10.42
Original	31.34	8202	8.60	**36.67**	8550	10.04
SPIRIT	30.22	7913	8.45	**36.67**	8788	10.04
ASAP	**32.84**	**4175**	**2.69**	36.67	**5314**	**6.97**

Table 4: Experimental results of different methods with DeepSeek-R1-Distill-Llama-8B. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds. The best results are highlighted in bold.

nal CoT. 2) *w/o Surprisal-based Refining*: which omits the surprisal-based refinement stage; and 3) *w/o Both Pruning*: equivalent to the original baseline, where the model is fine-tuned on the full, uncompressed CoT. Table 3 presents the results on LiveCodeBench v4_v5, which is representative of the consistent trends observed across benchmarks. Additional results are included in Appendix D. The results show that both pruning stages are essential and mutually complementary for optimal accuracy and efficiency. First, removing the anchor-guided pruning leads to a drop in both accuracy and efficiency. While the accuracy decrease is modest, the generation latency increases by a substantial **76.2%** (from 2.61s to 4.60s), underscoring the importance of stage 1. Second, removing the surprisal-based refining results in a significant degradation across all metrics. The accuracy drops by **12.4%** (Pass@1 decreases from 36.19% to 31.72%) relative to the ASAP, and efficiency improvements are largely lost. This highlights that our surprisal-based pruning mechanism is essential to select the most critical steps.

Generalization to Different Architectures. To validate the generalizability of ASAP, we replicate our main experiments on the DeepSeek-R1-Distill-Llama-8B. Following the same experimental protocol, we compare ASAP against three strong baselines: Zero-shot, Original, and SPIRIT. We observe consistent trends across all benchmarks. For brevity, we present representative results on two key benchmarks: LiveCodeBench v4_v5 and AIME24 in Table 4, while reporting the full results in the Appendix E. The results in the Llama3.1 series are highly consistent with our findings in the Qwen2.5 series, confirming the generalization of the ASAP. As shown in Table 4, ASAP achieves the highest accuracy on both benchmarks, and the efficiency improvements are even more pronounced. On LiveCodeBench, for instance, ASAP not only

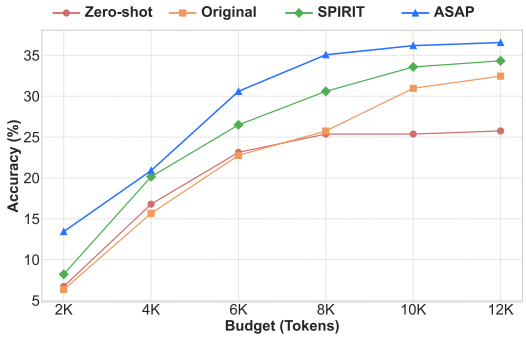


Figure 4: Performance of ASAP on LiveCodeBench v4_v5 under different token budgets.

surpasses the accuracy of the Original baseline (32.84% vs. 31.34%) but also generates **49.1%** fewer tokens and reduces latency by over **3x** (from 8.60s to 2.69s). This suggests that the ASAP is particularly effective in identifying and distilling the core reasoning patterns, validating its robustness and broad applicability for improving reasoning efficiency across different model families.

Impact of Token Budget. To evaluate the scalability and resource sensitivity of our method, we analyze its behavior under varying inference-time token budgets (i.e., the maximum number of tokens to the model). We compare ASAP against the three strong baselines-SPIRIT, Original, and Zero-shot—across all benchmarks, and observe consistent trends. For clarity, we present results on LiveCodeBench v4_v5 under six budget settings ranging from 2K to 12K tokens. Results for other benchmarks and additional statistics are provided in Appendix F. As shown in Figure 4, ASAP consistently outperforms all baselines across all budget settings. In particular, ASAP exhibits smooth performance scaling with respect to the token budget. We note that ASAP achieves superior performance-efficiency trade-offs. For example, ASAP with just

Methods	Tokens	Time
Original	13023	80.11
Selective Context	6722 (-48.4%)	63.41 (-20.9%)
LLMLingua-2	6919 (-46.9%)	65.25 (-18.6%)
TokenSkip	9813 (-24.6%)	77.27 (-3.6%)
SPIRIT	6082 (-53.3%)	57.45 (-28.3%)
ASAP	**3178 (-75.6%)**	**31.48 (-60.7%)**

Table 5: Training efficiency comparison on CodeForces-CoTs dataset. We report the average number of tokens per sample and training time measured in seconds per step. Percentages indicate the reduction relative to the Original baseline.

an 8K token budget achieves higher accuracy than SPIRIT and Original at a much larger 12K budget. These results further validate the practical utility of ASAP in real-world scenarios.

