Scaling LLM Inference Efficiently with Optimized Sample Compute Allocation

Kexun Zhang[1]*, **Shang Zhou**[2]*, **Danqing Wang**[1], **William Yang Wang**[3], **Lei Li**[1]

[1]Carnegie Mellon University, [2]UC San Diego, [3]UC Santa Barbara

Abstract

Sampling is a basic operation for large language models (LLMs). In reinforcement learning rollouts and meta generation algorithms such as Best-of-N, it is essential to sample high-quality trajectories efficiently within a given compute budget. To find an optimal allocation for sample compute budgets, several choices need to be made: Which sampling configurations (model, temperature, language, etc.) to use? How many samples to generate in each configuration? We formulate these choices as a learning problem and propose OSCA, an algorithm that Optimizes Sample Compute Allocation by finding an optimal mix of different inference configurations. Our experiments show that with our learned mixed allocation, we can achieve accuracy better than the best single configuration with 128x less compute on code generation and 25x less compute on 4 reasoning tasks. OSCA is also shown to be effective in agentic workflows beyond single-turn tasks, achieving a better accuracy on SWE-Bench with 3x less compute than the default configuration. Our code and generations are released at https://github.com/LeiLiLab/OSCA.

1 Introduction

Large language models (LLMs) solve more problems with more inference compute. Different ways of scaling up LLM inference include sampling (Chen et al., 2021), self-consistency (Wang et al., 2023c), tree search (Yao et al., 2024), etc. Among these, sampling is the most basic and serves as an atomic operation needed in all other more complicated methods. Sampling is also essential for collecting high-quality rollouts in reinforcement learning. Therefore, it is crucial to do it well.

Previous studies (Wang et al., 2023b) have investigated how to find the optimal sampling configuration, such as the best temperature. While

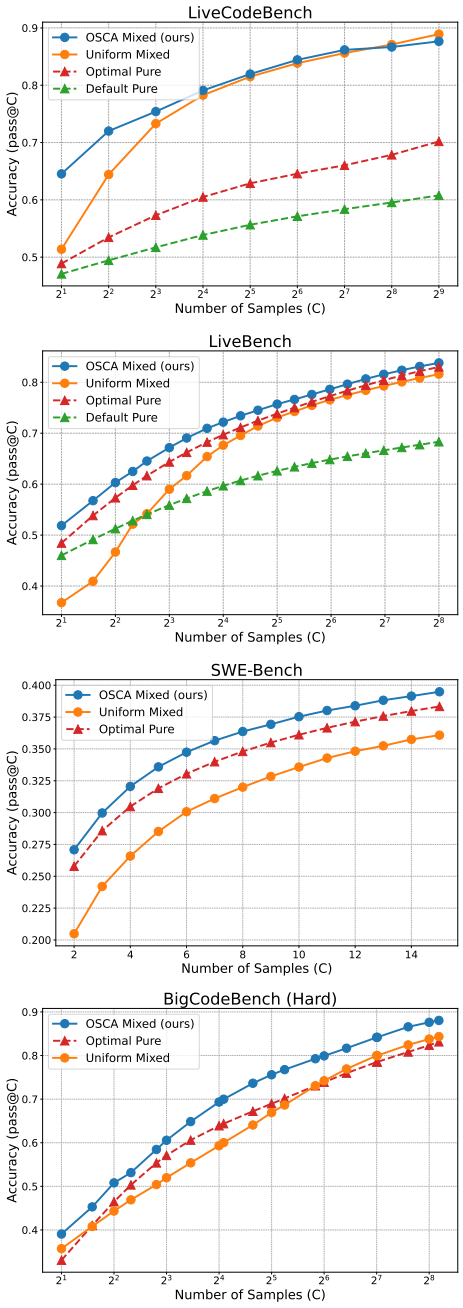

Figure 1: On 4 benchmarks with a total of 7 tasks, our optimized allocations are better than both optimal pure allocations and uniform allocations in most cases.

*Equal contribution. Correspondence to kexun@cmu.edu.

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 7959–7973

these methods are effective, they miss one key fact: ***Not all problems have the same optimal sampling configuration.*** Some problems are easier solved in one configuration while others in another (Li et al., 2022). This highlights the need for a *mixed* allocation of the sample budget instead of a *pure* allocation that uses a single configuration.

As shown in Figure 1, just by uniformly allocating the compute budget over all possible inference configurations (dubbed "uniform mixed"), LLMs' accuracy can already surpass the optimal pure allocation on LiveCodeBench. Uniform mixed allocation gets 64% accuracy with 4 samples, while optimal pure needs 16x more samples to get a similar accuracy. Since uniform allocation is only one of the exponentially many allocations in the search space, it is natural to ask: ***How do we find the optimal sample budget allocation for LLM inference?***

We formulate this as a learning problem: given a set of different inference configurations, a training problem set, and a compute budget, we need to distribute the budget to maximize the expected accuracy. To solve this problem, we use a hill-climbing algorithm and demonstrate its effectiveness with theoretical justification and experiments. As shown in Figure 1, the learned allocation from OSCA is significantly better than the optimal pure allocation, especially when the sample size is small.

To provide further insights for future adopters of OSCA, we conduct ablation studies on the effectiveness of different hyperparameters in the mix, the number of problems needed in the training set, and scenarios where mixed allocations do not offer significant improvements. Moreover, we show on SWE-Bench, a benchmark for LLM agents, that replacing pure sampling with OSCA's learned allocation in just one step boosts the entire workflow's performance. This demonstrates that a mixed sampling strategy not only improves single-turn tasks but also enhances agentic workflows, leading to better reasoning and decision-making.

Our contributions are the following:

- We highlight the need for mixed allocation of LLM sample compute and formulate it as a learning problem.
- We propose an effective algorithm OSCA for optimizing sample compute allocations and demonstrate its effectiveness on 4 benchmarks.
- We provide detailed analyses of when and why mixed allocations work, as well as their role in more complicated inference time algorithms.