Training Efficiency. To quantify the training efficiency gains, we present results of the CodeForces-CoTs dataset in Table 5 and results on other datasets are provided in Appendix G. The results highlight the training efficiency advantage of the ASAP. By generating the most compact yet logically rich CoTs, our approach significantly reduces training overhead. Compared to the uncompressed baseline (Original), our method reduces the number of training tokens by **75.6%** and shortens training time by **60.7%**. These savings substantially exceed those achieved by all other baselines. ASAP enables a more resource-efficient training process, making it a practical and cost-effective solution for real-world deployment.

5 Related Work

Chain-of-Thought and Advanced Reasoning. Chain-of-Thought (CoT) prompting (Wei et al., 2022) has evolved from heuristic prompting strategies (Yao et al., 2023; Lei et al., 2023; Ling et al., 2023) to the training of specialized Large Reasoning Models (LRMs) like OpenAI's o1 (Jaech et al., 2024) and DeepSeek-R1 (Guo et al., 2025). These models leverage reinforcement learning to scale test-time compute, generating lengthy reasoning traces to solve complex tasks (Kimi et al., 2025; Yang et al., 2025a; Yu et al., 2025; Wang et al., 2025). Unlike prior works that enhance performance by scaling up CoT length, we focus on pruning redundancy to improve efficiency without compromising reasoning performance.

Context Compression for LLMs. To mitigate the computational cost of long contexts, various compression techniques have been proposed. Approaches like Selective Context (Li et al., 2023), LLMLingua series (Jiang et al., 2023; Pan et al., 2024), and LongCodeZip (Shi et al., 2025) employ information-theoretic metrics or small external models to filter redundant tokens. However, these methods typically treat input as unstructured text. Applying them directly to CoT often disrupts the syntactic and logical coherence required for valid reasoning, a limitation that our pruning aims to overcome.

Efficient Reasoning via Fine-Tuning. Recent research has explored various efficiency mechanisms (Qu et al., 2025), ranging from compressing thoughts into continuous latent representations (Hao et al., 2024; Cheng and Van Durme, 2024; Shen et al., 2025) to compressing CoTs (Kang et al., 2025; Xia et al., 2025a; Cui et al., 2025). Approaches like TokenSkip (Xia et al., 2025a) and SPIRIT (Cui et al., 2025) reduce length by filtering tokens or steps based on heuristics or perplexity shifts. However, these metrics often struggle to differentiate between syntactic fluency and logical necessity. ASAP differs by combining anchor-guided structural pruning with first-token surprisal, offering a more robust proxy for cognitive pivots.

6 Conclusion

In this paper, we address the inefficiency of Large Reasoning Models stemming from the structural and logical redundancies in Chain-of-Thought reasoning. Grounded in an information-theoretic perspective, our large-scale empirical analysis reveals a fundamental property of reasoning traces: Information Concentration, where the logical uncertainty is highly concentrated at the onset of reasoning steps. Guided by this insight, we propose ASAP. This coarse-to-fine framework first aligns the reasoning structure with a logical anchor and then refines it using a novel First-Token Surprisal metric. Extensive experiments across multiple benchmarks demonstrate that ASAP outperforms existing baselines, establishing a new state-of-the-art Pareto frontier between accuracy and efficiency. Our work highlights the potential of using information-theoretic signals for efficient reasoning. Future work will explore applying ASAP to online inference acceleration.

Limitations

While ASAP demonstrates significant improvements in reasoning efficiency, we acknowledge several limitations. First, our method relies on the availability of a capable LLM to generate high-quality "Direct Thoughts" in Stage 1. If the anchor contains logical errors or hallucinations, it may misguide the subsequent pruning, although our pattern-matching constraint mitigates this risk. Second, our experiments primarily focus on code generation and mathematical reasoning. While we believe the principle of information concentration applies broadly, the effectiveness of ASAP on creative writing or commonsense reasoning tasks remains to be verified.