2 Related Work

Inference Time Algorithms. Following the taxonomy of Welleck et al. (2024) on inference-time algorithms, *chained meta-generators* run multiple LLM calls sequentially and use the output sample from each call as the input to the next one (Dohan et al., 2022; Schlag et al., 2023). *Parallel meta-generators* samples multiple candidates for a problem and selects the best candidate (Wang et al., 2023c; Chen et al., 2022; Jiang et al., 2023; Zhang et al., 2023; Huang et al., 2023). *Step-level search methods* regards problem-solving as a multi-step process and sample candidate next steps at each intermediate state, using algorithms like tree search (Yao et al., 2024), graph search, and Monte-Carlo Tree Search (Lample et al., 2022; Tian et al., 2024; Chi et al., 2024). *Refinement-based* methods samples candidate solutions sequentially, relying on some feedback to revise the next candidate (Madaan et al., 2024; Shinn et al., 2024). Although these algorithms scale up inference differently, they all need LLM sampling as a basic operation.

Scaling Inference. Many studies investigate how scaling in inference affects LLM performance. AlphaCode (Li et al., 2022; Leblond et al., 2023) scales up the sample number and finds the solve rates scale log-linearly with more samples. Brown et al. (2024) improves LLMs' performance on math problems by repetitively sampling candidate solutions with a high temperature. Wu et al. (2024) and Snell et al. (2024) study the scaling behaviors of various inference time algorithms, reward models, and the model sizes. In this paper, we investigate the allocation algorithm between various inference configurations for scaling up inference.

Inference Compute Optimization. There are two typical ways to optimize inference compute. One is to search for a single optimal configuration. For example, Wang et al. (2023a) proposes EcoOptiGen to optimize the inference hyperparameter under a limited compute budget, and Wang et al. (2024) finetunes LLMs to self-regularize its generation with the best hyperparameter set. The other is to find the optimal allocation between various inference configurations (Graves, 2016; Dehghani et al., 2019). Damani et al. (2024) allocates the compute budget based on the estimation of problem difficulty. Here, we optimize the compute allocation to different inference configurations.

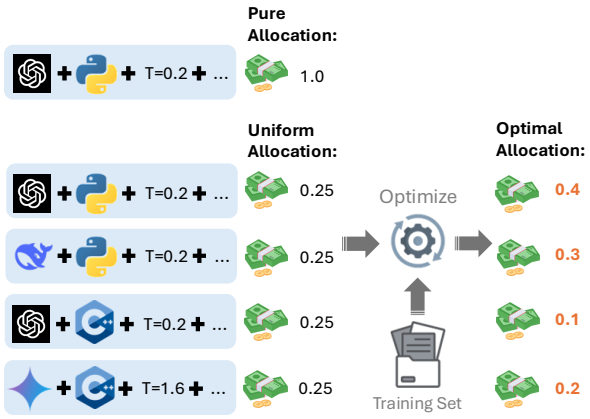

Figure 2: With 4 sampling configurations and a compute budget of 1, pure allocation spends all the budget on one configuration. Uniform allocation evenly distributes the budget across configurations. Optimized allocation learns where to spend more budget.

3 Method

3.1 Problem: Sample Compute Allocation

LLM Problem-Solving. We study how to improve LLM-based solvers for problems with binary correctness values. Formally, a ***problem*** is a pair (x, v), where $x \in \mathcal{X}$ is the problem specifications, $v : \mathcal{X} \times \mathcal{Y} \to \{\texttt{True}, \texttt{False}\}$ is the verifier that maps a solution of the problem to a truth value, and $\mathcal{Y}$ is the space of solutions. An ***LLM solver*** is given the problem specification and produces a distribution $\Pr_{\text{LM}}(y|x)$ over the solution space conditioned on the problem specification.

Compute Budget and pass@C. We evaluate LLM-based problem solvers with the average solve rate of test set problems given a fixed ***compute budget*** C, which can be defined as the maximum number of samples, the maximum number of tokens, the maximum FLOPs, etc. For any definition of compute budget C, we generate as many samples as possible from $\Pr_{\text{LM}}(y|x)$ within the compute limit. The solver can be evaluated using ***pass@C***, which is defined as the probability of at least one sample among the candidates being correct. In our experiments, we use the number of samples as a metric for C.

Sampling Configurations. Given a problem specification x, there are multiple inference hyperparameters to decide when solving it with an LLM – the model to use, inference temperature, language of the output, prompt, etc. For the i-th hyperparameter, we use H_i to denote the set of feasible values and h_i to denote an element in H_i that is actually chosen. Assuming the number of tunable hyperparameters is d, we call an d-tuple $\boldsymbol{h} = (h_1, h_2, \cdots, h_d) \in \mathcal{H} = H_1 \times H_2 \times \cdots \times H_d$ a ***sampling configuration***.

Sample Compute Allocation. Suppose there are $|\mathcal{H}| = m$ sampling configurations, and we want to allocate a compute budget C across them. We define an allocation as a mapping function $\pi : \mathcal{H} \to \mathbb{N}$ to represent the amount of compute assigned to each sampling configuration and it should satisfy $\sum_{\boldsymbol{h} \in \mathcal{H}} \pi(\boldsymbol{h}) = C$. For convenience we use π_i to denote $\pi(\boldsymbol{h}_i)$. We categorize sample budget allocations into two types: A ***pure allocation*** spends all the compute on one sampling configuration, i.e., there exists exactly one i for which $\pi_i > 0$; while a ***mixed allocation*** spends compute on more than one sampling configurations. Examples of different types of allocations can be found in Figure 2.

Learning Allocations. We want to find for a test set of problems $\mathcal{D}_{\text{test}}$ and a given per-problem compute budget C, a sample compute allocation π, such that the problem solve-rate pass@C can be maximized given the same amount of compute. We assume access to an i.i.d. training problem set $\mathcal{D}_{\text{train}}$ and a per-problem allocation-learning compute budget C_0 that can be used for trying out different sampling configurations.

3.2 Why Mixed Allocation?

We show an example that demonstrates why it is sometimes necessary to use a mixed allocation of sample compute. Consider two problems x_1 and x_2, and two configurations $\boldsymbol{h}_1$ and $\boldsymbol{h}_2$. We use $\Pr(\text{pass}|\boldsymbol{h}, x)$ to denote the probability of generating a correct solution to the problem x under configuration $\boldsymbol{h}$. Let's assume that $\Pr(\text{pass}|\boldsymbol{h_1}, x_1) = 10\%, \Pr(\text{pass}|\boldsymbol{h_2}, x_1) = 1\%, \Pr(\text{pass}|\boldsymbol{h_1}, x_2) = 1\%, \Pr(\text{pass}|\boldsymbol{h_2}, x_2) = 10\%$. In other words, $\boldsymbol{h}_1$ is better at solving x_1 and $\boldsymbol{h}_2$ is better at solving x_2.

Consider the allocation of 10 samples per problem to these two configurations. If we use a pure allocation, all 10 samples will be entirely given to either $\boldsymbol{h}_1$ or $\boldsymbol{h}_2$, the expected pass@10 would be

$$\frac{1 - (1 - 0.1)^{10} + 1 - (1 - 0.01)^{10}}{2} = 37.3\%.$$

On the other hand, if we use a mixed allocation and split 10 samples evenly between $\boldsymbol{h}_1$ and $\boldsymbol{h}_2$, the expected pass@10 would be

$$1 - (1 - 0.1)^5 \times (1 - 0.01)^5 = 43.8\%,$$

which is significantly higher than the pure allocation's result.

3.3 Our Algorithm

The proposed OSCA is described in Algorithm 1.

Estimating pass probability. For each sampling configuration $\boldsymbol{h}_i$ and each problem x_j in the training set, we estimate a probability matrix P with $|\mathcal{H}| \times |\mathcal{D}|$ elements. Each element p_{ij} indicates the probability of a sample from $\boldsymbol{h}_i$ solving x_j. We estimate p_{ij} by sampling C_0 times from $\boldsymbol{h}_i$ and computing the frequency of correct solutions,

$$p_{ij} \approx \frac{c_{ij}}{C_0}, \tag{1}$$

where c_{ij} is the number of correct samples generated for the problem x_j with the configuration $\boldsymbol{h}_i$. Note that to save compute, the estimation compute budget C_0 is much smaller than the actual compute budget C.

Maximizing expected pass@C on training set. Assuming that the test set is i.i.d. with the training set, we optimize π to maximize pass@C on training data. For a single problem x_j and a single configuration $\boldsymbol{h}_i \in \mathcal{H}$, its pass@$C_i$ is defined as it probability of being solved with compute C_i, which can be derived as

$$\begin{aligned}&\Pr(x_j \text{ solved with } C_i \text{ samples from } \boldsymbol{h}_i)\\ &\qquad = 1 - (1 - p_{ij})^{C_i}.\end{aligned} \tag{2}$$

By aggregating all problems and all configurations, we can obtain this optimization problem:

$$\begin{aligned}&\max_{\pi} \mathbb{E}[\text{pass@}C]\\ &\quad = \frac{1}{|\mathcal{D}|} \sum_{j=1}^{|\mathcal{D}|} \left(1 - \prod_{i=1}^{|\mathcal{H}|} (1 - p_{ij})^{\pi_i}\right),\\ &\text{s.t. } 0 \le \pi_i \le C,\\ &\quad \sum_{i=1}^{|\mathcal{H}|} \pi_i = C, \pi_i \in \mathbb{N}.\end{aligned} \tag{3}$$

Note that if we remove the integral constraint $\pi_i \in \mathbb{N}$ from Problem (3), the relaxed problem is convex (Proof in Appendix A.1) that can be solved optimally by hill climbing algorithm (Russell and Norvig, 2016). We start from a randomly picked distribution of compute $\pi^{(0)}$. At each iteration t, we examine all the neighbors of the current distribution that differ slightly from $\pi^{(t)}$ and "climb" to the neighbor if it's better than the current distribution to obtain $\pi^{(t+1)}$. The algorithm stops once there is no better neighbor. Though the algorithm is not guaranteed to produce global optima for integral solutions, it works well empirically.

This algorithm can be extended to problems with real scores with some slight modification, as discussed in Appendix A.4.

Algorithm 1 OSCA: Algorithm for Optimizing Sample Compute Allocation

Input: Inference budget C, estimation budget C_0, sampling configurations $\mathcal{H}$, training problem set $\mathcal{D}_{\text{train}}$, where $(x_i, v_i) \in D_{\text{train}}$ is a problem specification and its verifier.
Output: Inference strategy π.

function OPTIMIZEALLOCATION($C, C_0, \mathcal{H}, \mathcal{D}_{\text{train}}$)
  $P \leftarrow$ ESTIMATEPASSPROB($C, C_0, \mathcal{H}$)
  Randomly initialize allocation $\pi = \{\pi_1, \ldots, \pi_m\}$ such that $\sum_{i=1}^{m} \pi_i = C$.
  repeat
    improved $\leftarrow$ `False`
    // Enumerating all neighboring strategies
    for $i \leftarrow 1$ to m, $j \leftarrow 1$ to m and $i \neq j$ and $\pi_i > 0$ **do**
      $q \leftarrow \pi$
      $q_i, q_j \leftarrow \pi_i - 1, \pi_j + 1$
      if PREDACC(q, P)>PREDACC(π, P) **then**
        $\pi \leftarrow q$ *// Climb to a better strategy.*
        improved $\leftarrow$ `True`.
      end if
    end for
  until not improved
end function

function ESTIMATEPASSPROB($C, C_0, \mathcal{H}$)
  for $\boldsymbol{h}_i \in \mathcal{H}$ **do**
    for $(x_j, v_j) \in \mathcal{D}_{\text{train}}$ **do**
      Generate C_0 samples $y_1, y_2, \cdots, y_{C_0}$, $y_i \sim \Pr_{\text{LM}}(y|x, \boldsymbol{h})$.
      $c \leftarrow |\{y_i : v_j(y_i) = \texttt{True}\}|$.
      $p_{ij} \leftarrow c/C_0$.
    end for
  end for
end function

function PREDACC(π, P)
  return $\frac{1}{|\mathcal{D}|} \sum_{j=1}^{|\mathcal{D}|} \left(1 - \prod_{i=1}^{|\mathcal{H}|} (1 - p_{ij})^{\pi(\boldsymbol{h}_i)}\right)$
end function

Discussion on the definition of *compute budget*. In our experiments, we use the number of samples as a metric for compute, which is reasonable when the compute/price to generate each sample is similar for different configurations. However, there are other cases when the model sizes differ greatly or when the FLOPs budget for each sampling configuration varies significantly. For those cases, *sample budget* is not a good proxy for *compute budget*, and Problem (3) will need to be modified to accommodate different definitions of compute budget, possibly by adding a coefficient to weight the sample budget for each problem. Nonetheless, such modifi-

cation does not change the convexity of the relaxed problem. Thus our algorithm can still apply.

4 Evaluation on Single-Turn Tasks

We first evaluate OSCA on single-turn tasks, where a solution to a problem can be generated with a single LLM call.