References

Zhen Bi, Ningyu Zhang, Yinuo Jiang, Shumin Deng, Guozhou Zheng, and Huajun Chen. 2024. When do program-of-thought works for reasoning? In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 17691–17699.

Paul E Black. 2004. Ratcliff/obershelp pattern recognition. *Dictionary of algorithms and data structures*, 17.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Daixuan Cheng, Shaohan Huang, Xuekai Zhu, Bo Dai, Wayne Xin Zhao, Zhenliang Zhang, and Furu Wei. 2025. Reasoning with exploration: An entropy perspective. *arXiv preprint arXiv:2506.14758*.

Jeffrey Cheng and Benjamin Van Durme. 2024. Compressed chain of thought: Efficient reasoning through dense representations. *arXiv preprint arXiv:2412.13171*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Yingqian Cui, Pengfei He, Jingying Zeng, Hui Liu, Xianfeng Tang, Zhenwei Dai, Yan Han, Chen Luo, Jing Huang, Zhen Li, and 1 others. 2025. Stepwise perplexity-guided refinement for efficient chain-of-thought reasoning in large language models. *arXiv preprint arXiv:2502.13260*.

Yichao Fu, Xuewei Wang, Yuandong Tian, and Jiawei Zhao. 2025. Deep think with confidence. *arXiv preprint arXiv:2508.15260*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2024. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Minghao Hu, Junzhe Wang, Weisen Zhao, Qiang Zeng, and Lannan Luo. 2025. Flowmaltrans: Unsupervised binary code translation for malware detection using flow-adapter architecture. *arXiv preprint arXiv:2508.20212*.

Hugging Face. 2025. Open r1: A fully open reproduction of deepseek-r1.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

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

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376.

Yu Kang, Xianghui Sun, Liangyu Chen, and Wei Zou. 2025. C3ot: Generating shorter chain-of-thought without compromising effectiveness. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24312–24320.

Kimi, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, and 1 others. 2025. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. *arXiv preprint arXiv:2302.09664*.

Bin Lei, Chunhua Liao, Caiwen Ding, and 1 others. 2023. Boosting logical reasoning in large language models through a new framework: The graph of thought. *arXiv preprint arXiv:2308.08614*.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. 2023. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6342–6353.

Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. 2023. Deductive verification of chain-of-thought reasoning. *Advances in Neural Information Processing Systems*, 36:36407–36433.

Hanmeng Liu, Zhizhang Fu, Mengru Ding, Ruoxi Ning, Chaoli Zhang, Xiaozhang Liu, and Yue Zhang. 2025. Logical reasoning in large language models: A survey. *arXiv preprint arXiv:2502.09100*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Andrey Malinin and Mark Gales. 2020. Uncertainty estimation in autoregressive structured prediction. *arXiv preprint arXiv:2002.07650*.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, and 1 others. 2024. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 963–981.

Xiaoye Qu, Yafu Li, Zhaochen Su, Weigao Sun, Jianhao Yan, Dongrui Liu, Ganqu Cui, Daizong Liu, Shuxian Liang, Junxian He, and 1 others. 2025. A survey of efficient reasoning for large reasoning models: Language, multimodality, and beyond. *arXiv preprint arXiv:2503.21614*.

Claude E Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423.

Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. 2025. Codi: Compressing chain-of-thought into continuous space via self-distillation. *arXiv preprint arXiv:2502.21074*.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2025. Longcodezip: Compress long context for code language models. *arXiv preprint arXiv:2510.00446*.

Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.

Jiankai Sun, Chuanyang Zheng, Enze Xie, Zhengying Liu, Ruihang Chu, Jianing Qiu, Jiaqi Xu, Mingyu Ding, Hongyang Li, Mengzhe Geng, and 1 others. 2025. A survey of reasoning with foundation models: Concepts, methodologies, and outlook. *ACM Computing Surveys*, 57(11):1–43.