4.1 Baselines

We consider three baselines in our experiments – default pure allocation, optimal pure allocation, and uniform mixed allocation:

Default pure allocation is the one used to produce the leaderboard results on the benchmarks. These benchmarks usually run LLM inference once for each problem instance, with a very basic prompt and a low temperature for reproducibility and fair comparison. Therefore, the default pure allocation is not optimized for scaling up.

Optimal pure allocation is the one that has the highest pass@C_0 on the training set given a compute budget of C_0. Since the actual C is larger than C_0 in most cases, we use pass@C_0 to select the optimal configuration. Comparison between the optimal pure allocation and the default pure allocation indicates whether it is necessary to search for a good set of inference hyperparameters, which is what existing hyperparameter optimization techniques do.

Uniform mixed allocation naively distributes the compute budget evenly to every sampling configuration $\boldsymbol{h}_i$ in $\boldsymbol{H}$. We compare the uniform mixed allocation with our learned mixed allocation to examine whether it is necessary to optimize sample compute allocations for them to perform well.

4.2 Tasks and Benchmarks

We evaluate OSCA on 6 tasks from 3 benchmarks – LiveCodeBench (Jain et al., 2024), LiveBench (White et al., 2024), and BigCodeBench (Zhuo et al., 2024). The first two benchmarks are built with periodically released examinations and competitions so that the possibility of contamination can be minimized. To further avoid contamination, we choose the earlier released problems as training set $\mathcal{D}_{\text{train}}$.

LiveCodeBench (Jain et al., 2024) collects LeetCode-style problems from weekly held online programming competitions.Each problem comes with a natural language specification including problem description, sample test cases, and input range. When given a problem, an LLM is supposed to create a program that can pass both the sample tests visible to the model and the hidden tests that are usually more comprehensive. Within each platform in LiveCodeBench, we sort the problems according to the time they were released and use the first 3/5 of the problems as training data. This split gives us 305 problems for training and 205 problems for evaluation. Most training problems are released before 2024, while the evaluation problems are after 2024.

LiveBench (White et al., 2024) contains 6 problem categories: *data analysis, language, reasoning, math, instruction following, and coding*. The responses to the instruction-following problems are not easy to verify, and their coding problems are mostly taken from LiveCodeBench. The remaining four categories are used for evaluation. The problems have two release dates (*2024-06-24 and 2024-07-26*), but most categories have only one release. Therefore, we mixed these two versions and randomly split each category by 3:7, resulting in 201 training and 471 testing problems. Compared to LiveCodeBench, optimizing sample compute allocation is harder on LiveBench, because there are problems from different domains.

BigCodeBench (Zhuo et al., 2024) contains 148 function completion problems that require multiple python libraries to implement. We split up the problems randomly into 70 training problems and 78 testing problems.

		Model	Temp.	Lang.
LCB	Default	GPT-4o	0.2	Python
	Optimal	GPT-4o	1	Python
	Mixed	GPT-4o Gemini DeepSeek	0-1.6	Python, C++
LB	Default	Llama3	0	N/A
	Optimal	Llama3	0.8	N/A
	Mixed	Qwen2 Llama3 DeepSeek	0-1.8	N/A
BCB	Default	Qwen2	0.2	N/A
	Optimal	Llama3	1.4	N/A
	Mixed	Qwen2 Llama3	0.2-1.4	N/A

Table 1: Implementation details for the three benchmarks, including the hyperparameters we consider, the default inference strategy on the leaderboard, and the budget for strategy learning and inference. Temp. and Lang. are the temperature and the programming language used when sampling.

We list the sampling configurations in Table 1. The temperatures we consider are multiples of 0.2. For all configurations, we set C_0 to 50. Note that there is one more hyperparameter to optimize for LiveCodeBench – the programming language. For LiveBench, the open-weight models we choose all have about 70B parameters. We also provide the values of hyperparameters for the pure allocation baselines and the budgets for learning and testing.

4.3 Key Observations

We present the pass rates for different sample compute allocations in Table 2. For allocations other than the default pure, we show the difference between their pass rate and that of default pure. Several key observations can be made from these results:

Pure allocation is not enough, mixed allocation is necessary. By finding the optimal pure allocation on the training set, we get much better accuracy than the default allocation, highlighting the importance of a suitable inference configuration. However, pure allocation is not enough on LiveCodeBench. We find that code problems require a more diverse solution set, making the pure configuration not enough. On LiveCodeBench, just by allocating sample compute evenly across inference settings, we achieve a pass@8 of 73.3%, which is better than the pass rate of optimal pure allocation with 512 samples.

Uniform mixed allocation is not enough, OSCA's optimized allocation is necessary. On LiveBench, uniform mixed allocation does not outperform optimal pure, suggesting that we can't always opt for uniform mixed. However, OSCA's optimized allocation does. With 8 samples, OSCA's pass rate comes to 67%, which is higher than the pass rate of the default pure allocation with 128 samples. In fact, OSCA outperforms all others in all but two cases.

OSCA scales well with larger compute budgets. Although the sample compute budget (C_0) used for estimating the solve rate matrix was merely 50, OSCA can still produce strong sample compute allocations even when the test time compute budget C is as large as 1024 per problem.

4.4 Ablation Studies

We conduct ablation experiments on LiveBench to answer the following questions.

How many problems do we need in training to learn a good mixed allocation? We run OSCA on different proportions of the original training set and plot the results in Figure 3. Since there are multiple ways of subsampling the training set, we run it multiple times and compute the average. For reference, we also plot the results when we train on the test set, which should be the upperbound of OSCA. We observe that with the full training data, the allocation learned is very close to the upperbound. At smaller sample compute C, the allocation trained from less data is not much worse than the allocation from full training data. However, as C gets to over 2^6, the allocation gets much worse with less training data. We hypothesize that this is because when C is large, only the hard problems contribute to the difference in accuracy, which are prone to overfitting when training data is small.

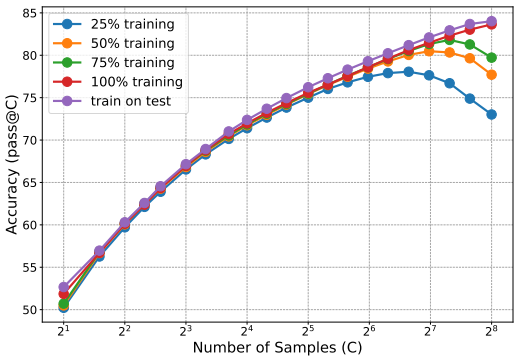

Figure 3: OSCA's pass rates on LiveBench when trained with different proportions of the training data.