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xionghui Chen, Jianxin Yang, Zhenru Zhang, and 1 others. 2025. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for llm reasoning. *arXiv preprint arXiv:2506.01939*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Yuyang Wu, Yifei Wang, Ziyu Ye, Tianqi Du, Stefanie Jegelka, and Yisen Wang. 2025. When more is less: Understanding chain-of-thought length in llms. *arXiv preprint arXiv:2502.07266*.

Heming Xia, Chak Tou Leong, Wenjie Wang, Yongqi Li, and Wenjie Li. 2025a. Tokenskip: Controllable chain-of-thought compression in llms. *arXiv preprint arXiv:2502.12067*.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025b. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *arXiv preprint arXiv:2504.14655*.

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. 2025. Logic-rl: Unleashing llm reasoning with rule-based reinforcement learning. *arXiv preprint arXiv:2502.14768*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Xinwei Yang, Zhaofeng Liu, Chen Huang, Jiashuai Zhang, Tong Zhang, Yifan Zhang, and Wenqiang Lei. 2025b. Elaboration: A comprehensive benchmark on human-llm competitive programming. *arXiv preprint arXiv:2505.16667*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Zhanke Zhou, Rong Tao, Jianing Zhu, Yiwen Luo, Zengmao Wang, and Bo Han. 2024. Can language models perform robust reasoning in chain-of-thought prompting with noisy rationales? *Advances in Neural Information Processing Systems*, 37:123846–123910.

A Algorithms

Algorithm 1 Pattern Matching

Require: Original CoT C, Coarse-grained Pruned CoT C_{coarse}, Threshold τ
Ensure: *True* if C_{coarse} is valid, *False* otherwise.

```
function PATTERNMATCH(C, C_coarse, τ)
    S_origin ← SplitStepsByBlankLine(C)
    S_coarse ← SplitStepsByBlankLine(C_coarse)
    origin_idx ← 0
    for each step s_coarse in S_coarse do
        found_match ← False
        while origin_idx < Length(S_origin) do
            s_origin ← S_origin[origin_idx]
            score ← GestaltSimilarity(s_origin, s_coarse)
            if score ≥ τ then
                found_match ← True
                origin_idx ← origin_idx + 1
                break
            end if
            origin_idx ← origin_idx + 1
        end while
        if not found_match then
            return False
        end if
    end for
    return True
end function
```

Algorithm 2 Iterative Pruning via First-Token Surprisal

Require: Coarse-grained Pruned CoT C_{coarse}, Max Tokens L_{max}, Model M, Tokenizer T
Ensure: Fine-grained Pruned CoT C'

```
function FINEGRAINEDPRUNE(C_coarse, L_max, M, T)
    if Length(T(C_coarse)) ≤ L_max then
        return C_coarse
    end if
    S ← SplitStepsByBlankLine(C_coarse)
    SurprisalScores ← CalculateAll(S, M, T)
    StepsToPrune ← SortByScore(S, SurprisalScores)
    S_current ← S
    for each step s_prune in StepsToPrune do
        S_temp ← S_current \ {s_prune}
        C_temp ← Join(S_temp)
        if Length(T(C_temp)) ≤ L_max then
            S_current ← S_temp
            break
        end if
        S_current ← S_temp
    end for
    C' ← Join(S_current)
    return C'
end function
```

B Prompt Templates and Generated Examples

We prompt the LLM to produce a direct thought: a concise, structured, step-by-step explanation that outlines how to derive the answer from the given question. A direct thought always consists of a small number of clear logical steps that directly contribute to the solution, and terminates with a single, explicit final answer. We use the prompt below to elicit such direct thoughts (the model is instructed to return only a detailed step-by-step solution containing only “Step-by-Step Solution" and “Final Answer").

Prompt for Generating the Direct CoT $\mathcal{P}$

Given a question, please tell me how to get this answer step by step.
Question:
```
{question}
```
Answer:
```python
{answer}
```
Only return a detailed step-by-step solution (containing only “Step-by-Step Solution" and “Final Answer"). The detailed step-by-step solution is:

The following example illustrates a direct thought for a simple programming task: converting an integer into its binary representation. This example demonstrates the defining characteristics of direct CoTs, namely, concise step-by-step reasoning that directly derives the solution, without redundant exploration or verbose commentary.