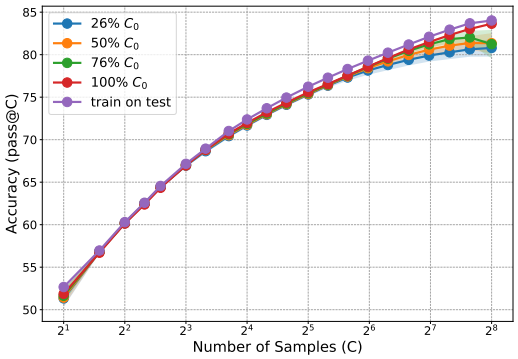

Figure 4: OSCA with different sample compute for estimating pass rates. $C_0 = 50$.

How large does C_0 need to be in order to estimate pass rates on training data? C_0, the sample compute budget for estimating pass rates on training data, can be much smaller than C, which might affect OSCA's performance. We study how estimating with different values of C_0 might affect the learning process. As Figure 4 shows, even with

		pass@C with varying $C \uparrow$								
		2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
LiveCodeBench	Default pure	47.1	49.4	51.7	53.9	55.7	57.1	58.4	59.5	60.8
	Optimal pure	+1.8	+4.1	+5.6	+6.6	+7.2	+7.5	+7.6	+8.4	+9.4
	Uniform mixed	+4.3	+15.0	+21.6	+24.4	+25.8	+26.7	+27.2	**+27.6**	**+28.1**
	Learned mixed (ours)	**+17.4**	**+22.6**	**+23.7**	**+25.2**	**+26.3**	**+27.3**	**+27.7**	+27.2	+26.9
LiveBench	Default pure	46.0	51.3	55.9	59.7	62.6	64.9	66.7	68.3	-
	Optimal pure	+2.4	+6.0	+9.3	+10.0	+11.2	+12.5	+13.8	+14.7	-
	Uniform mixed	-11.7	-4.4	+3.1	+8.0	+10.5	+11.6	+12.5	+12.8	-
	Learned mixed (ours)	**+5.9**	**+8.8**	**+11.1**	**+12.3**	**+13.0**	**+13.8**	**+14.8**	**+14.7**	-

Table 2: Accuracy at different compute budgets (C) using different sample compute allocations. We report the difference in accuracy for non-default allocations. LCB stands for code generation tasks in LiveCodeBench. LB stands for subtask categories in LiveBench. The best result for each setting is in bold.

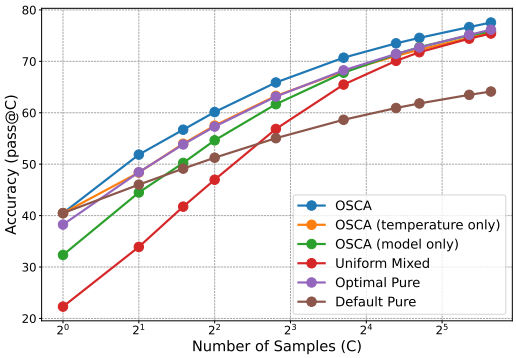

Figure 5: OSCA's pass rates on LiveBench when it is banned from allocating compute to multiple temperatures or multiple models.

26% of the original C_0, OSCA can still get an accuracy very close to its upperbound, indicating that it is not really sensitive to C_0.

Which hyperparameter in sampling configurations needs mixed allocation? To study whether mixed allocation is needed for the two hyperparameters – temperature or model, we consider two limited versions of OSCA by banning it from using more than one temperature or more than one model. As shown in Figure 5, when OSCA is not allowed to use multiple models, its accuracy degrades to that of optimal pure. When OSCA is not allowed to use multiple temperatures, its accuracy degrades to be worse than optimal pure. These results suggest that in order for OSCA to perform well, we need more diverse sampling configurations.

How general is OSCA's learned allocation? In Table 2, we report the overall performance on LiveBench, because OSCA's training data is a combination of 4 subtasks from LiveBench – math, reasoning, language and data analysis. To evaluate the generality of the learned allocation, we examine its performance on the 4 tasks separately. As Table 3 shows, OSCA is the best in 7/12 cases and the second best in the remaining 5. This indicates that the OSCA's learned allocation is domain-specific to some extent. Example problems of these 4 categories can be found in Appendix A.3.

		pass@C with varying $C \uparrow$		
		2^3	2^5	2^7
T1	Default pure	53.7	60.3	65.2
	Optimal pure	+7.0	+9.1	+12.4
	Uniform mixed	+2.5	+8.1	+9.7
	Learned mixed	**+10.3**	**+11.4**	**+14.0**
T2	Default pure	70.1	82.5	87.5
	Optimal pure	+6.0	+5.4	+6.2
	Uniform mixed	+1.4	**+10.6**	**+9.7**
	Learned mixed	**+12.1**	+10.1	+8.8
T3	Default pure	47.4	54.5	59.9
	Optimal pure	+8.9	+15.1	**+17.8**
	Uniform mixed	+2.5	+10.0	+12.6
	Learned mixed	**+14.1**	+16.1	+16.2
T4	Default pure	52.5	53.2	53.9
	Optimal pure	**+9.8**	+16.2	+20.0
	Uniform mixed	+6.4	+15.4	+20.6
	Learned mixed	+9.3	**+16.3**	**+21.9**

Table 3: Accuracy of different allocations on subtasks of LiveBench – math (T1), reasoning (T2), language (T3), data analysis (T4).

Can we learn the best configuration at instance level? Ideally, there is no need to use mixed allocation of sampling compute because there is only one optimal configuration for a specific problem. Therefore, we investigate whether it is possible to learn instance-level optimal configuration. We conduct this experiment on LiveCodeBench, as it has a larger training set and more homogeneous problems and is thus easier to learn. We propose a k-nearest neighbor algorithm under the assumption that similar problems need similar sampling configurations. For each test problem x, we retrieve the k

problems in the training set that are most similar to x, and allocate compute according to the distribution of optimal configuration over these problems. We use OpenAI's `text-embedding-3-large` to create semantic embeddings and compute the similarity between problems. As shown in Table 4, OSCA outperforms the problem-specific kNN allocation of compute, no matter what value k is. This suggests that it is not straightforward to learn problem-specific sampling configurations, justifying the need for a domain-specific allocation.

	pass@C with varying $C \uparrow$			
	2^1	2^3	2^5	2^7
$k = 4$	65.7	73.7	80.0	83.9
$k = 8$	66.9	74.5	79.9	82.9
$k = 16$	68.8	76.6	83.2	86.1
$k = 32$	68.2	77.3	83.0	86.6
$k = 64$	68.0	77.2	83.4	86.2
OSCA (ours)	**72.0**	**79.1**	**84.4**	**86.7**

Table 4: Accuracy of problem-specific sample allocation computed using k-nearest neighbor under different values of k is consistently lower than OSCA, especially when the compute budget is small.