Example of a Direct CoT

Step-by-Step Solution
1. Understand the Problem: The task is to convert a given positive integer `n` into its binary representation without leading zeros. The constraints are lenient ($n \leq 10^6$), so even a straightforward approach is efficient.
2. Input Handling: Read the integer `n` from standard input. Since the input consists of a single integer, we can directly use `input()` and convert it to an integer.
3. Binary Conversion: Python's built-in `bin()` function converts an integer to its binary string representation (prefixed with `0b`). To remove the prefix, slice the string from the third character onward (`[2:]`).
4. Output the Result: Print the resulting binary string. This ensures no leading zeros are present.
5. Efficiency: The approach is optimal because conversion and slicing both run in $O(\log n)$ time, and memory usage is minimal.
Final Code:
```python
n = int(input())
print(bin(n)[2:])
```

Given the direct thought as a reference, this stage instructs the model to prune redundant reasoning from the original chain of thought while preserving the essential logical structure. Specifically, the model is required to 1) remove unnecessary reasoning steps from the original CoT, 2) retain all key supporting content that aligns with the logic of direct CoT, and 3) strictly preserve the original wording and sentence order without introducing new information. This ensures that the compressed reasoning remains faithful to the original thought process while aligning with the concise, goal-oriented structure of the direct CoT. The following detailed prompt is used to elicit such coarse-grained pruning behavior.

Prompt for Coarse-grained Pruning

Compress the given thinking by referring to the provided solution. The goal is to remove irrelevant reasoning paths while retaining all content along the core reasoning path. Compression must be based on thinking, ensuring that the original wording and structure are preserved as much as possible. Follow these strict rules:
1. Use thinking as the foundation: Do not rewrite or replace its content with solution——only use solution to determine which parts are relevant.
2. Remove unnecessary reasoning: Aggressively remove alternative paths that are not part of the core reasoning path.
3. Retain key supporting content: Keep examples, reflections, and tests that help illustrate, verify, or analyze the core reasoning path.
4. Preserve original words: Do not paraphrase, reorder, or change any words.
5. Do not add new words: Do not introduce new concepts, symbols, or abbreviations.
If you understand, compress the following thinking based on the given solution.
Solution:
```
*{solution}*
```
Thinking:
```
*{think}*
```
The compressed thinking is:

C Implementation Details

Software and Hardware. For fine-tuning, we utilized the unsloth library[1] for its memory-efficient optimizations. For inference, we employed the vLLM engine[2] to maximize throughput and efficiency. All experiments were conducted on NVIDIA H20 GPUs and Intel Xeon Platinum 8480+ CPUs.

Fine-tuning Configuration. We performed full-parameter fine-tuning for all models in our experiments. Key hyperparameters included precision set to bf16, `num_train_epochs` set to 10, and a `max_seq_length` of 16384. We used a `per_device_train_batch_size` of 1 with `gradient_accumulation_steps` set to 16, resulting in an effective batch size of 16. For the optimizer, we used AdamW with a `cosine_with_min_lr` learning rate scheduler. The `warmup_ratio` was set to 0.03, and the scheduler's `min_lr_rate` was 0.1 of the peak learning rate. To stabilize training, we applied gradient clipping with a `max_grad_norm` of 0.2. Based on pre-

[1] https://pypi.org/project/unsloth/2025.5.6/
[2] https://pypi.org/project/vllm/0.8.4/

liminary experiments, we set the peak learning rate to 4×10^{-5} for the DeepSeek-R1-Distill-Qwen-7B and 2×10^{-5} for the DeepSeek-R1-Distill-Llama-8B. Due to the high computational cost of full-parameter fine-tuning, the model is fine-tuned by a single run with a fixed random seed 42.

Inference and Evaluation Protocol. All inference benchmarks were run using the vLLM engine with `dtype` set to bfloat16 and `gpu_memory_utilization` set to 0.9. To ensure deterministic and reproducible outputs, we set the sampling `temperature` to 0.0 and set `enable_prefix_caching` to False. The default token budget for generation is adjusted based on the task difficulty. Specifically, it is 2K for GSM8K, 4K for MATH500, 6K for HumanEval+, and 10K for AIME24, AIME25, LiveCodeBench, and LeetCodeDataset. Results with other token budget settings are shown in Appendix F.

Baseline Details. Following established practices, we used a consistent scoring model; as our primary model is DeepSeek-R1-Distill checkpoints, we employed DeepSeek-R1-Distill-Qwen-7B for all model-scoring tasks. To ensure a fair comparison, we standardize the input format across all methods by preserving the original question and final answer, and applying compression only to the CoT reasoning steps. To balance compression ratio and content retention, we set the target compression ratio to 0.5 for all baseline methods, except for TokenSkip, where we follow its original design that allows a controllable compression ratio between 0.5 and 1.0. Additionally, since the original SPIRIT method is computationally expensive when applied to extremely long CoTs, we adopt a modified version to ensure fair comparison: specifically, we compute perplexity once per reasoning step and iteratively remove steps until the target ratio is met. This variant retains the core idea of SPIRIT while improving scalability in our evaluation setting.

Hyperparameters for Our Method. Our method involves several stages. For the LLM-guided Coarse-grained Pruning stage, we employed DeepSeek-V3 for economic reasons. When generating the direct thought $\mathcal{P}$, we used a deterministic setting (`temperature=0.0`, `top_p=1.0`), while for making the final pruning result, we increased exploration (`temperature=1.0`, `top_p=1.0`). For Pattern Matching, the similarity threshold τ was set to 0.6. Finally, during Surprisal-based Fine-grained Pruning, the maximum token budget was set to 4096 to ensure a deep level of compression.

D Effect of Different Components.

To validate the contribution and necessity of each component in our two-stage pruning framework, we conduct a detailed ablation study. Specifically, we evaluate the following three variants: *ASAP w/o Coarse-grained Pruning*, *ASAP w/o Fine-grained Pruning*, and *ASAP w/o Any Pruning*. We present results on the HumanEval+, LiveCodeBench v1_v3, and LeetCodeDatsets benchmarks in Table 6, Table 7, and Table 8.

Variants	Acc ↑	Tok ↓	Lat ↓
ASAP	**78.66**	**2464**	**0.98**
w/o Coarse-grained Pruning	78.05	2839	1.10
w/o Fine-grained Pruning	67.07	2897	1.10
w/o Any Pruning	75.61	2973	1.12

Table 6: Ablation study of different pruning strategies for ASAP on HumanEval+. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Variants	Acc ↑	Tok ↓	Lat ↓
ASAP	**54.74**	**5177**	**2.09**
w/o Coarse-grained Pruning	53.92	6107	2.77
w/o Fine-grained Pruning	51.14	6599	3.20
w/o Any Pruning	52.12	6611	3.15

Table 7: Ablation study of different pruning strategies for ASAP on LiveCodeBench v1_v3. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Variants	Acc ↑	Tok ↓	Lat ↓
ASAP	**27.63**	**7541**	**3.48**
w/o Coarse-grained Pruning	24.12	7954	3.75
w/o Fine-grained Pruning	25.44	8326	4.77
w/o Any Pruning	25.00	8485	4.72

Table 8: Ablation study of different pruning strategies for ASAP on LeetCodeDataset. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

E Generalization to Different Architectures

To evaluate the generalizability of ASAP, we replicate our main experiments on the DeepSeek-R1-Distill-Llama-8B. Following the same experimental protocol, we compare ASAP against three baselines: Zero-shot, Original, and SPIRIT. The results of the code generation task on the HumanEval+, LiveCodeBench v1_v3, LiveCodeBench v4_v5, and LeetCodeDataset benchmarks are shown in Table 9. The results of the mathematical reasoning task on the GSM8K, MATH500, AIME24, and AIME25 benchmarks are shown in Table 10.

F Performance under Different Token Budgets

To evaluate the performance scalability and resource sensitivity of our method, we analyze its behavior under varying inference-time token budgets (i.e., the maximum number of tokens the model is allowed to generate). We compare ASAP with three strong baselines—SPIRIT, Original, and Zero-shot—on HumanEval+, LiveCodeBench v1_v3, LiveCodeBench v4_v5, LeetcodeDataset, GSM8K, MATH500, AIME24, and AIME25. For simpler benchmarks (including HumanEval+, GSM8K, and MATH500), we evaluate the performance under four budget settings, ranging from 1K to 6K tokens. For more complex benchmarks (including LiveCodeBench v1_v3, LiveCodeBench v4_v5, LeetcodeDataset, AIME24, and AIME25), we evaluate the performance under six budget settings, ranging from 2K to 12K tokens. Results are shown in Table 11, Table 12, Table 13, Table 14, Table 15, Table 16, Table 17, and Table 18.