5 Evaluation on Agentic Tasks

Optimizing sample compute allocation is not just useful for its own purpose, it is also useful for more complicated LLM inference algorithms, such as tree search and agentic workflows, because sampling is a basic operation in these. To demonstrate such usefulness, we apply OSCA to an agentic workflow for SWE-Bench, which needs 2 stages and 7 steps involving numerous LLM calls to resolve software engineering issues in repository-level code.

5.1 Benchmark and Workflow

SWE-Bench (Jimenez et al.) collects real-world software engineering issues from open-source GitHub repositories such as django and matplotlib. Each issue is paired with human-written unit tests. To resolve an issue, one needs to understand the issue description, examine the codebase (often hundreds of thousands of lines long), locate where to make changes and make necessary modifications by generating a patch. Decent solutions on this benchmark often takes an agentic approach by giving an LLM multiple tools to use and multiple actions to take and guiding it through a multi-stage workflow. We consider a subset of SWE-Bench curated by the authors called SWE-Bench Lite, which contains 300 issues.

Agentless (Xia et al., 2024) is one of the best-performing open-source solutions to SWE-Bench. An overview of how Agentless works can be found in Appendix A.5 Figure 6. It needs two stages – bug localization and bug repairing – to resolve a software engineering issue. There are 7 steps in total and 4 of them needs LLM sampling. We apply OSCA to one crucial step in the workflow – generating patches. Since SWE-Bench is really expensive to evaluate, we set both C_0 and C to be a smaller value 16.

Implementation Details. We use the same baselines from single-turn tasks – default pure allocation, optimal pure allocation, and uniform mixed allocation. We randomly sample 150 from the 300 issues as OSCA's training set, and use the remaining ones for testing. We consider three dimensions in sampling configuration – model choice, temperature, and prompting. The model choices we consider are *gpt-4o-2024-05-13*, *deepseek-coder-2.5*, and *qwen-2.5-70B*. The temperatures we consider are 0.4, 0.8, 1.2, and 1.6. The prompts are the intermediate results generated from the earlier stages of the workflow. We consider the two sets of intermediate results provided by the authors of Agentless[1]. They contain the relevant contexts in code files that the system finds relevant to the specific issue descriptions.

Both the default and optimal settings are gpt-4o, temperature 0.8 and prompt set 1.

5.2 Results

As demonstrated in Figure 1, OSCA can also improve the performance of Agentless by improving one of the steps in its agentic workflow. Compared to both the pure allocation (Temperature is set to 1 in both default and optimal pure allocation) and uniform mixed allocation, OSCA's learned allocation can get a similar accuracy with fewer samples. The gap between OSCA and the optimal pure allocation enlarges as sample compute gets larger, while uniform mixed allocation approaches and outperforms optimal pure with larger sample sizes.

These findings further demonstrate the necessity of optimizing sample budget allocation, as it can

[1]The prompt sets can downloaded at https://github.com/OpenAutoCoder/Agentless/releases/tag/v0.1.0. Prompt set 1 is in results/location_merged/loc_merged_0-1_outputs.jsonl. Prompt set 2 is in results/location_merged/loc_merged_2-3_outputs.jsonl.

be used in more complicated workflows to make them more efficient.

6 Conclusion

In conclusion, this paper presents OSCA, an algorithm designed to optimize sample compute allocation for large language models (LLMs) during inference. Through various experiments on both single-turn and agentic tasks, the study demonstrates that OSCA significantly improves accuracy with reduced compute resources compared to traditional methods, such as pure or uniform mixed allocations. By leveraging a mixed allocation allocation, OSCA balances different inference configurations, proving particularly effective in code generation and reasoning tasks, as seen in benchmarks like LiveCodeBench and LiveBench. This work demonstrates the importance of adapting sampling allocation to the specific characteristics of the problem at hand. Furthermore, OSCA's application to more complex workflows, such as multi-step agentic tasks, shows potential for broader utility in improving the efficiency of LLMs in various real-world applications. However, future work could explore optimizing additional hyperparameters and testing the scalability of the method with larger compute budgets.

Limitation

Although OSCA demonstrates an effective way to allocate sample compute there are still several limitations. First, this paper mainly focuses on four representative inference hyperparameters: model types, temperatures, response languages, and prompts. In addition to these aspects, there are other hyperparameters such as top k, top p, repetition penalty, etc. Combining these hyperparameters can make the sample configuration set more diverse. Besides, due to the computation limitation, we limited our inference compute budget to 512. It would be interesting to see how further scaling up will affect the performance.

Acknowledgements

L.L. is partly supported by a gift from Alpha Design AI. The views expressed are those of the authors and do not reflect the official policy or position of the funding agencies. We thank Terry Zhuo for his support in running experiments on BigCodeBench. We are also grateful for Alex Gu and Yangzhen Wu's suggestions in discussions.

Ethics Statement

We acknowledge that there might be some ethical considerations in enhancing LLMs' capability such as the OSCA presented in this paper. However, we believe that none must be specifically highlighted here.

References

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv e-prints*, pages arXiv–2207.

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

Yizhou Chi, Kevin Yang, and Dan Klein. 2024. Thoughtsculpt: Reasoning with intermediate revision and search. *arXiv preprint arXiv:2404.05966*.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. 2024. Learning How Hard to Think: Input-Adaptive Allocation of LM Computation.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. 2019. Universal transformers. In *International Conference on Learning Representations*.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A Saurous, Jascha Sohl-Dickstein, et al. 2022. Language model cascades. *arXiv preprint arXiv:2207.10342*.

Alex Graves. 2016. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*.

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*.

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*.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. LLM-Blender: Ensembling Large Language Models with Pairwise Ranking and Generative Fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14165–14178, Toronto, Canada. Association for Computational Linguistics.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349.

Rémi Leblond et al. 2023. Alphacode 2 technical report. Technical report, DeepMind.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Pearson.

Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen-tau Yih, Jason Weston, Jürgen Schmidhuber, and Xian Li. 2023. Large language model programs. *arXiv preprint arXiv:2305.05364*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters.

Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. 2024. Toward self-improvement of llms via imagination, searching, and criticizing. *Preprint*, arXiv:2404.12253.

Chi Wang, Susan Xueqing Liu, and Ahmed H. Awadallah. 2023a. Cost-effective hyperparameter optimization for large language model generation inference.

Chi Wang, Xueqing Liu, and Ahmed Hassan Awadallah. 2023b. Cost-effective hyperparameter optimization for large language model generation inference. In *International Conference on Automated Machine Learning*, pages 21–1. PMLR.

Siyin Wang, Shimin Li, Tianxiang Sun, Jinlan Fu, Qinyuan Cheng, Jiasheng Ye, Junjie Ye, Xipeng Qiu, Xuanjing Huang, Lun-Wei Ku, Andre Martins, and Vivek Srikumar. 2024. Llm can achieve self-regulation via hyperparameter aware generation. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 6632–6646. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023c. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Sean Welleck, Amanda Bertsch, Matthew Finlayson, Hailey Schoelkopf, Alex Xie, Graham Neubig, Ilia Kulikov, and Zaid Harchaoui. 2024. From decoding to meta-generation: Inference-time algorithms for large language models. *arXiv preprint arXiv:2406.16838*.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, et al. 2024. Livebench: A challenging, contamination-free llm benchmark. *arXiv preprint arXiv:2406.19314*.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems*, 36:54769–54784.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Appendix

A.1 Proof of Convexity

We aim to minimize the following objective function:

$$\min_{\pi} O(\pi) = \sum_{j=1}^{m} \prod_{i=1}^{n} (1 - p_{ij})^{\pi_i}$$

subject to:

- $0 \leq \pi_i \leq C$, with $\pi_i \in \mathbb{N}$
- $\sum_{i=1}^{n} \pi_i = C$

To simplify the analysis, we take the negative logarithm of the objective function, which preserves convexity properties:

$$O(\pi) = \sum_{j=1}^{m} \exp\left(\sum_{i=1}^{n} \pi_i \ln(1 - p_{ij}) \right)$$

Define:

$$g_j(\pi) = \sum_{i=1}^{n} \pi_i \ln(1 - p_{ij})$$

Since $g_j(\pi$ is a linear combination of π_i, it is an affine function in π. As affine functions are both convex and concave, $g_j(\pi)$ is convex.

The exponential function, $\exp(z)$, is convex. Since the composition of a convex function with an affine function remains convex, it follows that:

$$h_j(\pi) = \exp(g_j(\pi))$$

is convex in π.

Thus, the overall objective function:

$$\begin{aligned} f(\pi) &= \sum_{j=1}^{m} h_j(\pi) \\ &= \sum_{j=1}^{m} \exp\left(\sum_{i=1}^{n} \pi_i \ln(1 - p_{ij}) \right) \end{aligned}$$

is a sum of convex functions, implying that $f(\pi)$ is convex.

□

A.2 Example Problems from LiveCodeBench

Codeforces: Short Sort

Problem Description: There are three cards with letters a, b, and c placed in a row in some order. You can do the following operation at most once:

- Pick two cards and swap them.

Is it possible that the row becomes abc after the operation? Output YES if it is possible, and NO otherwise.

Input:

The first line contains a single integer t ($1 \leq t \leq 6$) — the number of test cases.

The only line of each test case contains a string consisting of the characters a, b, and c exactly once, representing the cards.

Output:

For each test case, output YES if you can make the row abc with at most one operation, or NO otherwise.

Example:

```
Input:
6
abc
acb
bac
bca
cab
cba
Output:
YES
YES
YES
NO
NO
YES
```

Python Solution:

```
def is_possible_to_sort_to_abc(s):
    if s == "abc":
        return "YES"
   if s == "acb" or s == "bac" or s == "cba":
        return "YES"
    return "NO"
t = int(input())
for _ in range(t):
    s = input().strip()
    print(is_possible_to_sort_to_abc(s))
```

C++ Solution:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    int t;
    cin >> t;
    while (t--) {
        string s;
        cin >> s;
        if (s == "abc" || s == "acb"
        || s == "bac" || s == "cba") {
            cout << "YES" << endl;
        } else {
            cout << "NO" << endl;
        }
    }
    return 0;
}
```

LeetCode: Number of Senior Citizens

Problem Description: You are given a 0-indexed array of strings `details`. Each element provides information about a passenger compressed into a string of length 15. The format of the string is:

- The first 10 characters consist of the phone number of the passengers.
- The next character denotes the gender of the person.
- The following two characters indicate the age of the person.
- The last two characters determine the seat allotted to that person.

Return the number of passengers who are strictly more than 60 years old.
Example:

```
Input: details = ["7868190130M7522",
                  "5303914400F9211",
                  "9273338290F4010"]
Output: 2

Explanation: The passengers at indices 0, 1,
and 2 have ages 75, 92, and 40. Thus, there
are 2 people who are over 60 years old.
```

Python Solution:

```
class Solution(object):
    def countSeniors(self, details):
        count = 0
        for detail in details:
            age = int(detail[11:13])
            if age > 60:
                count += 1
        return count
```

C++ Solution:

```
class Solution {
public:
   int countSeniors(vector<string>& details) {
        int count = 0;
      for (const string& detail : details) {
         int age = stoi(detail.substr(11, 2));
            if (age > 60) {
                count++;
            }
        }
        return count;
    }
};
```

AtCoder: Wrong Answer

Problem Description: You are given two integers A and B, each between 0 and 9, inclusive. Print any integer between 0 and 9, inclusive, that is not equal to $A + B$.
Input:
The input consists of two integers A and B.
Output:
Print any integer between 0 and 9, inclusive, that is not equal to $A + B$.
Example:

```
Input:
2 5

Output:
2

Input:
0 0

Output:
9

Input:
7 1

Output:
4
```

Python Solution:

```
A, B = map(int, input().split())
S = A + B
for i in range(10):
    if i != S:
        print(i)
        break
```

C++ Solution:

```
#include <iostream>
using namespace std;

int main() {
    int A, B;
    cin >> A >> B;
    int sum = A + B;
    for (int i = 0; i <= 9; ++i) {
        if (i != sum) {
            cout << i << endl;
            break;
        }
    }
    return 0;
}
```

A.3 Example Problems from LiveBench

Math Problem

Real numbers x and y with $x, y > 1$ satisfy $\log_x(y^x) = \log_y(x^4y) = 10$. What is the value of xy? Please think step by step, and display the answer at the very end of your response. The answer is an integer consisting of exactly 3 digits (including leading zeros), ranging from 000 to 999, inclusive.

Ground Truth: 025

Reasoning Problem

In this question, assume each person either always tells the truth or always lies. Tala is at the movie theater. The person at the restaurant says the person at the aquarium lies. Ayaan is at the aquarium. Ryan is at the botanical garden. The person at the park says the person at the art gallery lies. The person at the museum tells the truth. Zara is at the museum. Jake is at the art gallery. The person at the art gallery says the person at the theater lies. Beatriz is at the park. The person at the movie theater says the person at the train station lies. Nadia is at the campground. The person at the campground says the person at the art gallery tells the truth. The person at the theater lies. The person at the amusement park says the person at the aquarium tells the truth. Grace is at the restaurant. The person at the aquarium thinks their friend is lying. Nia is at the theater. Kehinde is at the train station. The person at the theater thinks their friend is lying. The person at the botanical garden says the person at the train station tells the truth. The person at the aquarium says the person at the campground tells the truth. The person at the aquarium saw a firetruck. The person at the train station says the person at the amusement park lies. Mateo is at the amusement park. Does the person at the train station tell the truth? Does the person at the amusement park tell the truth? Does the person at the aquarium tell the truth? Think step by step, and then put your answer in **bold** as a list of three words, yes or no (for example, **yes, no, yes**. If you don't know, guess.

Ground Truth: no, yes, yes

Language Problem

You are given 8 words/phrases below. Find two groups of four items that share something in common. Here are a few examples of groups: bass, flounder, salmon, trout (all four are fish); ant, drill, island, opal (all four are two-word phrases that start with 'fire'); are, why, bee, queue (all four are homophones of letters); sea, sister, sin, wonder (all four are members of a septet). Categories will be more specific than e.g., '5-letter-words', 'names', or 'verbs'. There is exactly one solution. Think step-by-step, and then give your answer in **bold** as a list of the 8 items separated by commas, ordered by group (for example, **bass, founder, salmon, trout, ant, drill, island, opal**). If you don't know the answer, make your best guess. The items are: row, drift, curl, tide, current, press, fly, wave.

Ground Truth: current, drift, tide, wave, curl, fly, press, row

Example questions from the Data Analysis category can be lengthy, so examples can be viewed here.

A.4 Mathematical Formulas for Evaluating Results with Fractional Scores

Probability Mass Function (PMF) for the Maximum Score When Sampling k Scores

To calculate the probability that the maximum score X_{max} among k samples is exactly x:

$$P(X_{\text{max}} = x) = \frac{\binom{c_{\leq x}}{k} - \binom{c_{<x}}{k}}{\binom{m}{k}}$$

Where:

- $m = \sum_x c_x$ is the total sample size.
- $c_{\leq x} = \sum_{y \leq x} c_y$ is the cumulative count of scores less than or equal to x.
- $c_{<x} = \sum_{y<x} c_y$ is the cumulative count of scores strictly less than x.

Expected Maximum Score Across Multiple Settings with Known PMF

The expected value $E[X]$ of the maximum score across multiple settings is:

$$E[X] = \sum_x x \cdot P(X = x)$$

Where $P(X = x)$ is computed as:

$$P(X = x) = \prod_{j=1}^{s} P_j(X_j \leq x) - \prod_{j=1}^{s} P_j(X_j < x)$$

Here, $P_j(X_j \leq x)$ represents the probability that the maximum score in setting j is less than or equal to x. The difference between the products isolates the probability that the maximum score is exactly x.

Expected Maximum Score with Excess Samples

The expected value of the maximum score when sampling n times is estimated by:

$$E[X] = \sum_x (x \cdot [(c + f_x)^n - c^n])$$

Where:

- n is the number of samples,
- f_x is the probability density for score x,
- c is the cumulative probability up to score x.

The term $(c + f_x)^n - c^n$ reflects the probability of selecting exactly x as the maximum score from n samples.

A.5 Agentless Workflow on SWE-Bench

Figure 6 is the overview of Agentless taken from their paper (Xia et al., 2024).

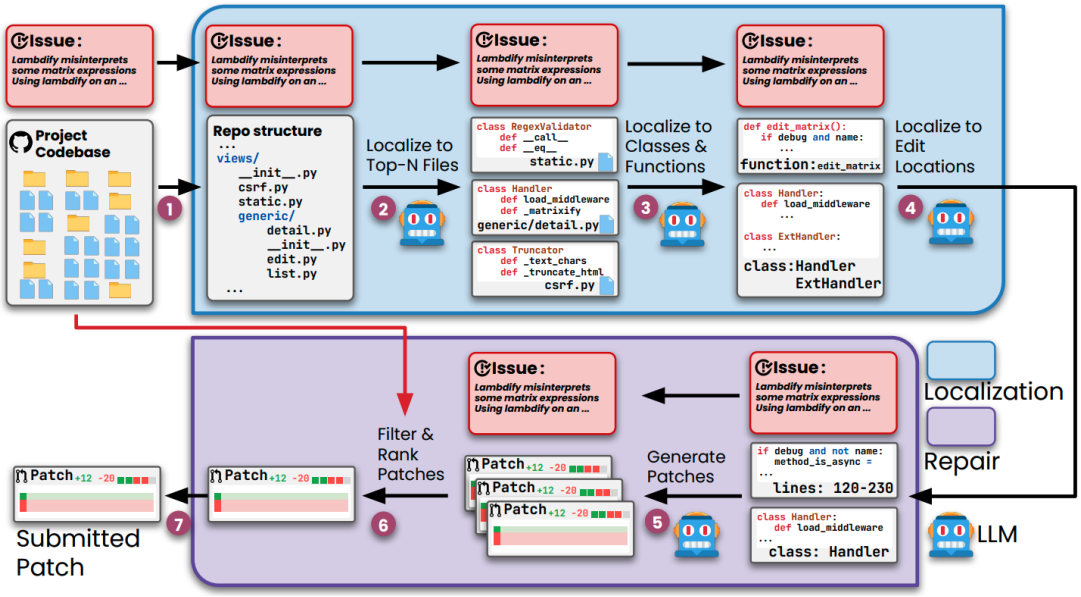

Figure 6: Overview of Agentless, directly taken from their paper (Xia et al., 2024).