G Training Efficiency

To quantify the training efficiency gains, we present results of the CodeForces-CoTs dataset in Table 5 and results of the OpenR1-Math dataset in Table 19. We report two key metrics: the *average number of tokens* per sample and the *average training time* measured in seconds per step.

Methods	Tokens	Time
Original	5807	47.82
Selective Context	3149 (-45.8%)	25.85 (-45.9%)
LLMLingua-2	3478 (-40.1%)	28.75 (-39.9%)
TokenSkip	4728 (-18.6%)	39.20 (-18.0%)
SPIRIT	2858 (-50.8%)	23.67 (-50.5%)
ASAP	**1834 (-68.4%)**	**15.36 (-67.9%)**

Table 19: Training efficiency comparison on OpenR1-Math dataset. We report the average number of tokens per sample and training time measured in seconds per step. Percentages indicate the reduction relative to the Original baseline.

Methods	HE+			LCBv1_v3			LCBv4_v5			LCD		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
Zero-shot	64.02	3334	1.86	44.12	7162	6.92	25.00	8508	8.90	27.19	8358	8.65
Original	76.22	2978	1.63	52.61	6614	6.16	31.34	8202	8.60	26.32	8413	8.85
SPIRIT	72.56	3159	1.74	52.61	6280	5.84	30.22	7913	8.45	26.75	8449	8.73
ASAP	76.83	2494	1.30	48.86	3605	2.18	32.84	4175	2.69	27.63	3792	2.42

Table 9: Experimental results of different methods on code generation benchmarks with DeepSeek-R1-Distill-Llama-8B. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Methods	GSM8K			MATH500			AIME24			AIME25		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
Zero-shot	79.15	1262	0.36	57.20	2612	1.08	33.33	8445	10.42	26.67	8597	10.54
Original	84.91	1310	0.37	63.00	2534	1.01	36.67	8550	10.04	30.00	8268	10.05
SPIRIT	85.67	1256	0.35	62.60	2533	1.01	36.67	8788	10.04	36.67	8094	9.57
ASAP	87.34	768	0.20	66.00	1734	0.65	36.67	5314	6.97	33.33	5348	7.05

Table 10: Experimental results of different methods on mathematical reasoning benchmarks with DeepSeek-R1-Distill-Llama-8B. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
1K	9.76	1007	0.28	14.63	983	0.28	10.98	995	0.28	23.78	946	0.27
2K	42.68	1813	0.53	43.29	1702	0.49	47.56	1690	0.49	54.88	1502	0.44
4K	66.46	2561	0.85	65.85	2511	0.82	69.51	2401	0.80	71.34	2116	0.72
6K	68.29	3051	1.16	75.61	2973	1.12	75.61	2764	1.07	78.66	2464	0.98

Table 11: Results of different methods under different budgets on HumanEval+. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
2K	16.50	1966	0.52	17.16	1920	0.51	18.95	1908	0.51	21.57	1833	0.49
4K	32.68	3499	1.06	30.72	3432	1.05	34.80	3370	1.03	34.97	3244	1.00
6K	39.05	4806	1.70	42.65	4673	1.67	43.14	4605	1.64	46.24	4358	1.54
8K	44.28	5903	2.46	47.71	5723	2.43	51.80	5515	2.27	52.61	4919	1.90
10K	42.16	7088	3.59	52.12	6611	3.15	50.82	6524	3.09	54.74	5177	2.09
12K	43.95	7988	5.10	54.41	7473	4.22	51.63	7362	4.09	55.56	5322	2.27

Table 12: Results of different methods under different budgets on LiveCodeBench v1_v3. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
2K	6.72	2021	0.59	6.34	1999	0.57	8.21	1993	0.56	13.43	1930	0.54
4K	16.79	3820	1.22	15.67	3799	1.20	20.15	3712	1.18	20.90	3594	1.15
6K	23.13	5444	2.07	22.76	5397	2.00	26.49	5237	1.93	30.60	4988	1.85
8K	25.37	6927	3.27	25.74	6882	3.24	30.60	6634	3.09	35.07	5793	2.38
10K	25.37	8336	5.15	30.97	8289	4.83	33.58	7892	4.62	36.19	6035	2.61
12K	25.75	9706	7.44	32.46	9567	7.10	34.33	8987	6.73	36.57	6128	2.76

Table 13: Results of different methods under different budgets on LiveCodeBench v4_v5. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
2K	7.02	2028	0.53	6.14	2020	0.53	7.02	2001	0.53	10.09	1965	0.53
4K	13.16	3848	1.21	13.16	3854	1.21	16.23	3789	1.19	15.79	3758	1.19
6K	16.23	5553	2.04	16.67	5548	2.04	18.86	5407	2.00	19.30	5387	2.00
8K	19.30	7165	3.27	22.37	7104	3.18	22.37	6882	3.04	23.25	6722	2.88
10K	19.74	8680	4.95	25.00	8485	4.72	25.00	8186	4.45	27.63	7541	3.48
12K	21.49	10142	7.58	28.07	9717	7.09	26.32	9354	6.86	27.63	7902	3.83

Table 14: Results of different methods under different budgets on LeetCodeDataset. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
1K	48.75	963	0.19	59.29	942	0.19	54.66	925	0.18	83.93	693	0.14
2K	83.55	1301	0.27	86.35	1250	0.26	88.55	1118	0.23	90.75	753	0.16
4K	88.65	1553	0.37	90.37	1432	0.34	90.52	1227	0.28	91.28	778	0.18
6K	89.23	1714	0.46	91.05	1513	0.39	91.28	1297	0.33	91.81	790	0.20

Table 15: Results of different methods under different budgets on GSM8K. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
1K	19.00	1020	0.19	28.00	1017	0.19	18.40	1012	0.19	36.40	935	0.19
2K	42.20	1804	0.39	52.00	1767	0.39	54.40	1592	0.36	59.80	1347	0.31
4K	60.40	2629	0.70	63.80	2511	0.66	64.20	2144	0.57	70.80	1649	0.43
6K	66.60	3100	0.94	70.60	2843	0.84	69.60	2460	0.74	71.00	1758	0.52

Table 16: Results of different methods under different budgets on MATH500. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
2K	0.00	2048	0.93	16.67	2048	0.93	6.67	2048	0.93	10.00	1978	0.93
4K	20.00	4003	2.12	20.00	3984	2.11	23.33	3877	2.10	36.67	3415	1.99
6K	33.33	5682	3.58	36.67	5695	3.55	40.00	5216	3.32	36.67	4410	3.03
8K	30.00	7073	5.30	40.00	7093	5.14	46.67	6243	4.47	40.00	5159	4.10
10K	36.67	8352	6.76	46.67	8034	6.43	46.67	7198	5.78	46.67	5552	5.04
12K	40.00	9318	7.84	46.67	8990	7.75	46.67	8363	7.21	46.67	5767	5.88

Table 17: Results of different methods under different budgets on AIME24. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.

Budget	Zero-shot			Original			SPIRIT			ASAP		
	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓	Acc ↑	Tok ↓	Lat ↓
2K	6.67	2046	0.95	13.33	2048	0.94	3.33	2044	0.94	10.00	2020	0.95
4K	20.00	3851	2.17	26.67	3834	2.16	16.67	3792	2.15	20.00	3511	2.15
6K	30.00	5369	3.62	33.33	5452	3.63	36.67	5360	3.63	30.00	4484	3.14
8K	36.67	6848	5.30	36.67	6798	5.32	36.67	6611	5.16	33.33	5002	4.10
10K	40.00	8145	6.67	43.33	8026	6.75	43.33	7817	6.57	36.67	5434	5.10
12K	36.67	9442	8.25	40.00	9461	8.32	46.67	8598	7.78	36.67	5720	6.06

Table 18: Results of different methods under different budgets on AIME25. We report accuracy (Acc), average number of generated tokens (Tok), and average generation latency (Lat) measured in seconds.